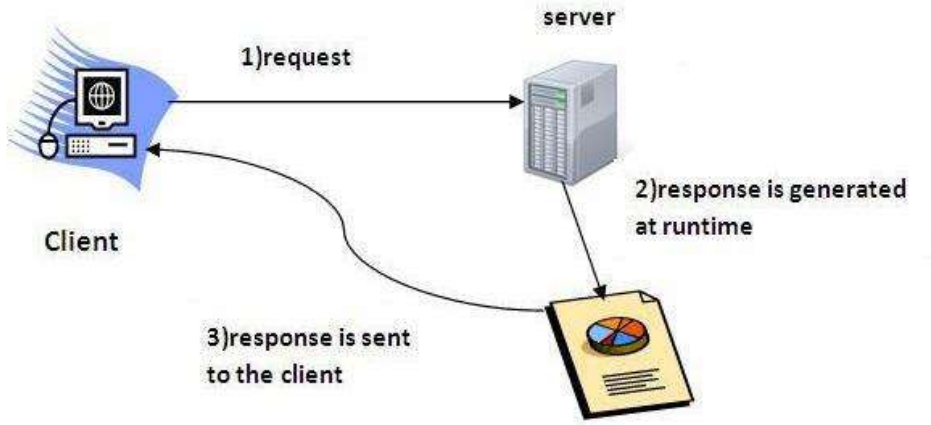
UNIT-III

SERVLETS

Servlet technology is used to create web application (resides at server side and generates dynamic web page). **Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there was many disadvantages of this technology. We have discussed these disadvantages below.
There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

What is a Servlet?
Servlet can be described in many ways, depending on the context.

- It is a technology i.e. used to create web application.
- It is an API that provides many interfaces and classes including documentations.
- It is an interface that must be implemented for creating any servlet.
- It is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
- It is a web component that is deployed on the server to create dynamic web page.



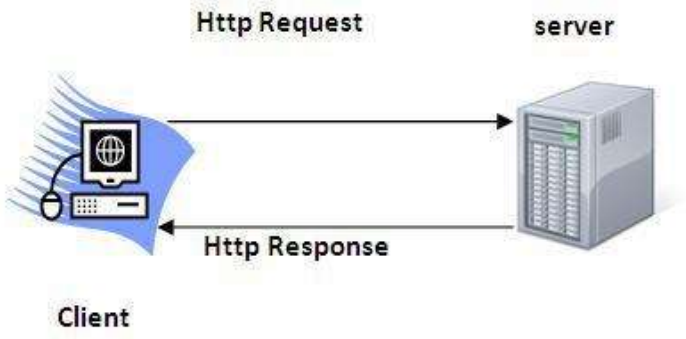
What is web application?
A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

Servlet Terminology:
There are some key points that must be known by the servlet programmer like server, container, get request, post request etc. The basic **terminology used in servlet** are given below:

1. HTTP
2. HTTP Request Types
3. Difference between Get and Post method
4. Container
5. Server and Difference between web server and application server
6. Content Type
7. Introduction of XML
8. Deployment

HTTP (Hyper Text Transfer Protocol):

1. Http is the protocol that allows web servers and browsers to exchange data over the web.
2. It is a request response protocol.
3. Http uses reliable TCP connections bydefault on TCP port 80.
4. It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user bydefault.



Http Request Methods

Every request has a header that tells the status of the client. There are many request methods. Get and Post requests are mostly used. The http request methods are:

- GET
- POST
- HEAD
- PUT
- DELETE
- OPTIONS
- TRACE

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

---

What is the difference between Get and Post?

There are many differences between the Get and Post request.

GET	POST
1) In case of Get request, only **limited amount of data** can be sent because data is sent in header.	In case of post request, **large amount of data** can be sent because data is sent in body.
2) Get request is **not secured** because data is exposed in URL bar.	Post request is **secured** because data is not exposed in URL bar.
3) Get request **can be bookmarked**	Post request **cannot be** bookmarked
4) Get request is **idempotent**. It means second request will be ignored until response of first request is delivered.	Post request is **non-idempotent**
5) Get request is **more efficient** and used more than Post	Post request is **less efficient** and used less than get.

CONTAINER:
It provides runtime environment for JavaEE (J2EE) applications. It performs many operations that are given below:

1. Life Cycle Management
2. Multithreaded support
3. Object Pooling
4. Security etc.

SERVER:
It is a running program or software that provides services. There are two types of servers:

1. Web Server
2. Application Server

Web Server: Web server contains only web or servlet container. It can be used for servlet, JSP, struts, JSF etc. It can't be used for EJB.
Example of Web Servers are: **Apache Tomcat** and **Resin**.

Application Server: Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc.
Example of Application Servers are:

1. **JBoss** Open-source server from JBoss community.
2. **Glassfish** provided by Sun Microsystem. Now acquired by Oracle.
3. **Weblogic** provided by Oracle. It more secured.
4. **Websphere** provided by IBM.

Content Type: Content Type is also known as MIME (Multipurpose internet Mail Extension) Type. It is a HTTP header that provides the description about what are you sending to the browser.
There are many content types:

- text/html
- text/plain
- application/msword
- application/vnd.ms-excel
- application/jar
- application/pdf
- application/octet-stream
- application/x-zip
- images/jpeg
- video/quicktime etc.

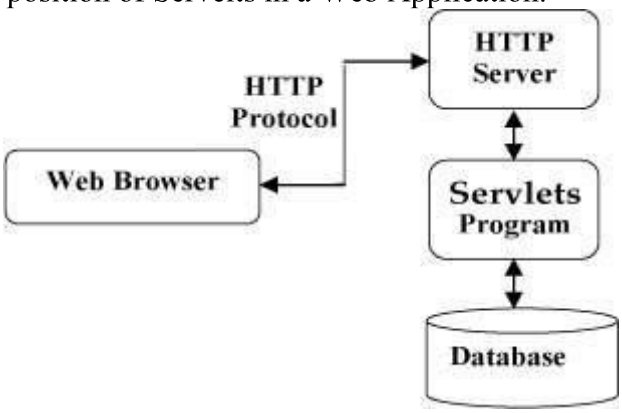
ADVANTAGES OF SERVLET

Servlets offer several advantages.

- 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.

SERVLET API

Servlet API provides Classes and Interface to develop web based applications.

Package: Servlet API contains two java packages are used to develop the servlet programs, they are:

- javax.servlet
- javax.servlet.http

javax.servlet: javax.servlet package contains list of interfaces and classes that are used by the servlet or web container. These classes and interface are not specific to any protocol.

Interface	
Filter	A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.
FilterChain	A FilterChain is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource.
FilterConfig	A filter configuration object used by a servlet container to pass information to a filter during initialization.
RequestDispatcher	Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server.

Servlet	Defines methods that all servlets must implement.
ServletConfig	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
ServletContextAttributeListener	Implementations of this interface receive notifications of changes to the attribute list on the servlet context of a web application.
ServletContextListener	Implementations of this interface receive notifications about changes to the servlet context of the web application they are part of.
ServletRequest	Defines an object to provide client request information to a servlet.
ServletRequestAttributeListener	A ServletRequestAttributeListener can be implemented by the developer interested in being notified of request attribute changes.
ServletRequestListener	A ServletRequestListener can be implemented by the developer interested in being notified of requests coming in and out of scope in a web component.
ServletResponse	Defines an object to assist a servlet in sending a response to the client.
SingleThreadModel	Deprecated. *As of Java Servlet API 2.4, with no direct replacement.*
CLASSES	
GenericServlet	Defines a generic, protocol-independent servlet.
ServletContextAttributeEvent	This is the event class for notifications about changes to the attributes of the servlet context of a web application.
ServletContextEvent	This is the event class for notifications about changes to the servlet context of a web application.
ServletInputStream	Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time.
ServletOutputStream	Provides an output stream for sending binary data to the client.
ServletRequestAttributeEvent	This is the event class for notifications of changes to the attributes of the servlet request in an application.
ServletRequestEvent	Events of this kind indicate lifecycle events for a ServletRequest.
ServletRequestWrapper	Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
ServletResponseWrapper	Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.
EXCEPTIONS	
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

javax.servlet.http: javax.servlet.http package contains list of classes and interfaces to define http servlet programs. This package is used to interact with browser using http protocol. It is only responsible for http requests.

