Jakarta Server Pages (JSP)

Jakarta Server Pages (JSP) is a technology used to create dynamic web content. JSP is part of the Jakarta EE platform, which allows developers to write server-side code that can generate HTML, XML, or other types of documents in response to client requests. Here's a structured path to learn JSP:

What is JSP?

Jakarta Server Pages (JSP) is a technology that allows for the creation of dynamic, server-side web content. It is part of the Jakarta EE (formerly Java EE) platform, which provides a robust and scalable environment for developing enterprise-level web applications. JSP pages are essentially HTML pages embedded with Java code, enabling developers to build web pages that can dynamically generate content based on user interactions, database queries, or other server-side processes.

Key Features of JSP

1. Dynamic Content Generation:

- **JSP allows embedding Java code directly into HTML using special tags. This makes it possible to generate dynamic web content based on various inputs and conditions.**

2. Ease of Development:

- **JSP simplifies web application development by allowing the use of Java programming language within HTML pages. This reduces the need to write complex servlets for generating HTML.**

3. Separation of Concerns:

- **By using JSP for the presentation layer and servlets for business logic, developers can separate the concerns of web application development, leading to a cleaner and more manageable codebase.**

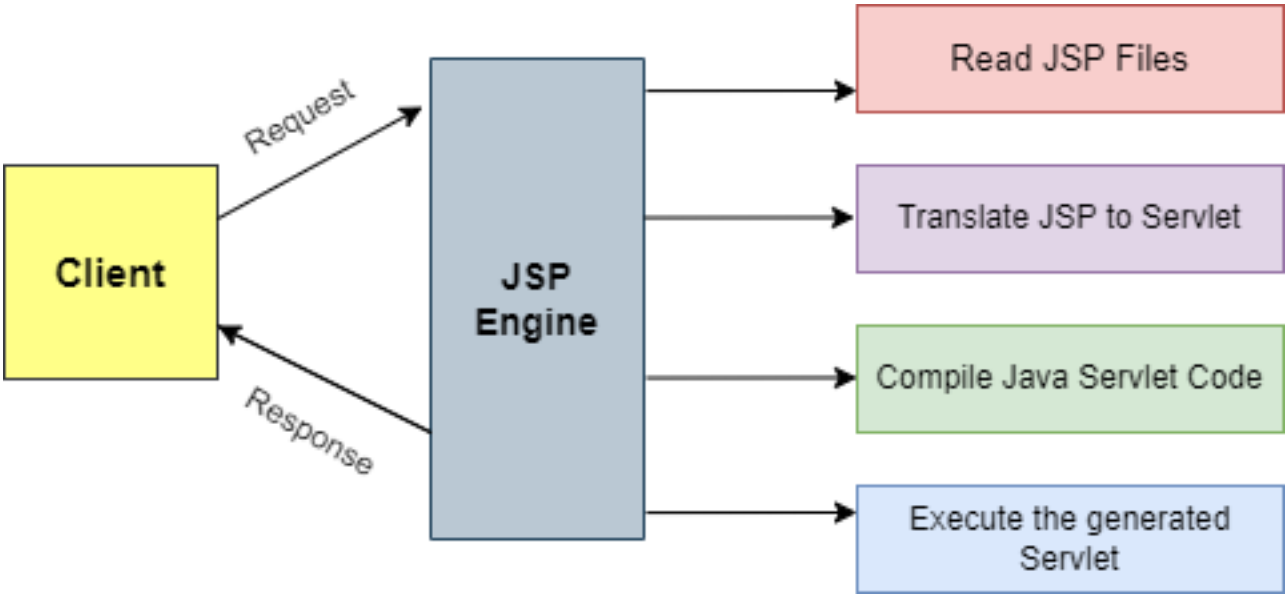
4. Reusability:

- **JSP supports the use of reusable components like JavaBeans, custom tags, and tag libraries, which can be used across multiple JSP pages.**

5. Integration with Servlets:

- **JSP pages are compiled into servlets by the server. This means they have all the capabilities of servlets and can interact seamlessly with other servlets and Java classes.**

