**JSP**

JavaServer Pages (JSP) is a technology for developing web pages that support dynamic content which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, we can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages etc.

**Why Use JSP?**

JavaServer Pages often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But JSP offer several advantages in comparison with the CGI.

* Performance is significantly better because JSP allows embedding Dynamic Elements in HTML Pages itself instead of having a separate CGI files.
* JSP are always compiled before it's processed by the server unlike CGI/Perl which requires the server to load an interpreter and the target script each time the page is requested.
* JavaServer Pages are built on top of the Java Servlets API, so like Servlets, JSP also has access to all the powerful Enterprise Java APIs, including JDBC, JNDI, EJB, JAXP etc.
* JSP pages can be used in combination with servlets that handle the business logic, the model supported by Java servlet template engines.

Finally, JSP is an integral part of Java EE, a complete platform for enterprise class applications. This means that JSP can play a part in the simplest applications to the most complex and demanding.

**Advantages of JSP:**

Following is the list of other advantages of using JSP over other technologies:

* JSP vs. Active Server Pages (ASP): The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.
* JSP vs. Pure Servlets: It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.
* JSP vs. Server-Side Includes (SSI): SSI is really only intended for simple inclusions, not for "real" programs that use form data, make database connections, and the like.
* JSP vs. JavaScript: JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.
* JSP vs. Static HTML: Regular HTML, of course, cannot contain dynamic information.

**What are Servlets?**

Java Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

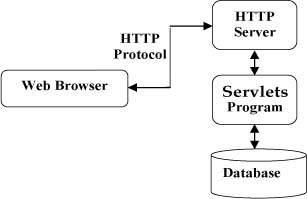
Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Java Servlets often serve the same purpose as programs implemented using the Common Gateway Interface (CGI). But Servlets offer several advantages in comparison with the CGI.

* Performance is significantly better.
* Servlets execute within the address space of a Web server. It is not necessary to create a separate process to handle each client request.
* Servlets are platform-independent because they are written in Java.
* Java security manager on the server enforces a set of restrictions to protect the resources on a server machine. So servlets are trusted.
* The full functionality of the Java class libraries is available to a servlet. It can communicate with applets, databases, or other software via the sockets and RMI mechanisms that you have seen already.

**Servlets Architecture:**

Following diagram shows the position of Servelts in a Web Application.



**Servlets Tasks:**

Servlets perform the following major tasks:

* Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
* Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
* Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
* Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
* Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

**Servlets Packages:**

Java Servlets are Java classes which service HTTP requests and implement the javax.servlet.Servlet interface.It is run by a web server that has an interpreter that supports the Java Servlet specification.

Servlets can be created using the javax.servlet and javax.servlet.http packages, which are a standard part of the Java's enterprise edition, an expanded version of the Java class library that supports large-scale development projects. These classes implement the Java Servlet and JSP specifications.

Java servlets have been created and compiled just like any other Java class. After installing the servlet packages and add them to computer's Class path, can compile servlets with the JDK's Java compiler or any other current compiler.

**Servlets - Life Cycle**

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet

* The servlet is initialized by calling the init () method.
* The servlet calls service() method to process a client's request.
* The servlet is terminated by calling the destroy() method.
* Finally, servlet is garbage collected by the garbage collector of the JVM.

**The init() method :**

The init method is designed to be called only once. It is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this:

public void init() throws ServletException

{ // Initialization code...}

**The service() method :**

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client( browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server produce a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method:

public void service(ServletRequest request,

ServletResponse response)

throws ServletException, IOException{

}

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

**The destroy() method :**

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this:s

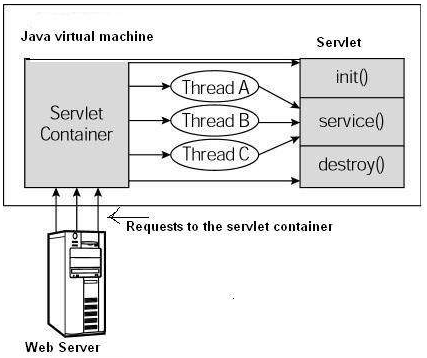
public void destroy()

{ // Finalization code... }

**Architecture Diagram:**

The following figure depicts a typical servlet life-cycle scenario.