INTERFACES	
HttpServletRequest	Extends the ServletRequest interface to provide request information for HTTP servlets.
HttpServletResponse	Extends the ServletResponse interface to provide HTTP-specific functionality in sending a response.
HttpSession	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
HttpSessionActivationListener	Objects that are bound to a session may listen to container events notifying them that sessions will be passivated and that session will be activated.
HttpSessionAttributeListener	This listener interface can be implemented in order to get notifications of changes to the attribute lists of sessions within this web application.
HttpSessionBindingListener	Causes an object to be notified when it is bound to or unbound from a session.
HttpSessionContext	**Deprecated.** *As of Java(tm) Servlet API 2.1 for security reasons, with no replacement.*
HttpSessionListener	Implementations of this interface are notified of changes to the list of active sessions in a web application.
CLASSES	
Cookie	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
HttpServlet	Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
HttpServletRequestWrapper	Provides a convenient implementation of the HttpServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
HttpServletResponseWrapper	Provides a convenient implementation of the HttpServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.
HttpSessionBindingEvent	Events of this type are either sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to aHttpSessionAttributeListener that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session.
HttpSessionEvent	This is the class representing event notifications for changes to sessions within a web application.
HttpUtils	**Deprecated.** *As of Java(tm) Servlet API 2.3.*

SET CLASSPATH IN SERVLET

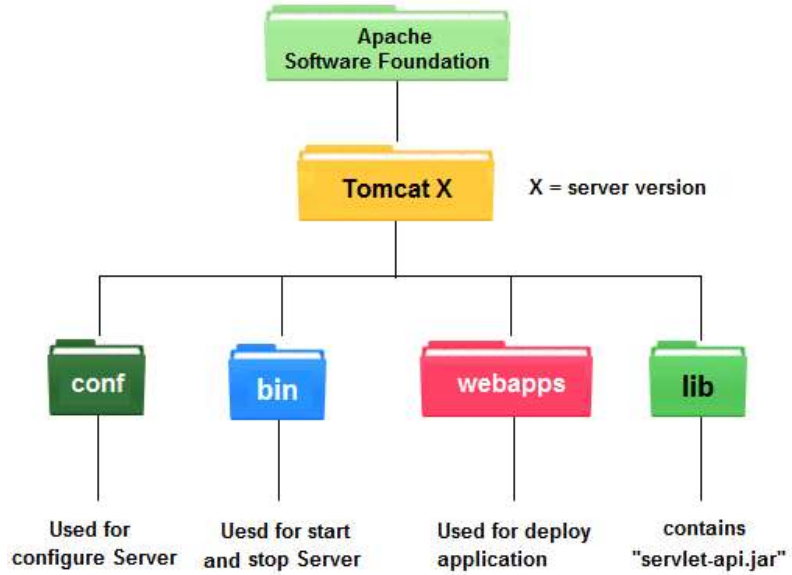
Installation of server

Download tomcat from the following url https://tomcat.apache.org/download-70.cgi Download tomcat select tomcat installer version(.exe), will get download as exe file, install into default location than "tomcat home directory" will be;

> "C:\program files\apache software foundation\tomcat 6.0"

Or we can direct download from our website Tomcat 7.0 Download Tomcat 7.0

Hierarchy of Tomcat Server



Below gives the description about all folders related to Tomcat Server and their uses.

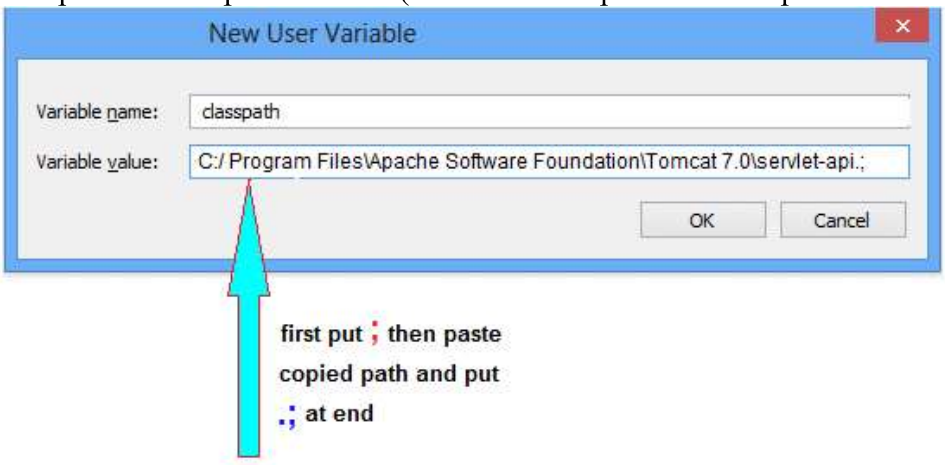
- **bin:** This sub folder used to start and stop the tomcat server.
- **conf:** This folder is used to configure server port number for this use server.xml file.
- **lib:** This folder contains "servlet-api.jar" used to set classpath for servlet programming, this is vender specific.jar file for servlet programming.
- **webapps:** This is the web application deployment folder used to deploy web application by copying webroot or for copy you application.

CLASSPATH

classpath variable is set for providing path of all java classes which is used in our application. All classes related to servlet are available in **lib/servlet-api.jar** so we set classpath upto **lib/servlet-api.jar**.

Set Classpath in Servlet

Copy **Servlet-api.jar** file location and set the classpath in environment variable. If classpath is already set for core java programming, then need to edit classpath variable. To edit classpath just put **;** at end of previous variable and paste new copied location (without delete previous classpath variable).



C:\Program Files\Apache Software Foundation\Tomcat 7.0\lib\servlet-api.;

In the above image we can see "***C:\Program Files\Java\jdk1.6.0\jre\lib\rt.jar***" path is already set. Now we can only put our **servlet-api.jar** location at the end of this path. Finally our complete path is:

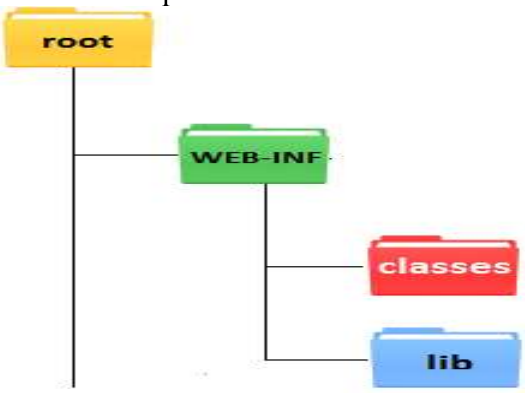
```
C:\Program Files\Java\jdk1.6.0\jre\lib\rt.jar; C:\Program Files\Apache Software
Foundation\Tomcat 7.0\lib\servlet-api.jar.;
```

DIRECTORY STRUCTURE

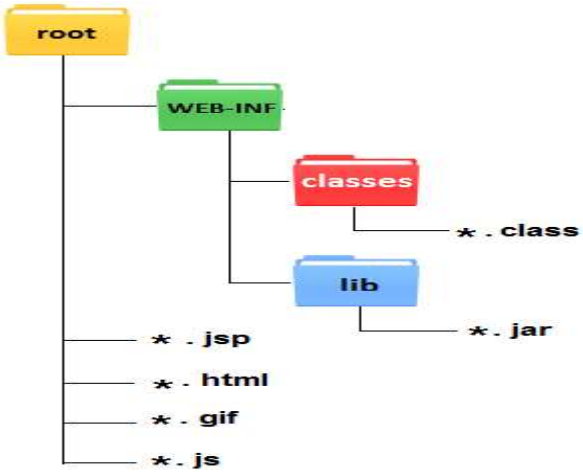
For creating web application we should follow standard directory structure provided by sun MicroSystem. Sun MicroSystem has given directory structure to make a web application server independent.

According to directory structure

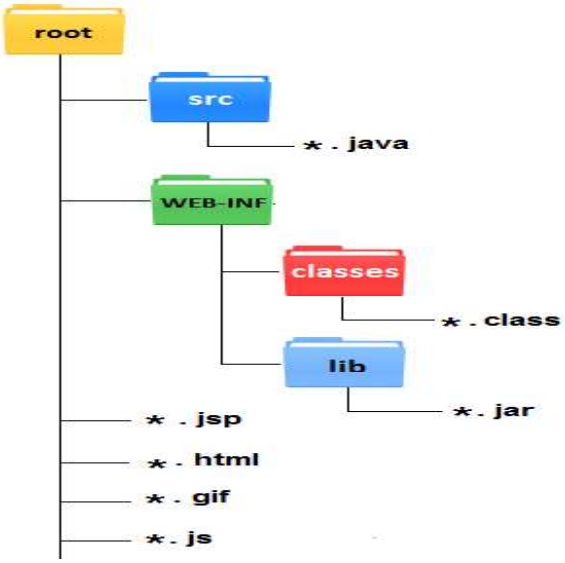
- An application contain root folder with any name.
- Under root folder a sub folder is required with a name WEB-INF.
- Under WEB-INF two sub-folder are required classes and lib.



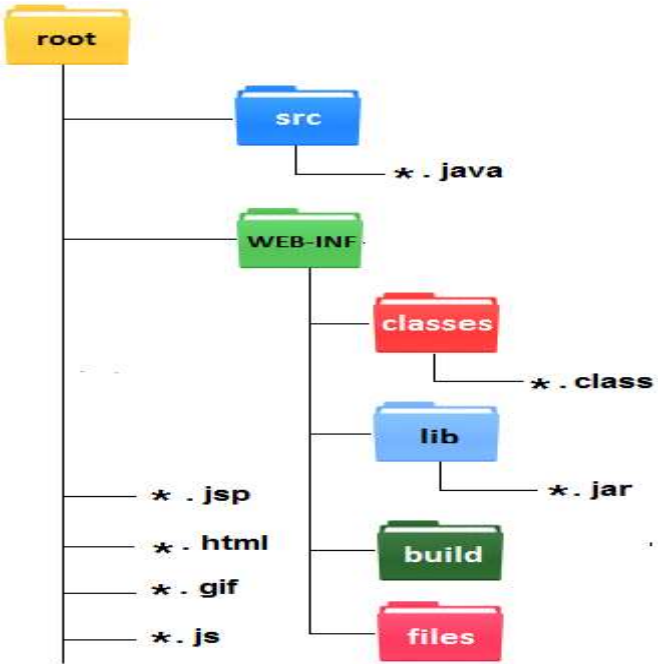
- All jar files placed inside lib folder.



- Under root folder src folder are place for .java files



- Under root folder or under WEB-INF any other folders can exits.
- All image, html, .js, jsp, etc files are placed inside root folder
- All .class files placed inside classes folder.