JSP Architecture



JSP works on the request-response model of the web:

1. Client Request:

- **A user sends a request to a JSP page via a web browser.**

2. JSP Processing:

- **The web server processes the JSP page. If it is the first request, the JSP page is compiled into a servlet. The servlet then processes the request.**

3. Response Generation:

- **The servlet generates the dynamic content, which is typically HTML, and sends it back to the client's web browser.**

Basic Syntax

JSP pages use special tags to embed Java code into HTML:

1) **Directives: `<%@ directive %>`**
   - **For example, `<%@ page language="java" %>`**
2) **Scriptlets: `<% java code %>`**
   - **For example, `<% out.println("Hello, World!"); %>`**
3) **Expressions: `<%= expression %>`**
   - **For example, `<%= request.getParameter("name") %>`**
4) **Declarations: `<%! declaration %>`**
   - **For example, `<%! int counter = 0; %>`**

Example of a Simple JSP Page

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html><head><title>My First JSP Page</title></head>
<body><h1>Welcome to JSP!</h1>
<p>The current date and time is: <%= new java.util.Date() %></p>
</body></html>
```

In this example, the JSP page outputs the current date and time by embedding Java code within the HTML content.

Advantages of Using JSP

- **Simplifies Web Development: Easier to write and understand compared to servlets for generating HTML.**
- **Powerful Integration: Combines the power of Java with the flexibility of HTML.**
- **Reusable Components: Supports the use of JavaBeans and custom tag libraries.**
- **Efficient Maintenance: Separation of business logic and presentation layer improves code maintainability.**

Understand the basic concept of JSP and how it fits into the Jakarta EE ecosystem.

How JSP Fits into the Jakarta EE Ecosystem

Jakarta EE is a comprehensive suite of APIs and services for building robust, scalable, and secure enterprise applications. JSP is one of the technologies provided by Jakarta EE for the web layer of applications.

Integration with Other Jakarta EE Components:

1. Servlets:

- **JSP pages are translated into servlets by the web container. This tight integration allows JSP to take advantage of all the features provided by the servlet API, such as session management, request dispatching, and lifecycle management.**

2. JavaBeans:

- **JSP can interact with JavaBeans components to encapsulate business logic. JavaBeans can be used to retrieve data from a database, perform computations, and more. JSP can then use this data to generate the final web page content.**

3. Expression Language (EL):

- **JSP supports the use of EL, which simplifies the process of accessing data stored in JavaBeans, request parameters, session attributes, and more. EL helps in making JSP code cleaner and easier to read.**

4. JSP Standard Tag Library (JSTL):

- **JSTL provides a set of tags for common tasks like iteration, conditionals, and database access. This allows developers to write less Java code within JSP pages and use standard tags instead.**

5. Custom Tags:

- **Developers can create custom tag libraries to encapsulate reusable logic. This further enhances the separation of presentation and business logic and promotes code reuse.**

Example Workflow in Jakarta EE with JSP

1. Client Request:

- **A client (typically a web browser) sends a request to the server for a JSP page.**

2. Request Handling:

- **The web server forwards this request to the JSP engine. If the JSP page is requested for the first time, it is compiled into a servlet.**

3. Processing:

- **The servlet (generated from the JSP) processes the request. It can interact with JavaBeans, perform business logic, and use JSTL or custom tags.**

4. Response Generation:

- **The servlet generates dynamic HTML content and sends it back to the client.**

5. Client Display:

- **The client's web browser displays the generated HTML content to the user.**

Jakarta Server Pages (JSP) and servlets are both server-side technologies used in the Jakarta EE platform to create dynamic web applications. While they share similarities and can often be used interchangeably, they have distinct roles and advantages that make them suitable for different aspects of web development. Understanding the differences and how JSP complements servlets is essential for building robust web applications.

Differences between JSP and Servlets

1. Syntax and Code Structure

-Servlets:

- **Java Code: Servlets are written entirely in Java. They extend the `HttpServlet` class and override methods like `doGet` and `doPost` to handle HTTP requests and responses.**
- **HTML within Java: Generating HTML content within servlets requires embedding HTML code within Java code using methods like `response.getWriter().println()`.**

- JSP:

- **HTML Code with Java: JSP pages are primarily HTML with embedded Java code. This makes them more readable and easier to write for web designers who are more familiar with HTML than Java.**
- **Tags and Scriptlets: JSP uses special tags (`<% %>`, `<%= %>`, etc.) to embed Java code within HTML.**

2. Use Cases

- Servlets:

- **Control Logic: Servlets are ideal for handling business logic and processing requests. They act as controllers in the MVC (Model-View-Controller) pattern, managing the application's flow and interactions.**
- **Data Processing: Suitable for tasks like form submission handling, interacting with databases, and performing backend computations.**

- JSP:

- **Presentation Layer: JSP is designed for creating the presentation layer of web applications. It is used to generate the dynamic content that is sent to the client's web browser.**
- **Templating: Ideal for generating dynamic HTML, XML, or other types of documents.**

3. Compilation and Lifecycle

- Servlets:

- **Direct Compilation: Servlets are compiled directly into bytecode and executed by the servlet container.**
- **Lifecycle: Lifecycle methods include `init()`, `service()`, and `destroy()`, providing fine-grained control over how the servlet handles requests.**

- JSP:

- **JSP to Servlet Translation: JSP pages are first translated into servlets by the JSP engine. This servlet is then compiled and executed. This process is typically transparent to the developer.**
- **Simplified Lifecycle: JSP pages do not require explicit lifecycle management. The container handles the translation, compilation, and execution.**

How JSP Complements Servlets

JSP and servlets complement each other by allowing developers to separate the concerns of business logic and presentation. This separation leads to cleaner, more maintainable code.

1. Model-View-Controller (MVC) Pattern

- Controller (Servlet):

- **Handles user requests, processes inputs, and interacts with the model (business logic).**
- **Decides which view (JSP page) to render based on the processed data.**

- View (JSP):

- **Retrieves data set by the servlet and presents it to the user.**
- **Contains minimal business logic, focusing instead on displaying content.**

2. Reusability and Maintainability

- Separation of Concerns:

- **By using servlets for business logic and JSP for the presentation layer, developers can more easily update and maintain their code. For instance, changes to business logic do not affect the presentation layer and vice versa.**

- Reusable Components:

- **JSP can use reusable components like JavaBeans and custom tags, further promoting modularity and reuse.**

3. Efficiency

- Reduced Complexity:

- **JSP reduces the complexity of HTML generation within Java code, making it easier to develop and maintain web pages.**
- **Servlets handle complex processing and data manipulation, allowing JSP to focus solely on presenting the data.**

---

2. Setting Up the Environment

Add Jakarta servlet jsp into pom.xml file