* First the HTTP requests coming to the server are delegated to the servlet container.
* The servlet container loads the servlet before invoking the service() method.
* Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



**The servlet can be created by three ways:**

* By implementing Servlet interface,
* By inheriting GenericServlet class, (or)
* By inheriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

**Example of Servlet**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class HelloWorld extends HttpServlet {

private String message;

public void init() throws ServletException

{ // Do required initialization

message = "Hello World";

}

public void doGet(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException

{

response.setContentType("text/html");

PrintWriter out = response.getWriter();

out.println("<h1>" + message + "</h1>");

}

public void destroy()

{ // do nothing. }

}

**Compiling a Servlet:**

$ javac HelloWorld.java

If the servlet depends on any other libraries, you have to include those JAR files on your CLASSPATH as well.

**Servlet Deployment:**

For using apache tomcat server. The deployment steps are as follows:

* Create a directory structure
* Create and Compile the Servlet. Then, copy the servlet’s class file to the proper directory, and add the servlet’s name and mapping to the proper **web.xml** file
* Create a deployment descriptor
* Start the server and deploy the project
* Access the servlet

By default, a servlet application is located at the path <Tomcat-installation-directory>/webapps/ROOT and the class file would reside in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes.

If you have a fully qualified class name of com.myorg.MyServlet, then this servlet class must be located in WEB-INF/classes/com/myorg/MyServlet.class.

For now, let us copy HelloWorld.class into <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/classes and create following entries in web.xml file located in <Tomcat-installation-directory>/webapps/ROOT/WEB-INF/

<servlet>

<servlet-name>HelloWorld</servlet-name>

<servlet-class>HelloWorld</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HelloWorld</servlet-name>

<url-pattern>/HelloWorld</url-pattern>

</servlet-mapping>

Above entries to be created inside <web-app>...</web-app> tags available in web.xml file. There could be various entries in this table already available, but never mind.

You are almost done, now let us start tomcat server using <Tomcat-installation-directory>\bin\startup.bat (on windows) or <Tomcat-installation-directory>/bin/startup.sh (on Linux/Solaris etc.) and finally type http://localhost:8080/HelloWorld in browser's address box. If everything goes fine, you would get following result:



**HTTP requests Handling**

To handle HTTP requests in a servlet, extend the HttpServlet class and override the servlet methods that handle the HTTP requests which servlet supports. Servlets can be used for handling both the GET Requests and the POST Requests. The methods that handle these requests are

* doGet()
* doPost()
* doPut()
* doDelete()
* doOptions()
* doTrace()

**Handling HTTP GET Requests**

The Servlet is invoked when a form on a web page is submitted. The URL identifies a servlet to process the HTTP GET request.

The doGet() method is overridden to process any HTTP GET requests that sent to servlet. It may use getParameter() method of HttpServletRequest to obtain information that was made by the user. A response is then formulated.

**Handling HTTP POST Requests**

The doPost() method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the getParameter() method of HttpServletRequest to obtain the information that was made by the user. A response is then formulated

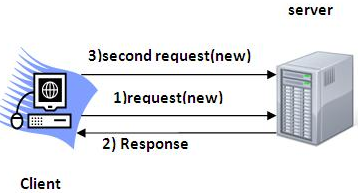
**Session Management**

Session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It requires to recognize the user.It is used to recognize the particular user.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:



**Session Tracking Techniques**

There are four techniques used in Session tracking:

* Cookies
* Hidden Form Field
* URL Rewriting
* HttpSession

**Cookies in Servlet**

A cookie is a small piece of information that is persisted between the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

**Example**

Cookie ck=new Cookie("user","binod");//creating cookie object

response.addCookie(ck);//adding cookie in the response

**Advantage of Cookies**

Simplest technique of maintaining the state. Cookies are maintained at client side.

**Disadvantage of Cookies**

It will not work if cookie is disabled from the browser.

Only textual information can be set in Cookie object.

**2) Hidden Form Field**

In case of Hidden Form Field a hidden (invisible) textfield is used for maintaining the state of an user.

In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser.

Let's see the code to store value in hidden field.

<input type="hidden" name="uname" value="Vimal Jaiswal">

Here, uname is the hidden field name and Vimal Jaiswal is the hidden field value.