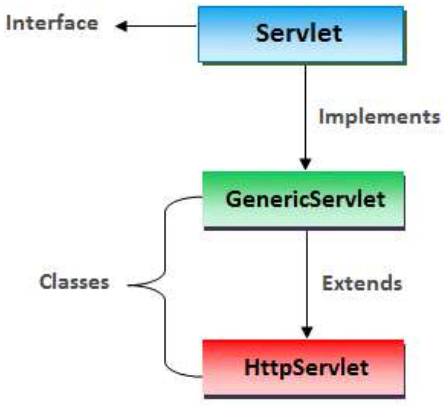
WEB.XML: It is a web application deployment descriptor file, contains detail description about web application like configuration of Servlet, Session management, Startup parameters, Welcome file..etc. We cannot change the directory or extension name of this **web.xml** because it is standard name to recognized by container at run-time. **web.xml** is present inside the Web-INF folder.

WEB-INF: This is a web application initialization folder to recognized by the web container at run time then folder name should be WEB-INF to deployed the web application successfully otherwise deployment name unsuccessful. Inside WEB-INF folder we put web.xml file, classes folder, lib folder and any user defined folder.

CREATING SERVLET

According to servlet API we have three ways to creating a servlet class.

- By implementing servlet interface
- By extending GenericServlet class
- By extending HttpServlet class



By implementing servlet interface

Example

```
public class myServlet implements Servlet {....... ....... }
```

By extending GenericServlet class

Example

```
public class myServlet extends GenericServlet{ ....... ....... }
```

By extending HttpServlet class

Example

```
public class myServlet extends HttpServlet{ ..... ..... }
```

Servlet Interface

It is an interface to define a Servlet, the implementation class of this Servlet should override all methods of Servlet interface. Servlet interface needs to be implemented for creating any Servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the Servlet, to service the requests, and to destroy the Servlet and 2 non-life cycle methods.

Methods of Servlet interface

Method	Description
public void init(ServletConfig config)	initializes the Servlet. It is the life cycle method of Servlet and invoked by the web container only once.
public void service(ServletRequest request,ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that Servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about Servlet such as writer, copyright, version etc.

Example of servlet by implementing Servlet interface

Syntax

```
public class myServlet implements server { .... }
public void destroy() { ..... }
public void init(ServletConfig se) { ..... }
public ServletConfig getServletConfig(){ ..... }
public String getServiceInfo(){ .... }
public void service(ServletRequest req, ServletResponse resp)throws
IOException, ServletException
{ .... }
```

GenericServlet

GenericServlet class Implements Servlet, ServletConfig and Serializable Interfaces. It provides the implementation of all the methods of these Interfaces except the service method. GenericServlet class can handle any type of request so it is protocol Independent. We may create a generic Servlet by inheriting the GenericServlet class and providing the Implementation of the service method.

This is a implemented abstract class for Servlet Interface, have the implementation for all methods of Servlet interface except service method.

Methods of GenericServlet

All methods of Servlet Interface are inherit in GenericServlet class because it implements Servlet Interface. Following methods are used in GenericServlet

Example of Servlet by inheriting the GenericServlet class

```
import java.io.*;
import javax.servlet.*;
public class GenericServletDemo extends GenericServlet
{
public void service(ServletRequest req,ServletResponse resp)
throws IOException,ServletException
{
res.setContentType("text/html");
PrintWriter out=resp.getWriter();
out.print("<html><body>");
out.print("<b>Example of GenericServlet</b>");
out.print("</body></html>");
}
}
```

HttpServlet

This is used to define HttpServlet, to receive http protocal request and to send http protocol response to the client. The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace, doDelete, doOption etc.

1. **doGet** : This method is called by the server (via the service method) to allow a servlet to handle a GET request.
   Syntax:
   ```
   protected void doGet(HttpServletRequest req,  HttpServletResponse resp)
               throws ServletException,  java.io.IOException
   ```
   where
   `req` - an `HttpServletRequest` object that contains the request the client has made of the servlet
   `resp` - an `HttpServletResponse` object that contains the response the servlet sends to the client
2. **doPost**: This method is called by the server (via the service method) to allow a servlet to handle a POST request. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers.
   Syntax:
   ```
   protected void doPost(HttpServletRequest req, HttpServletResponse resp)
          throws ServletException, java.io.IOException
   ```
3. **doHead**: This method receives an HTTP HEAD request from the protected service method and handles the request. The client sends a HEAD request when it wants to see only the headers of a response, such as Content-Type or Content-Length. The HTTP HEAD method counts the output bytes in the response to set the Content-Length header accurately.
   Syntax:
   ```
   protected void doHead(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, java.io.IOException
   ```
4. **doTrace**: This method is called by the server (via the service method) to allow a servlet to handle a TRACE request. A TRACE returns the headers sent with the TRACE request to the client, so that they can be used in debugging. There's no need to override this method.

   Syntax:
   ```
   protected void doTrace(HttpServletRequest req,HttpServletResponse resp)
          throws ServletException, java.io.IOException
   ```

5. **doDelete**: This method is called by the server (via the service method) to allow a servlet to handle a DELETE request. The DELETE operation allows a client to remove a document or Web page from the server.

   Syntax:

```
protected void doDelete(HttpServletRequest req,HttpServletResponse resp)
            throws ServletException, java.io.IOException
```

6. **doOption**: Called by the server (via the service method) to allow a servlet to handle a OPTIONS request. The OPTIONS request determines which HTTP methods the server supports and returns an appropriate header. For example, if a servlet overrides doGet, this method returns the following header:
   Allow: GET, HEAD, TRACE, OPTIONS

   Syntax:

```
protected void doDelete(HttpServletRequest req,HttpServletResponse resp)
            throws ServletException, java.io.IOException
```

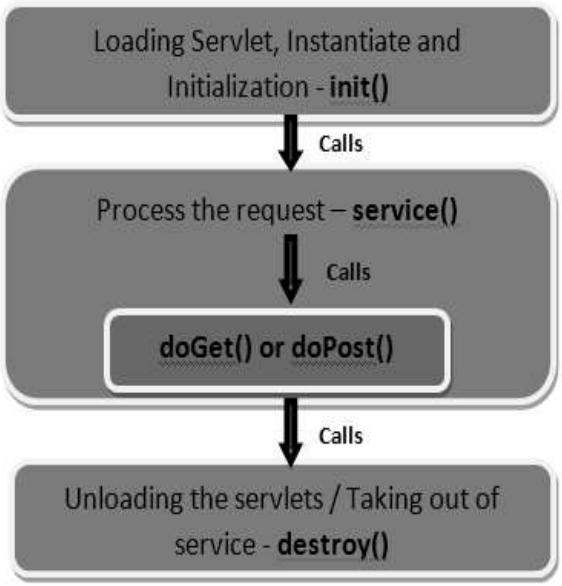
7. **doPut**: This method is called by the server (via the service method) to allow a servlet to handle a PUT request. The PUT operation allows a client to place a file on the server and is similar to sending a file by FTP.
   Syntax:

```
protected void doDelete(HttpServletRequest req,HttpServletResponse resp)
            throws ServletException, java.io.IOException
```

SERVLET LIFE CYCLE

The javax.servlet.Servlet interface defines the **life cycle methods of servlet** such as init( ), service( ) and destroy( ). The Web container invokes the init( ), service( ) and destroy( ) methods of a servlet during its life cycle. The sequence in which the Web container calls the life cycle methods of a servlet is:

1. The Web container loads the servlet class and creates one or more instances of the servlet class.
2. The Web container invokes init() method of the servlet instance during initialization of the servlet. The init() method is invoked only once in the servlet life cycle.
3. The Web container invokes the service() method to allow a servlet to process a client request.
4. The service() method processes the request and returns the response back to the Web container.
5. The servlet then waits to receive and process subsequent requests as explained in steps 3 and 4.
6. The Web container calls the destroy() method before removing the servlet instance from the service. The destroy() method is also invoked only once in a servlet life cycle.



1. **The init( ) Method:** The init() method is called during initialization phase of the servlet life cycle. The Web container first maps the requested URL to the corresponding servlet available in the Web container and then instantiates the servlet. The Web container then creates an object of the ServletConfig interface, which contains the startup configuration information, such as initialization parameters of the servlet. The Web container then calls the init() method of the servlet and passes the ServletConfig object to it.

   The init() method throws a ServletException if the Web container cannot initialize the servlet resources. The servlet initialization completes before any client requests are accepted. The following code snippet shows the init() method:

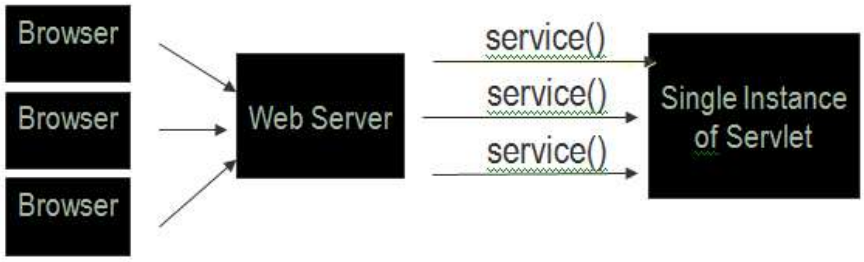
```
public class ServletLifeCycle extends HttpServlet
{
   static int count;
   public void init(ServletConfig config) throws ServletException
   {
      count=0;
   }
}
```

2. **The service( ) Method:** The service( ) method processes the client requests. Each time the Web container receives a client request, it invokes the service( ) method. The service( ) method is invoked only after the initialization of the servlet is complete. When the Web container calls the service( ) method, it passes an object of the ServletRequest interface and an object of the ServletResponse interface. The ServletRequest object contains information about the service request made by the client. The ServletResponse object contains the information returned by the servlet to the client. The following code snippet shows the service( ) method:

```
public void service(ServletRequest req, ServletResponse res)
throws ServletException, IOException
```

   The service() method throws ServletException exception when an exception occurs that interferes with the normal operation of the servlet. The service() method throws IOException when an input or output exception occurs.