```
<!-- https://mvnrepository.com/artifact/jakarta.servlet.jsp/jakarta.servlet.jsp-api -->
<dependency>
    <groupId>jakarta.servlet.jsp</groupId>
    <artifactId>jakarta.servlet.jsp-api</artifactId>
    <version>4.0.0</version>
    <scope>provided</scope></dependency>
```

Lifecycle Of Jakarta Server Page (JSP)

The lifecycle of a Jakarta Server Page (JSP) involves several phases that ensure the JSP is processed correctly and efficiently. Understanding these phases helps developers debug and optimize their JSP pages. The lifecycle phases include translation, compilation, initialization, execution, and cleanup.

Phases of JSP Lifecycle

1. Translation Phase

- **What Happens: The JSP engine translates the JSP file into a servlet source file.**
- **Details: The JSP engine parses the JSP file, converting the JSP tags, scriptlets, expressions, and other JSP elements into corresponding Java code within a servlet class. This phase occurs the first time a JSP is requested or when the JSP file has changed.**

2. Compilation Phase

- **What Happens: The translated servlet source file is compiled into a Java bytecode class.**
- **Details: The servlet source file generated in the translation phase is compiled by the JSP engine into a servlet class file (bytecode). This compiled class is then loaded into the servlet container.**

3. Initialization Phase

- **What Happens: The servlet instance (generated from the JSP) is initialized.**
- **Details: The servlet container initializes the JSP servlet by invoking its `jspInit()` method. This method is called only once, similar to the `init()` method in a regular servlet. Any initialization code, such as resource allocation, can be placed here.**

4. Execution Phase

- **What Happens: The servlet processes client requests.**
- **Details: Each client request triggers the execution of the servlet's `jspService()` method. This method corresponds to the `service()` method in a regular servlet and handles `GET`, `POST`, or other HTTP requests. The method generates the dynamic content (e.g., HTML) and sends it back to the client's browser. The `jspService()` method is invoked for every request to the JSP page.**

5. Cleanup Phase

- **What Happens: The servlet instance is destroyed.**
- **Details: When the servlet container decides to remove the JSP servlet instance (e.g., during a shutdown or redeployment), it calls the `jspDestroy()` method. This method is used for any cleanup code, such as releasing resources or closing connections.**

Lifecycle Methods in JSP

- **jspInit(): Called once during the initialization phase.**
- **jspService(HttpServletRequest request, HttpServletResponse response): Called for each client request during the execution phase.**
- **jspDestroy(): Called once during the cleanup phase.**

Example Lifecycle in Action

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

<html><head><title>JSP Lifecycle Example</title></head><body>

  <h1>Welcome to JSP Lifecycle Example</h1>

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

</body></html>
```