**Advantage of Hidden Form Field**

It will always work whether cookie is disabled or not.

**Disadvantage of Hidden Form Field:**

It is maintained at server side.Extra form submission is required on each pages.

Only textual information can be used.

**3) URL rewriting**

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.

**Advantage of URL Rewriting**

It will always work whether cookie is disabled or not (browser independent).

Extra form submission is not required on each pages.

**Disadvantage of URL Rewriting**

It will work only with links.

It can send only textual information.

**4) HttpSession interface**

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

* bind objects
* view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.

**|Example**

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

out.print("Welcome "+n);

HttpSession session=request.getSession();

session.setAttribute("uname",n);

**JSP life cycle**

A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

The following are the paths followed by a JSP

* Compilation
* Initialization
* Execution
* Cleanup

The four major phases of JSP life cycle are very similar to Servlet Life Cycle and they are as follows:



**The Scriptlet:**

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the page scripting language.

Following is the syntax of Scriptlet:

<% code fragment %>

You can write XML equivalent of the above syntax as follows:

<jsp:scriptlet> code fragment </jsp:scriptlet>

Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple and first example for JSP:

<html>

<head><title>Hello World</title></head>

<body>

Hello World!<br/>

<% out.println("Your IP address is " + request.getRemoteAddr()); %>

</body>

</html>

**JavaServer Pages Standard Tag Library (JSTL)**

The JavaServer Pages Standard Tag Library (JSTL) is a collection of useful JSP tags which encapsulates core functionality common to many JSP applications.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags.

The JSTL tags can be classified, according to their functions:

* Core Tags
* Formatting tags
* SQL tags
* XML tags
* JSTL Functions

**Core Tags:**

The JSTL core tag provide variable support, URL management, flow control etc. Following is the syntax to include JSTL Core library in your JSP:

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

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <c:out > | Like <%= ... >, but for expressions. |
| <c:set > | Sets the result of an expression evaluation in a 'scope' |
| <c:forEach > | The basic iteration tag, accepting many different collection types and supporting subsetting and other functionality . |
| <c:redirect > | Redirects to a new URL. |

**Formatting tags:**

The JSTL formatting tags are used to format and display text, the date, the time, and numbers for internationalized Web sites. Following is the syntax to include Formatting library in your JSP:

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

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <fmt:formatNumber> | To render numerical value with specific precision or format. |
| <fmt:formatDate> | Formats a date and/or time using the supplied styles and pattern |
| <fmt:message> | To display an internationalized message. |

**SQL tags:**

The JSTL SQL tag library provides tags for interacting with relational databases (RDBMSs) such as Oracle, mySQL, or Microsoft SQL Server.

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

|  |  |
| --- | --- |
| **Teg** | **Description** |
| <sql:setDataSource> | Creates a simple DataSource suitable only for prototyping |
| <sql:query> | Executes the SQL query defined in its body or through the sql attribute. |
| <sql:update> | Executes the SQL update defined in its body or through the sql attribute. |
| <sql:param> | Sets a parameter in an SQL statement to the specified value. |

**XML tags:**

The JSTL XML tags provide a JSP-centric way of creating and manipulating XML documents. Following is the syntax to include JSTL XML library in your JSP.

The JSTL XML tag library has custom tags for interacting with XML data. This includes parsing XML, transforming XML data, and flow control based on XPath expressions.

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

|  |  |
| --- | --- |
| **Tag** | **Description** |
| <x:out> | Like <%= ... >, but for XPath expressions. |
| <x:parse> | Use to parse XML data specified either via an attribute or in the tag body. |
| <x:set > | Sets a variable to the value of an XPath expression. |
| <x:forEach> | To loop over nodes in an XML document. |

**JSTL Functions:**

JSTL includes a number of standard functions, most of which are common string manipulation functions. Following is the syntax to include JSTL Functions library in your JSP:

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| fn:contains() | Tests if an input string contains the specified substring. |
| fn:endsWith() | Tests if an input string ends with the specified suffix. |
| fn:indexOf() | Returns the index withing a string of the first occurrence of a specified substring. |
| fn:replace() | Returns a string resulting from replacing in an input string all occurrences with a given string. |