   The service() method dispatches a client request to one of the request handler methods of the HttpServlet interface, such as the doGet(), doPost(), doHead() or doPut(). The request handler methods accept the objects of the HttpServletRequest and HttpServletResponse as parameters from the service() method.



3. **The doGet() Method:** The doGet() method processes client request, which is sent by the client, using the HTTP GET method. GET is a type of HTTP request method that is commonly used to retrieve static resources. To handle client requests that are received using GET method, we need to override the doGet() method in the servlet class. In the doGet() method, we can retrieve the client information of the HttpServletRequest object.

   We can use the HttpServletResponse object to send the response back to the client. The following code snippet shows the doGet() method:

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
```

4. **The doPost() Method:** The doPost() method handles requests in a servlet, which is sent by the client, using the HTTP POST method. For example, if a client is entering registration data in an HTML form, the data can be sent using the POST method. Unlike the GET method, the POST request sends the data as part of the HTTP request body. As a result, the data sent does not appear as a part of URL.

   To handle requests in a servlet that is sent using the POST method, we need to override the doPost() method. In the doPost() method, we can process the request and send the response back to the client. The following code snippet shows the doPost() method:

```
public void doPost(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
```

5. **The doHead() Method:**The doHead() method handles requests sent using the HTTP HEAD method. Similar to the GET method, the HEAD method also sends requests to the server. The only difference between the GET and the HEAD methods is that the HEAD method returns the header of the response, which contains entries, such as Content-Type, Content-Length, and Last-Modified.

   The HEAD method is used to find whether a requested resource exists. It is also used to obtain information about the resource, such as type of the resource. This information is required before requesting for resource content. The following code snippet shows the doHead() method:

```
public void doHead(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
```

6. **The doPut() Method:** The doPut() method handles requests sent using the HTTP PUT method. The PUT method allows a client to store information on the server. The following code snippet shows the doPut() method:

```
public void doPut(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
```

7. **The destroy() Method:** The destroy() method marks the end of the life cycle of a servlet. The Web container calls the destroy() method before removing a servlet instance from the service. The Web container calls the destroy() method when
   - The time period specified for the servlet has elapsed. The time period of a servlet is the period for which the servlet is kept in the active state by the Web container to service the client request.
   - The Web container needs to release servlet instances to conserve memory.
   - The Web container is about to shut down.

   In the destroy() method we can write the code to release the resources occupied by the servlet. The destroy() method is also used to save any persistent information before the servlet instance is removed from the service. The following code snippet shows the destroy() method:

```
public void destroy( );
```

Servler Life Cycle Example

```
import javax.servlet.*;
import java.io.*;

public class myservlet extends GenericServlet
{
 public void init(ServletConfig sc)
{
  System.out.println("init executed...");
}
```

```
public void service(ServletRequest req, ServletResponse resp)throws
IOException, ServletException
{
System.out.println("service executed...");
PrintWriter out=resp.getWriter();
resp.setContentType("text/html");
out.println("plz observe output on server console window");
}
public void destroy()
{
 System.out.println("Destroy executed...");
}
}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>srv</servlet-name>
<servlet-class>myservlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>srv</servlet-name>
<url-pattern>/ms</url-pattern>
</servlet-mapping>
</web-app>
```

CONTENT TYPE

Content Type is also known as **MIME Type**. MIME stand for **Multipurpose internet Mail Extension**. It is a HTTP header that provides the description about what are you sending to the browser (like send image, text, video etc.). This is the format of http protocol to carry the response contains to the client..

Example: Suppose you send html text based file as a response to the client the MIME type specification is

Syntax

```
response.setcontentType("text/html");
```

MIME type have two parts, They are:

- **Base name:** It is the generic name of file.
- **Extension name:** It is extension name for specific file type.

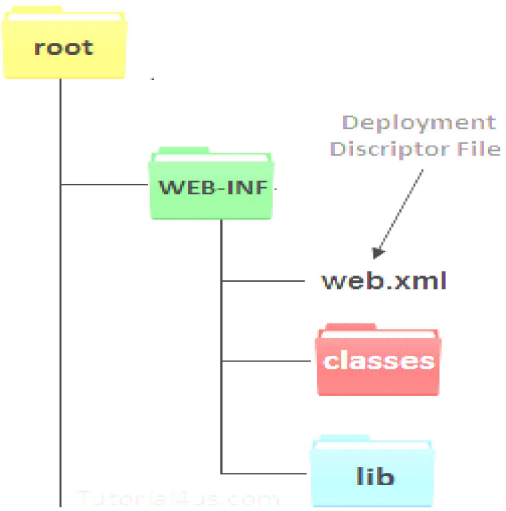
The supporting MIME type by http protocol are:

File	MIME Type	Extension
Xml	text/xml.	.xml.
HTML	text/html.	.html.

Plaintext File	text/plain.	.txt.
PDF	application/pdf.	.pdf.
gif Image	image/gif.	.gif.
JPEG Image	image/jpeg.	.jpeg.
PNG Image	image/x-png.	.png.
MP3 Music File	audio/mpeg.	.mp3.
MS Word Document	application/msword.	.doc.
Excel work sheet	application/vnd.ms-sheet.	.xls.
Power Point Document	application/vnd.ms-powerpoint.	.ppt.

WEB.XML

It is an xml file which acts as a mediator between a web application developer and a web container. It is also called Deployment Descriptor Files.



Note: we cannot change the directory or extension name of this web.xml because it is standard name to recognized by container at run-time.

In web.xml file, we configure the following

- Servlet
- JSP
- Filter
- Listeness
- Welcome files
- Security etc....

To configure a servlet file we need the following two xml tag.

- **<servlet>:** tag are used for configure the servlet, Within this tag we write Servlet name and class name.
- **<Serlet-mapping>:** tag are used for map the servlet to a URL .

The structure of web.xml files is already defined by sun MicroSystem. So as a developer we should follows the structure to develop the configuration of web resources.

While configuring a servlet in web.xml, three names are added for it.

- Alias name or register name
- Fully qualified class name

Syntax

```
<web-app>
<servlet>
<servlet-name>alias name</servlet-name>
<servlet-class>fully qualified class name</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>alias name</servlet-name>
<url-pattern>/url pattern</url-pattern>
</servlet-mapping>
</web-app>
```

<load-on-startup> Element

This is a **web.xml** configuration elements used to configure a servlet to create servlet object during startup time of the server or application deploy on server. It is also known as **pre initialization** of servlet. This element need integer value to configure.

Advantage of load-on-startup element : As we know well, servlet is loaded at first request. That means it consumes more time at first request. If we specify the load-on-startup in web.xml, servlet will be loaded at project deployment time or server start. So, it will take less time for responding to first request.

Passing positive value: If we pass the positive value, the lower integer value servlet will be loaded before the higher integer value servlet. In other words, container loads the servlets in ascending integer value. The 0 value will be loaded first then 1, 2, 3 and so on.

web.xml

```
<web-app>
  ....

  <servlet>
   <servlet-name>servlet1</servlet-name>
   <servlet-class>com.tutorial4us.FirstServlet</servlet-class>
   <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet>
   <servlet-name>servlet2</servlet-name>
   <servlet-class>com.tutorial4us.SecondServlet</servlet-class>
   <load-on-startup>1</load-on-startup>
  </servlet>
...
</web-app>
```

There are defined 2 servlets, both servlets will be loaded at the time of project deployment or server start. But, servlet1 will be loaded first then servlet2.

Passing negative value: If a negative value is configure then a container ignores this tag and waits for first request to create an object of a servlet but a container does not throws any exception.

Welcome File Configuration

A **welcome file** is the file that is invoked automatically by the server, if you don't specify any file name. Welcome file is a default starting page of the website.

The welcome-file-list element of web-app, is used to define a list of welcome files. Its sub element is welcome-file that is used to define the welcome file. By default server looks for the welcome file in following order:

1. index.html
2. index.htm
3. index.jsp

Note: If welcome file is not configure and index.html or index.jsp does not exist in an application then container sends http status 404 message to the browser.

If you have specified welcome-file in web.xml, and all the files index.html, index.htm and index.jsp exists, priority goes to welcome-file.

If welcome-file-list entry doesn't exist in web.xml file, priority goes to index.html file then index.htm and at last index.jsp file.

Example

```
<web-app>
 ....
    <welcome-file-list>
   <welcome-file>home.html</welcome-file>
   <welcome-file>home.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Now, home.html and default.html will be the welcome files.

If you have the welcome file, you can directly invoke the project as given below:

Example

```
http://localhost:8888/myproject/
```

SERVLET FIRST PROGRAM

Servlet programming is very simple but we need some basic knowledge for example interface, abstract class, exception handling.

Steps to write servlet program

- Define web directory structure
- Define .jsp/.html pages
- Define servlet program, compile into classes folder
- Define web.xml, into WEB-INF folder
- Deploy the webroot into web server deployment folder location.
- Start the server
- Open web browser and invoke servlet/.html/.jsp.

Write a web application to send hello word as response to client using servlet.