1. Translation Phase:

- **The JSP engine parses the above JSP and translates it into a servlet source file, e.g., `JspLifecycleExample_jsp.java`.**

2. Compilation Phase:

- **The translated servlet source file is compiled into a servlet class file, e.g., `JspLifecycleExample_jsp.class`.**

3. Initialization Phase:

- **The servlet container initializes the servlet by invoking the `jspInit()` method. If you need to include custom initialization code, you can override this method in the servlet.**

4. Execution Phase:

- **Each request to the JSP page triggers the `jspService()` method, which handles the HTTP request and generates the HTML content dynamically. The current date and time are included in the response.**

5. Cleanup Phase:

- **When the servlet container decides to unload the servlet, the `jspDestroy()` method is called. This method can be overridden to include cleanup code.**

Recompilation: If the JSP file is modified, the JSP engine re-triggers the translation and compilation phases to reflect the changes.

Thread Safety: The `jspService()` method must be thread-safe since it can handle multiple requests simultaneously.

Resource Management: Proper resource management in `jspInit()` and `jspDestroy()` methods is crucial to avoid memory leaks and other resource-related issues.

JSP Tags and Directives

https://jakarta.ee/specifications/pages/4.0/

https://jakarta.ee/specifications/pages/4.0/jakarta-server-pages-spec-4.0

JSP (Jakarta Server Pages) uses tags and directives to provide a powerful and flexible way to include dynamic content and control the behavior of the JSP page.

JSP Directives

JSP directives provide global information about the entire JSP page and are used to set page-level instructions. Directives are enclosed within `<%@ %>` tags.

1. `<%@ page %>` Directive

The `<%@ page %>` directive defines attributes that apply to the entire JSP page, such as the scripting language, content type, and buffer size.

Common Attributes:

- **language: Specifies the scripting language used in the JSP page. The default is `"java"`.**

  <%@ page language="java" %>

- **contentType: Sets the MIME type and character encoding of the response. The default is `"text/html;charset=ISO-8859-1"`.**

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

- **import: Imports Java classes or packages for use in the JSP page, similar to the `import` statement in Java.**

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

- **session: Specifies whether the JSP page participates in an HTTP session. The default is `true`.**

  <%@ page session="true" %>

- **buffer: Sets the buffer size for the JSP output. The default is `"8kb"`.**

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

- **autoFlush: Determines whether the buffer is automatically flushed when full. The default is `true`.**

  <%@ page autoFlush="true" %>

- **isThreadSafe: Indicates whether the JSP page is thread-safe. The default is `true`.**

  <%@ page isThreadSafe="true" %>

- **errorPage: Specifies the URL of another JSP page to handle exceptions.**

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

- **isErrorPage: Indicates whether the current JSP page is an error page. The default is `false`.**

  <%@ page isErrorPage="false" %>

Example:

```
<%@ page language="java" contentType="text/html;charset=UTF-8" import="java.util.*" %>
<html><head> <title>JSP Page Directive Example</title></head>
<body> <h1>Current Date and Time: <%= new Date() %></h1>
</body></html>
```

2. `<%@ include %>` Directive

The `<%@ include %>` directive statically includes the content of another file during the translation phase. This is different from the `<jsp:include>` action, which includes content at runtime.

<%@ include file="relativeURL" %>

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

<html><body><h1>Welcome to the main page!</h1>

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

</body></html>
```

In this example, the content of `header.jsp` and `footer.jsp` will be included in the main JSP page during translation.

3. `<%@ taglib %>` Directive

The `<%@ taglib %>` directive declares a tag library that contains custom tags, which can be used in the JSP page. Custom tags extend the functionality of JSP by allowing developers to create reusable components.

Attributes:

- **uri: Specifies the URI that uniquely identifies the tag library.**
- **prefix: Sets a prefix that distinguishes the custom tags in the JSP page.**

Example:

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

<html><body><h1>JSTL Example</h1>

  <c:out value="Hello, World!" />

</body></html>
```

JSP directives are powerful tools that help control various aspects of the JSP page. The three primary directives are:

1. `<%@ page %>`: Defines page-level settings and attributes.

2. `<%@ include %>`: Includes static content from another file during the translation phase.

3. `<%@ taglib %>`: Declares custom tag libraries for use within the JSP page.

Scriptlets, Declarations, And Expressions

In JSP (Jakarta Server Pages), scriptlets, declarations, and expressions are used to embed Java code directly within the HTML markup. Each of these elements serves a distinct purpose:

Scriptlets (`<% %>`)

Scriptlets are used to embed arbitrary Java code within the JSP page. The code inside a scriptlet is inserted into the `_jspService` method, which is called for each request.

```
<%@ page import="java.util.*" %>
<html><head><title>Scriptlet Example</title></head>
<body><h1>Scriptlet Example</h1>
  <%
    Date now = new Date();
    out.println("Current Date and Time: " + now.toString());
  %>
</body></html>
```

In this example, the scriptlet initializes a `Date` object and prints the current date and time using `out.println()`.

Declarations (`<%! %>`)

Declarations are used to define variables and methods at the class level, which means they become instance variables or methods of the servlet that the JSP is converted into. Declarations are not executed within the `_jspService` method but as part of the servlet class.

```
<%@ page import="java.util.*" %>
<html><head><title>Declaration Example</title></head>
<body>
  <h1>Declaration Example</h1>
  <p>Current Date and Time: <%= getCurrentDate() %></p> </body>
<%!   public String getCurrentDate() { return new Date().toString(); }%>  </html>
```

In this example, the `getCurrentDate()` method is declared at the class level and can be used anywhere in the JSP page.

Expressions (`<%= %>`)

Expressions are used to output the value of a Java expression directly to the response. The result of the expression is converted to a string and included in the output stream at the location of the expression.

```
<%@ page import="java.util.*" %>
<html><head><title>Expression Example</title></head>
<body>
  <h1>Expression Example</h1>
  <p>Current Date and Time: <%= new Date() %></p>
</body></html>
```

In this example, the expression `<%= new Date() %>` is evaluated and the result (the current date and time) is output to the HTML response.

Scriptlets (`<% %>`): Embed arbitrary Java code within the JSP page, executed within the `_jspService` method.

Declarations (`<%! %>`): Define variables and methods at the class level, accessible throughout the JSP page.