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;

public class FirstServlet extends HttpServlet {
```

```
  public void service(HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
  {
   // get request parameter
   // business operation
   String resultvalue="<body bgcolor="cyan" text="red"> <h1> hello
word</h1></body>";

   // prepare response
   resp.setContentType("text/html");
   printWriter out=resp.getWriter();

   // send response
   out.print(resultvalue);
   out.close();
   }
}
```

web.xml

```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

</web-app>
```

ServletConfig

An object of ServletConfig is created by the web container for each servlet using its initialization phase. This object can be used to get configuration information from web.xml file. An object of ServletConfig is available to the Servlet during its execution, once the servlet execution is completed, automatically ServletConfig interface object will be removed by the container.

An object of ServletConfig interface contains <init-param> details at web.xml, of a particular servlet. The moment when we are using an object of ServletConfig, we need to configure the web.xml by writing < init-param > tag under <servlet> tag of web.xml.

Whenever compiler executes init() method then the ServletConfig will be created in general. An object of ServletConfig contain the <init-param> data in the form of key-value pairs, here the keys represents init param names and values are its values, which are represented in the web.xml file

Advantage of ServletConfig: If the configuration information is modified from the web.xml file, we don't need to change the Servlet. So it is easier to manage the web application if any specific content is modified from time to time.
The core advantage of ServletConfig is that you don't need to edit the Servlet file if information is modified from the web.xml file.

Methods of ServletConfig interface

- **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
- **public Enumeration getInitParameterNames():**Returns an enumeration of all the initialization parameter names.
- **public String getServletName():**Returns the name of the Servlet.
- **public ServletContext getServletContext():**Returns an object of ServletContext.

How to Get ServletConfig Object into Servlet

An object of ServletConfig can be obtained in 2 ways,

Syntax

```
ServletConfig conf = getServletConfig();
```

In the above statement, we are directly calling getServletConfig() method as it is available in Servlet interface, inherited into GenericServlet and defined and further inherited into HttpServlet and later inherited into our own servlet class.

Syntax

```
ServletConfig object will be available in init() method of the
servlet.
public void init(ServletConfig config)
{
// ......
}
```

How to Retrieve Data from ServletConfig Interface Object: In order to retrieve the data of the ServletConfig we have two methods, which are present in ServletConfig interface.

Syntax

```
1.   public String getInitParameter("param name");
2.   public Enumeration getInitParameterNames();
```

ServletContext

ServletContext is one of pre-defined interface available in javax.servlet.*; Object of ServletContext interface is available one per web application. An object of ServletContext is automatically created by the container when the web application is deployed.

Assume there exist a web application with 2 servlet classes, and they need to get some technical values from web.xml, in this case ServletContext concept will works great, i mean all servlets in the current web application can access these context values from the web.xml but it's not the case in ServletConfig, there only particular servlet can access the values from the web.xml which were written under <servlet> tag, hope you remember. Have doubt ? just check Example of ServletConfig.

How to Get ServletContext Object into Our Servlet Class

In servlet programming we have 3 approaches for obtaining an object of ServletContext interface

Way 1.

Syntax

```
ServletConfig conf = getServletConfig();
ServletContext context = conf.getServletContext();
```

First obtain an object of ServletConfig interface ServletConfig interface contain direct method to get Context object, getServletContext();.

Way 2.

Direct approach, just call getServletContext() method available in GenericServlet [pre-defined]. In general we are extending our class with HttpServlet, but we know HttpServlet is the sub class of GenericServlet.

Syntax

```
public class Java4s extends HttpServlet
{
public void doGet/doPost(-,-)
{
……..
}
ServletContext ctx = getServletContext();
}
```

Way 3.

We can get the object of ServletContext by making use of HttpServletRequest object, we have direct method in HttpServletRequest interface.

Syntax

```
public class Java4s extends HttpServlet
{
public void doGet/doPost(HttpServletRequest req,-)
{
ServletContext ctx = req.getServletContext();
}
}
```

How to Retrieve Data from ServletConfig Interface Object

ServletContext provide these 2 methods, In order to retrieve the data from the web..xml [In web.xml we have write <context-param> tag to provide the values, and this <context-param> should write outside of <servlet> tag as context should be accessed by all servlet classes].

In general database related properties will be written in this type of situation, where every servlet should access the same data.

Syntax

```
public String getInitParameter("param name");
public Enumeration getInitParameterNames();
```

doPost and doGet

Http protocol mostly use either get or post methods to transfer the request. post method are generally used whenever you want to transfer secure data like password, bank account etc.

	Get method	Post method
1	Get Request sends the request parameter as query string appended at the end of the request.	Post request send the request parameters as part of the http request body.
2	Get method is visible to everyone (It will be displayed in the address bar of browser).	Post method variables are not displayed in the URL.
3	Restriction on form data, only ASCII characters allowed.	No Restriction on form data, Binary data is also allowed.
4	Get methods have maximum size is 2000 character.	Post methods have maximum size is 8 mb.
5	Restriction on form length, So URL length is restricted	No restriction on form data.
6	Remain in browser history.	Never remain the browser history.

EXCEPTION HANDLING

The process of converting system error messages into user friendly error message is known as **Exception handling**. This is one of the powerful feature of Java to handle run time error and maintain normal flow of java application.

Exception: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's Instructions.

- Programmatically Exception Handling mechanism
- Declarative Exception Handling mechanism

Programmatically Exception Handling mechanism: The approach to use try, catch block in java code to handle exceptions is known as Programmatically Exception Handling mechanism.

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/> <br/>
<input type="submit" value="continue"/>
</form>
```

Example

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)
      {
    try{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

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

```
    Cookie ck=new Cookie("uname",n);//creating cookie object
    response.addCookie(ck);//adding cookie in the response

    //creating submit button
    out.print("<form action='servlet2'>");
    out.print("<input type='submit' value='continue'>");
    out.print("</form>");

    out.close();

        }catch(Exception e){System.out.println(e);}
  }
}
```

Declarative Exception Handling mechanism: The approach to use xml tags in web.xml file to handle the exception is known as declarative exception handling mechanism. This mechanism is useful if exception are common for more than one servlet program. In real time application this mechanism is widely use.

error.html

```
<html>
<body>
<p> Oooops..... page not found</p>
</body>
</html>
```

Myservlet.java

```
import java.io.*;
import javax.servlet.*;

public class FirstServlet extends HttpServlet {

  public void service(HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException
  {
   // get request parameter
   // business operation
   String resultvalue="<body bgcolor="cyan" text="red"> <h1> hello
word</h1></body>";

   // prepare response
   resp.setContentType("text/html");
   printWriter out=resp.getWriter();

   // send response
   out.print(resultvalue);
   out.close();
   }  }
```

web.xml

```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
```

```
</servlet>

<error-page>
<exception-type>java.lang.NumberFormateException</exception-type>
<location>/error.html</location>

</error-page>

</web-app>
```

War File

A war (web archive) File is a compressed format of files of a web project. It may have servlet, xml, jsp, image, html, css, js etc.

Advantage of war file:

Saves time: The war file combines all the files into a single unit. It is a compressed format of all files, So it takes less time while transferring file from client to server.

How to create war file?

To create war file, we need jar tool of JDK. We need to use -c switch of jar, to create the war file. Go inside the project directory of your project (outside the WEB-INF), then write the following command:

Syntax

```
jar -cvf projectname.war *
```

Here, -c is used to create file, -v to generate the verbose output and -f to specify the archive file name. The * (asterisk) symbol signifies that all the files of this directory (including sub directory).

How to extract war file manually?

To extract the war file, you need to use -x switch of jar tool of JDK. Let's see the command to extract the war file.

Syntax

```
jar -xvf projectname.war
```

Note: Server will extract the war file internally.

SERVLET COLABORATION

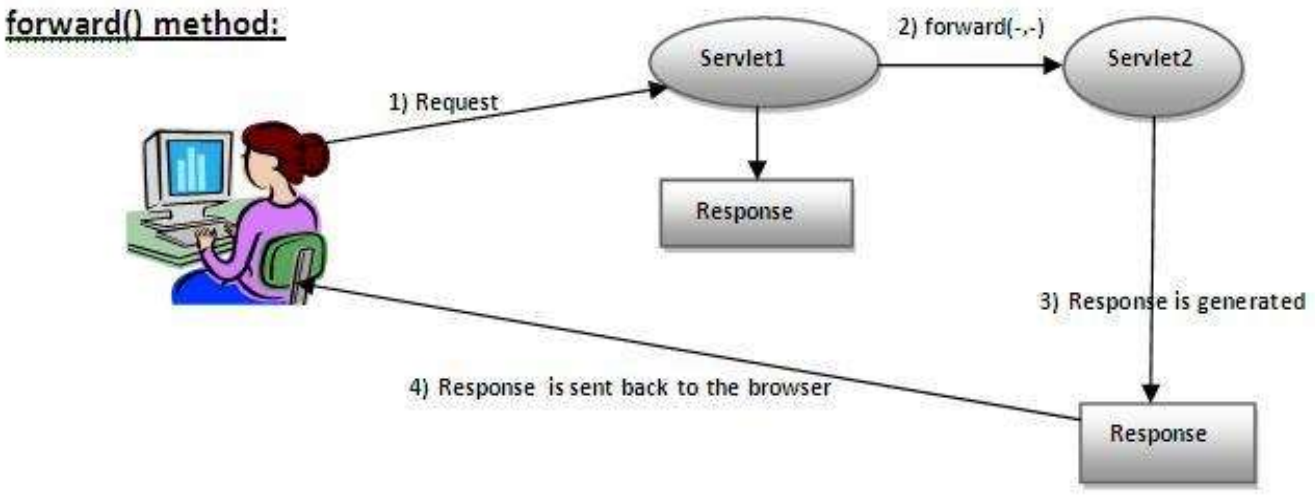
REQUESTDISPATCHER

The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp. This interface can also be used to include the content of another resource also. It is one of the way of servlet collaboration. There are two methods defined in the RequestDispatcher interface.