Expressions (`<%= %>`): Output the result of a Java expression directly to the response.

it is recommended to minimize the use of scriptlets and declarations in favor of more modern approaches like JavaBeans, custom tags, and the JSP Expression Language (EL) for better separation of concerns and maintainability.

JSP Implicit Objects

JSP (Jakarta Server Pages) provides a set of implicit objects that are automatically available to developers without needing explicit declaration. These objects simplify the interaction with the underlying servlet API and facilitate common web development tasks.

Common JSP Implicit Objects

1. `request`

- **Type: `HttpServletRequest`**
- **Description: Represents the client's request. It provides methods to get request parameters, headers, attributes, and more.**

```
<html> <body><h1>Request Example</h1>
```

```
    <p>Client IP: <%= request.getRemoteAddr() %></p>
    <p>Request URI: <%= request.getRequestURI() %></p>
</body></html>
```

2. `response`

- **Type: `HttpServletResponse`**
- **Description: Represents the response sent to the client. It provides methods to set response headers, status codes, and manage output streams.**

```
<%
    response.setContentType("text/html;charset=UTF-8");
%>
<html><body>
    <h1>Response Example</h1>
</body></html>
```

3. `out`

- **Type: `JspWriter`**
- **Description: Used to send content to the client's browser. It provides methods to write text and manage the output buffer.**

```
<html> <body><h1>Out Example</h1>
    <p>Hello, <%= "World" %>!</p>
</body></html>
```

4. `session`

- **Type: `HttpSession`**
- **Description: Represents the session between the client and the server. It allows storing and retrieving session attributes.**

```
<% session.setAttribute("username", "JakartaUser"); %>
<html> <body> <h1>Session Example</h1>
    <p>Username: <%= session.getAttribute("username") %></p>
</body> </html>
```

5. `application`

- **Type: `ServletContext`**
- **Description: Represents the servlet context. It allows interaction with the web application as a whole, such as getting initialization parameters and managing application-wide attributes.**

```
<% application.setAttribute("appName", "MyWebApp"); %>

<html><body><h1>Application Example</h1>

  <p>Application Name: <%= application.getAttribute("appName") %></p>

</body></html>
```

6. `config`

- **Type: `ServletConfig`**
- **Description: Represents the servlet configuration. It provides initialization parameters and context information about the servlet.**

```
<html><body><h1>Config Example</h1>

  <p>Servlet Name: <%= config.getServletName() %></p>

</body> </html>
```

7. `pageContext`

- **Type: `PageContext`**
- **Description: Provides access to all the namespaces associated with a JSP page, including `request`, `session`, and `application`. It also provides methods for forwarding requests and handling errors.**

```
<html> <body> <h1>PageContext Example</h1>

  <p>Request URI: <%= pageContext.getRequest().getRequestURI() %></p>

</body>  </html>
```

8. `page`

- **Type: `Object`**
- **Description: Refers to the instance of the JSP page's servlet. It is rarely used in practice.**

```
<html> <body> <h1>Page Example</h1>

  <p>Page Object: <%= page.toString() %></p> </body>  </html>
```

9. `exception`

- **Type: `Throwable`**
- **Description: Represents the exception object in an error page. It is available only in JSP pages configured as error pages.**

```
<%@ page isErrorPage="true" %>
<html>  <body>  <h1>Exception Example</h1>
  <p>Error: <%= exception.getMessage() %></p>
</body>  </html>
```

JSP implicit objects provide a convenient way to access and manipulate request, response, session, application, and other data without needing explicit declarations. They streamline the development process by offering built-in functionality for common web application tasks.

---

6. JSP Actions

- Standard Actions
  - `<jsp:include>`, `<jsp:forward>`, `<jsp:param>`, etc.
- Custom Actions
  - How to create and use custom tags with the help of tag libraries.

7. Expression Language (EL)

- Using EL
  - Simplify access to data stored in JavaBeans components, request parameters, and other objects.

8. JSP Standard Tag Library (JSTL)

- Core Tags
  - Learn about core JSTL tags for iteration, conditionals, etc.
- Formatting Tags
  - Number formatting, date formatting.
- SQL Tags