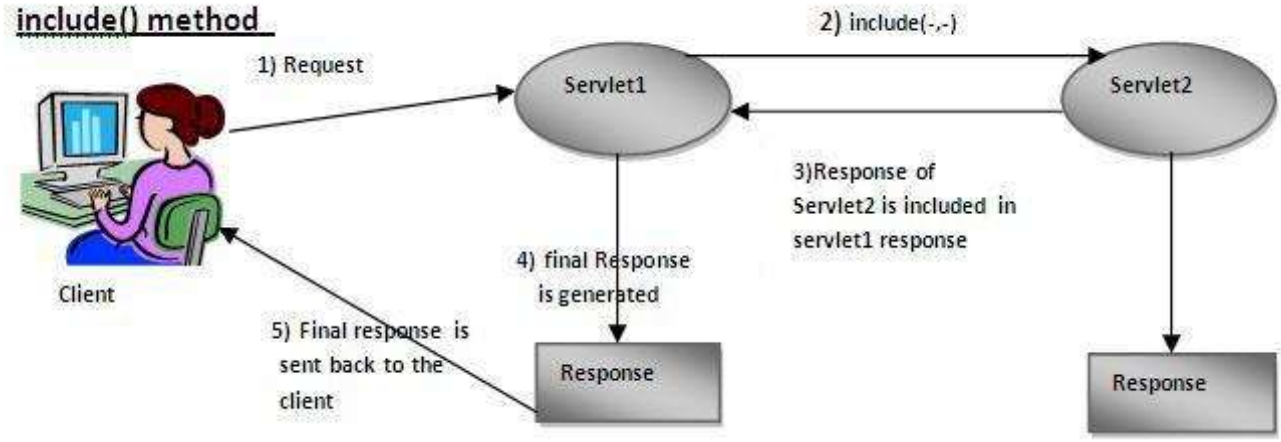
Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

1. **public void forward(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
2. **public void include(ServletRequest request,ServletResponse response)throws ServletException,java.io.IOException:**Includes the content of a resource (servlet, JSP page, or



In the above figure, response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.



In the above figure, response of second servlet is included in the response of the first servlet that is being sent to the client.

How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher.

Syntax of getRequestDispatcher method

```
public RequestDispatcher getRequestDispatcher(String resource);
```

Example of using getRequestDispatcher method

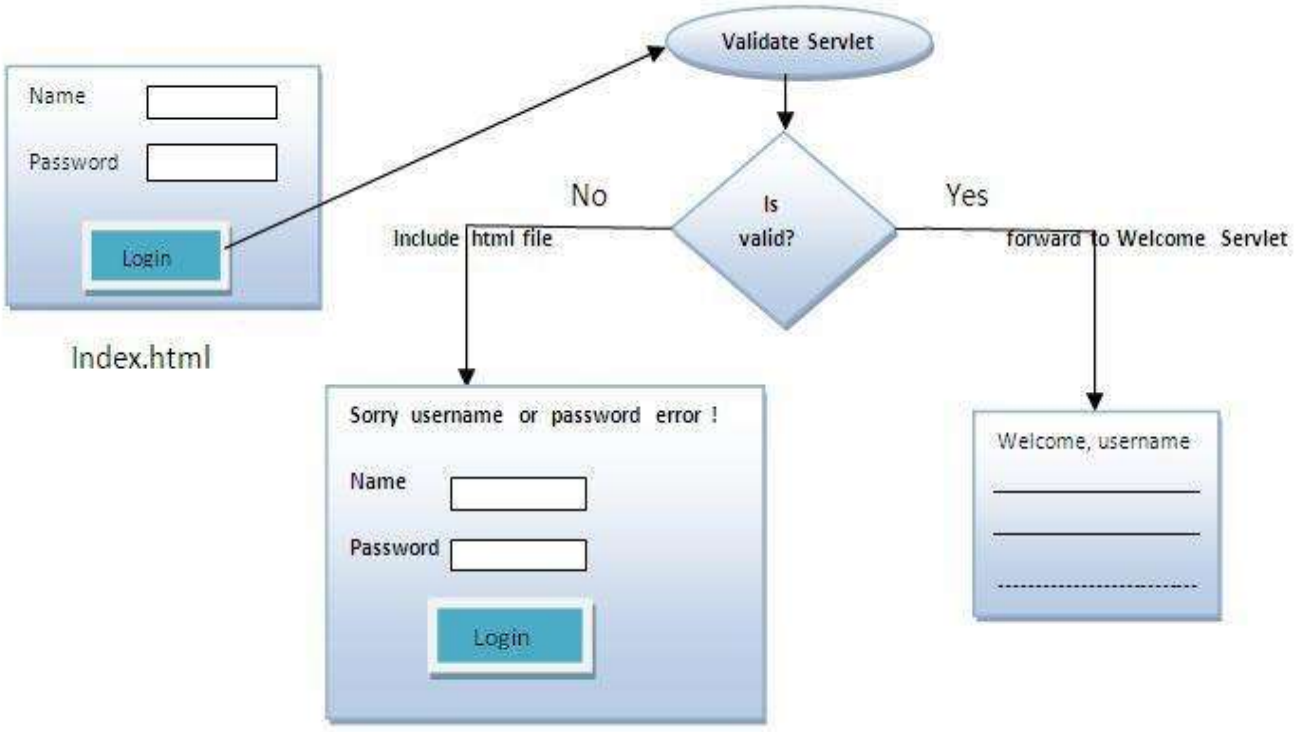
```
RequestDispatcher rd=request.getRequestDispatcher("servlet2");
//servlet2 is the url-pattern of the second servlet

rd.forward(request, response);//method may be include or forward
```

Example of RequestDispatcher interface

In this example, we are validating the password entered by the user. If password is servlet, it will forward the request to the WelcomeServlet, otherwise will show an error message: sorry username or password error!. In this program, we are checking for hardcoded information. But you can check it to the database also that we will see in the development chapter. In this example, we have created following files:

- **index.html file:** for getting input from the user.
- **Login.java file:** a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
- **WelcomeServlet.java file:** a servlet class for displaying the welcome message.

- **web.xml file:** a deployment descriptor file that contains the information about the servlet.



index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
Password:<input type="password" name="userPass"/><br/>
<input type="submit" value="login"/>
</form>
```

Login.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Login extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

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

    if(p.equals("servlet"){
        RequestDispatcher rd=request.getRequestDispatcher("servlet2");
        rd.forward(request, response);
    }
    else{
        out.print("Sorry UserName or Password Error!");
        RequestDispatcher rd=request.getRequestDispatcher("/index.html");
        rd.include(request, response);

        }
    }  }
```

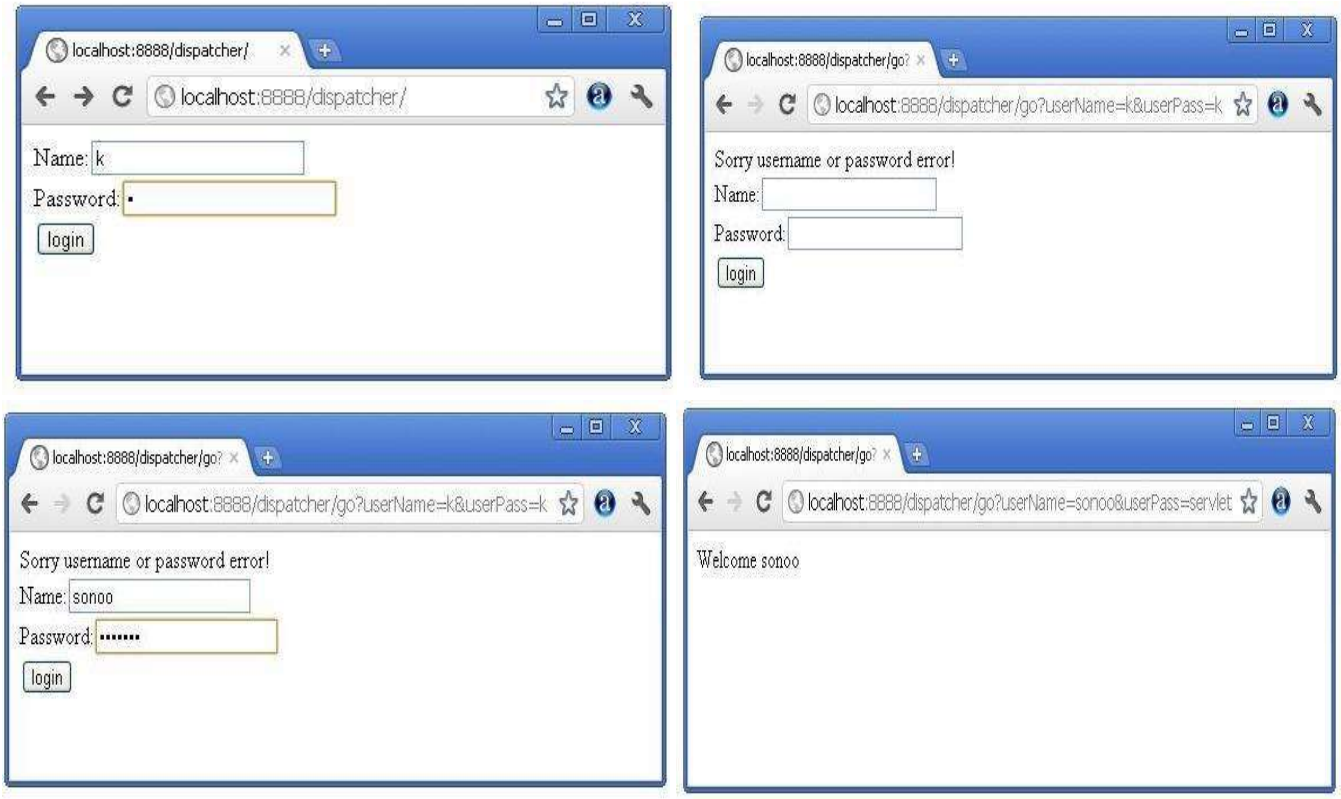
WelcomeServlet.java

```
import java.io.*;
```

```
import javax.servlet.http.*;

public class WelcomeServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)

        throws ServletException, IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
    out.print("Welcome "+n);
    }

}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>Login</servlet-name>
<servlet-class>Login</servlet-class>
</servlet>
<servlet>
<servlet-name>WelcomeServlet</servlet-name>
<servlet-class>WelcomeServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Login</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>WelcomeServlet</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

<welcome-file-list>
<welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>
```



SendRedirect in servlet

The **sendRedirect()** method of **HttpServletResponse**interface can be used to redirect response to another resource, it may be servlet, jsp or html file.It accepts relative as well as absolute URL. It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

Difference between forward() and sendRedirect() method

There are many differences between the forward() method of RequestDispatcher and sendRedirect() method of HttpServletResponse interface. They are given below:

forward() method	sendRedirect() method
The forward() method works at server side.	The sendRedirect() method works at client side.
It sends the same request and response objects to another servlet.	It always sends a new request.
It can work within the server only.	It can be used within and outside the server.
Example: request.getRequestDispacher("servlet2").forward(request,response);	Example: response.sendRedirect("servlet2");

Syntax of sendRedirect() method

```
public void sendRedirect(String URL)throws IOException;
```

Example of sendRedirect() method

```
response.sendRedirect("http://www.javatpoint.com");
```

Example of sendRedirect method in servlet

In this example, we are redirecting the request to the google server. Notice that sendRedirect method works at client side, that is why we can our request to anywhere. We can send our request within and outside the server.

DemoServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();

response.sendRedirect("http://www.google.com");

pw.close();
}}
```

Creating custom google search using sendRedirect

In this example, we are using sendRedirect method to send request to google server with the request data.

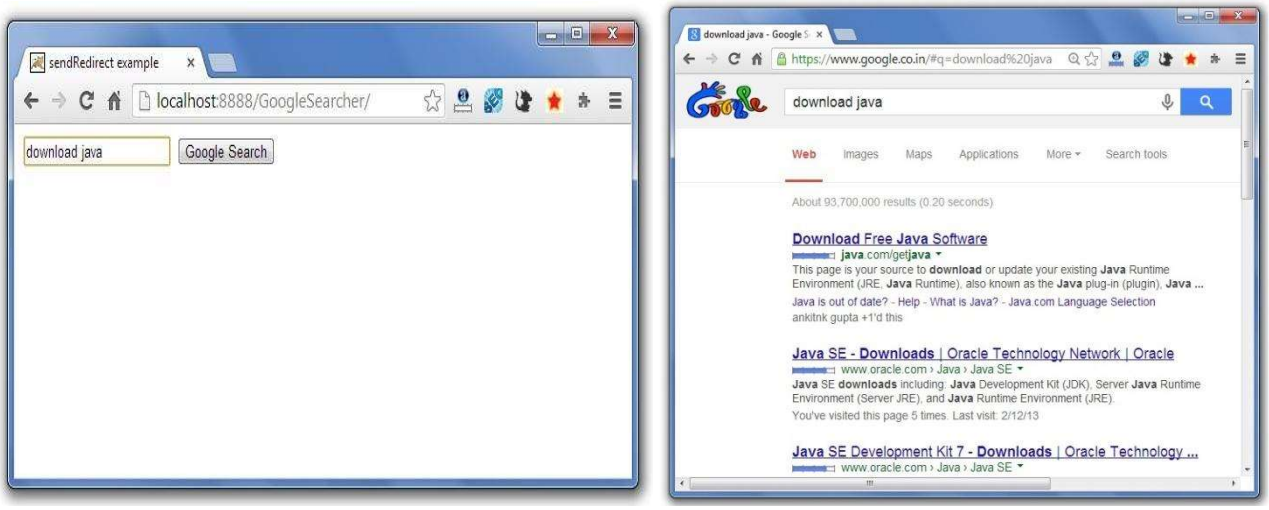
index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>sendRedirect example</title>
</head>
<body>

<form action="MySearcher">
<input type="text" name="name">
<input type="submit" value="Google Search">
</form>

</body>
</html>
```

MySearcher.java

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MySearcher extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

                throws ServletException, IOException {

            String name=request.getParameter("name");
            response.sendRedirect("https://www.google.co.in/#q="+name);
        }
}
```

Output



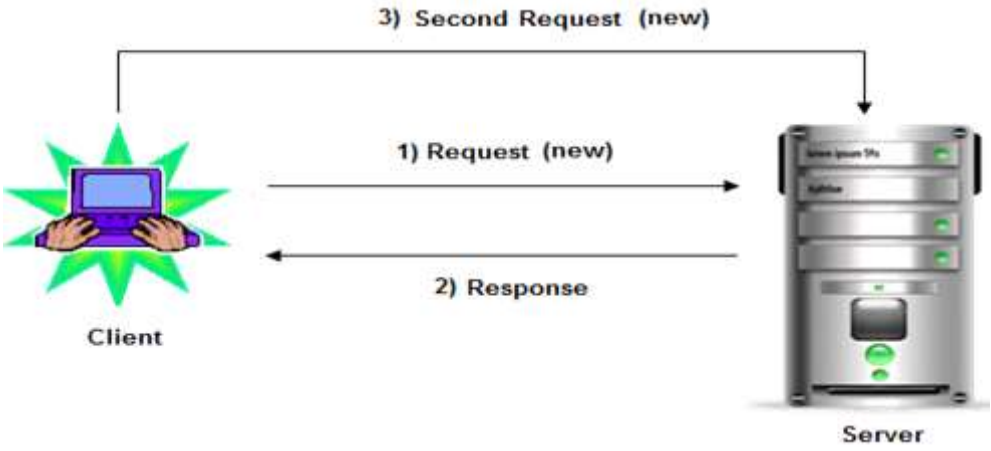
SESSION TRACKING IN SERVLET

Session is the conversion of user within span of time or particular interval of time. **Tracking** is the recording of the thing under session.

Session Tracking is remembering and recording of client conversion in span of time. It is also called as session management. If web application is capable of remembering and recording of client conversion in span of time then that web application is called as **Stateful web application**.

Need for Session Tracking:

- Http protocol is stateless, to make stateful between client and server we need Session Tracking.
- Session Tracking is useful for online shopping, mailing application, E-Commerce application to track the conversion.
- Http protocol is stateless, that means each request is considered as the new request. You can see in below image.



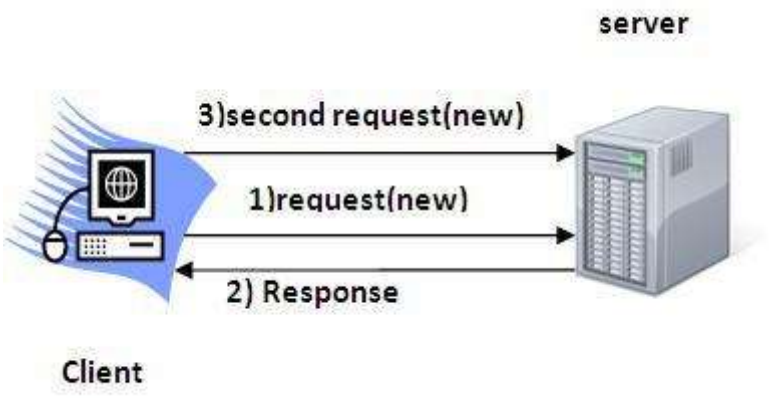
Session Tracking is used to recognize the particular user.

Why Http is design as stateless protocol ?

If Http is stateful protocol for multiple requests given by client to web application single connection will be used between browser and web server across the multiple requests. This may make clients to engage connection with web server for long time event though the connection are ideal. Due to this the web server reach to maximum connections even though most of its connection are idle. To overcome this problem Http is given as stateless.

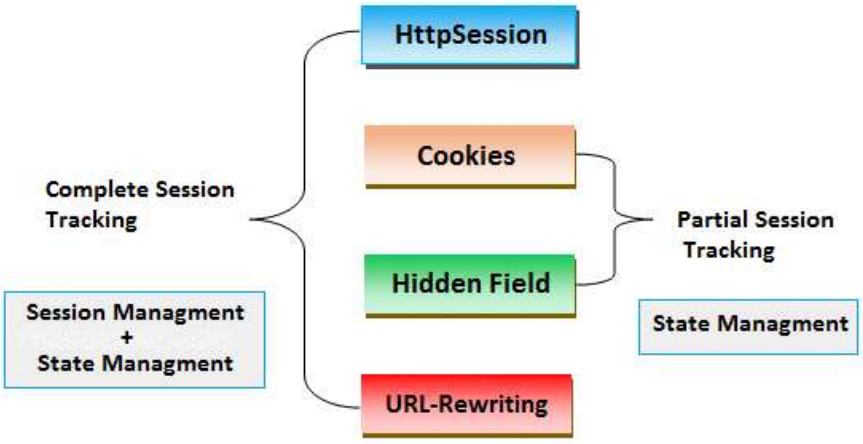
SESSION TRACKING TECHNIQUES

Session simply means a particular interval of time. **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet. 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:



To recognize the user It is used to recognize the particular user. Servlet technology allows four technique to track conversion, they are;

- Cookies
- URL Rewriting
- Hidden Form Field
- HttpSession



COOKIES

Cookies are text files stored on the client computer and they are kept for various information like name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

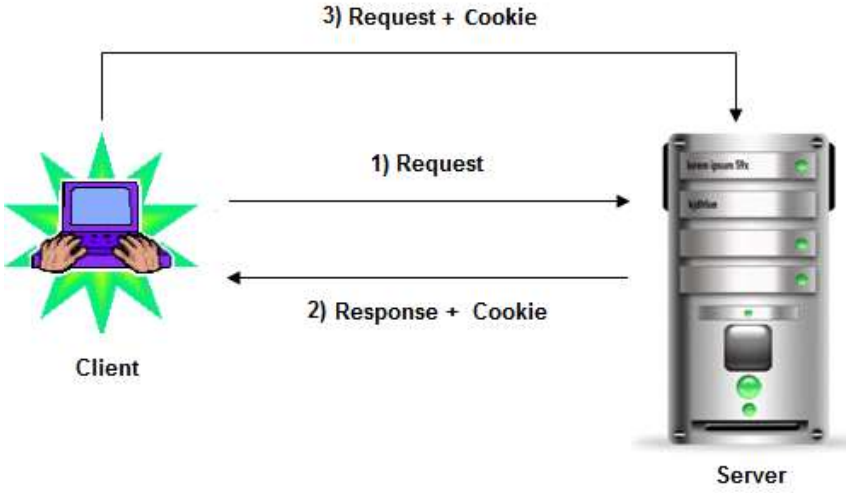
Cookies are created using Cookie class present in Servlet API. Cookies are added to response object using the addCookie() method. This method sends cookie information over the HTTP response stream. getCookies() method is used access the cookies that are added to response object.

In Http Session technique, container internally generates a cookies for transferring the session ID between server and client. Apart from container generated cookie a servlet programmer can also generate cookies for storing the data for a client.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response

from the servlet. So, cookie is stored in the cache of the browser (chrome, firefox) at client side. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



When session ID is not required and when less number of input values are submitted by client in that case in place of using HttpSession Technique we can use cookies Technique to reduce the burden on server.

Points to Remember

- Cookies is persistence resource which is stores at client location.
- We can store 3000 cookies in cookies file at a time.
- The cookies are introduced by netscape communication.
- Cookies files exist up to 3 year.
- Size of cookies is 4 kb.

Type of Cookies

There are two types of cookies, those are given below;

- **In-memory cookies:** By default cookie is in-memory cookie. This type of cookie lives until that browser is destroy(close). It is valid for **single session** only. It is removed each time when user closes the browser.
- **Persistent cookies:** Persistent cookie lives on a browser until its expiration time is reached it means, even though the browser is close or reopen but still the cookie exists on the browser. It is valid for **multiple session**. It is not removed each time when user closes the browser. It is removed only if user logout or signout.

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides some constructor and methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	Used for constructs a cookie.
Cookie(String name, String value)	Used for constructs a cookie with a specified name and value.

Methods of Cookie class

Methods	Description
public void setMaxAge(int expiry)	It is used for Sets the maximum age of the cookie in seconds.
public String getName()	It is used for Returns the name of the cookie. The name cannot be changed after creation.

public String getValue()	It is used for Returns the value of the cookie.
public void setName(String name)	It is used for changes the name of the cookie.
public void setValue(String value)	It is used for changes the value of the cookie.
public void addCookie(Cookie ck)	It is method of HttpServletResponse interface which is used to add cookie in response object.
public Cookie[] getCookies()	It is method of HttpServletRequest interface which is used to return all the cookies from the browser.

Create Cookies: To create cookies you need to use Cookie class of javax.servlet.http package.

Syntax

```
Cookie c=new Cookie(name, value);
// here name and value are string type
```

Add Cookies: To add a cookie to the response object, we use addCookie() mehtod.

Syntax

```
Cookie c=new Cookie();      //creating cookie object
response.addCookie(c1); //adding cookie in the response
```

Read Cookies for browser: To read Cookies from browser to a servlet, we need to call getCookies methods given by request object and it returns an array type of cookie class.

Syntax

```
response.addCookie(c1);
Cookie c[]=request.getCookie();
```

Advantage of Cookie

- Simplest technique of maintaining the state.
- Cookie are maintained at client side so they do not give any burden to server.
- All server side technology and all web server, application servers support cookies.
- Persistent cookies can remember client data during session and after session with expiry time.

Limitation of Cookie

- It will not work if cookie is disabled from the browser.
- Cookies are text files, It does not provides security. Anyone can change this file.
- With cookies need client support that means if client disable the cookies then it does not store the client location.
- Cookies cannot store java objects as values, they only store text or string.

Example of session tracking by using Cookies

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/> <br/>
<input type="submit" value="continue"/>
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {

  public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String n=request.getParameter("userName");
    out.print("Welcome "+n);

    Cookie ck=new Cookie("uname",n);//creating cookie object
    response.addCookie(ck);//adding cookie in the response

    //creating submit button
    out.print("<form action='servlet2'>");
    out.print("<input type='submit' value='continue'>");
    out.print("</form>");

    out.close();

        }catch(Exception e){System.out.println(e);}
  }
}
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response){
    try{

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    Cookie ck[]=request.getCookies();
    out.print("Hello "+ck[0].getValue());

    out.close();

         }catch(Exception e){System.out.println(e);}
    }
}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
```

```
<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

HIDDEN FORM FIELD

Tracking client conversion using Html hidden variables in secure manner is known as hidden form field.

How to use Hidden Form Field ?

In Hidden Form Field we are use html tag is <input type="hidden"> and with this we assign session ID value.

Syntax

```
<input type="hidden" name="uname" value="porter">
```

Hidden Form Field Advantage

- Basic knowledge of html is enough to work with this technique.
- It will always work whether cookie is disabled or not.
- Hidden boxes resides in web pages of browser window so they do not provide burden to the server.
- This technique can be used along with all kind of web server or application server.

Hidden Form Field Disadvantage

- More complex than URL Rewriting.
- It is maintained at server side.
- Extra form submission is required on each pages.
- Hidden form field cannot store java object as values. They only store text value
- It also increases the network traffic because hidden boxes data travels over the network along with request and response.
- Hidden box does not provide data security because their data can be view through view source option.

Example of session tracking by using Hidden Form Field

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/> <br/>
<input type="submit" value="continue"/>  </form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response){
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);

        //creating form that have invisible textfield
        out.print("<form action='servlet2'>");
        out.print("<input type='hidden' name='uname' value='"+n+"'>");
        out.print("<input type='submit' value='continue'>");
        out.print("</form>");
        out.close();

            }
               catch(Exception e){System.out.println(e);}
    }
}
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SecondServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //Getting the value from the hidden field
        String n=request.getParameter("uname");
        out.print("Hello "+n);

        out.close();
        }
              catch(Exception e){System.out.println(e);}
    }
}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
```

```
</web-app>
```

URL Rewriting

URL Rewriting track the conversion in server based on unique session ID value. 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&??
```

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.

When use URL Rewriting ?
If the client has disabled cookie in the browser then cookie will not work for session management. In that case we can use URL rewriting technique for session management. URL rewriting will always work.

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

Dis-advantage of URL Rewriting

- Generate more network traffic.
- It will work only with links.
- It can send Only textual information.
- Less secure because query string in session id displace on address bar.

Example of session tracking by using URL Rewriting

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/> <br/>
<input type="submit" value="continue"/>
</form>
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

FirstServlet.java

```
public class FirstServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);
```

```
        HttpSession session=request.getSession();
        session.setAttribute("uname",n);

        out.print("<a href='servlet2'>visit</a>");

        out.close();

                }catch(Exception e){System.out.println(e);}
    }
}
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //getting value from the query string
        String n=request.getParameter("uname");
        out.print("Hello "+n);

        out.close();

           }
                    catch(Exception e){System.out.println(e);}
    }
}
```

web.xml

```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

HttpSession

HttpSession is another kind of session management technique. In this technique a session object is created at server side for each client. Session is available till the session time out or until the client log out. The default session time is 30 minutes and can configure explicit session time in web.xml file.

Configure session time in web.xml

Example

```
<web-app>
<session-config>
<session-timeout>40</session-timeout>
</session-config>
</web-app>
```

HttpSession Api

Http session is an interface define in java.http package.

Getting session object

Example

```
HttpSession hs=req.getSession();  // create new session object
```

Methods of HttpSession interface

Method	Description
public HttpSession getSession():	It returns the current session associated with this request, or if the request does not have a session, creates one.
public HttpSession getSession(boolean create)	It returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.
public String getId()	It returns a string containing the unique identifier value.
public long getCreationTime()	It returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
public long getLastAccessedTime()	It returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
public void invalidate()	Invalidates this session then unbinds any objects bound to it.

Disadvantage of HttpSession

- Http session objects allocate memory on the server so this increase burden on the server.
- If cookies are restricted coming to browser window this technique fails to perform session tracking.

Example of session tracking by using httpsession

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/> <br/>
<input type="submit" value="continue"/>
```

```
</form>
```

FirstServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response){
        try{

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String n=request.getParameter("userName");
        out.print("Welcome "+n);

        HttpSession session=request.getSession();
        session.setAttribute("uname",n);

        out.print("<a href='servlet2'>visit</a>");

        out.close();

                }catch(Exception e){System.out.println(e);}
    }
}
```

SecondServlet.java

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SecondServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
        try{
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(false);
        String n=(String)session.getAttribute("uname");
        out.print("Hello "+n);
         out.close();
                }
catch(Exception e){System.out.println(e);}
 }   }
```

web.xml

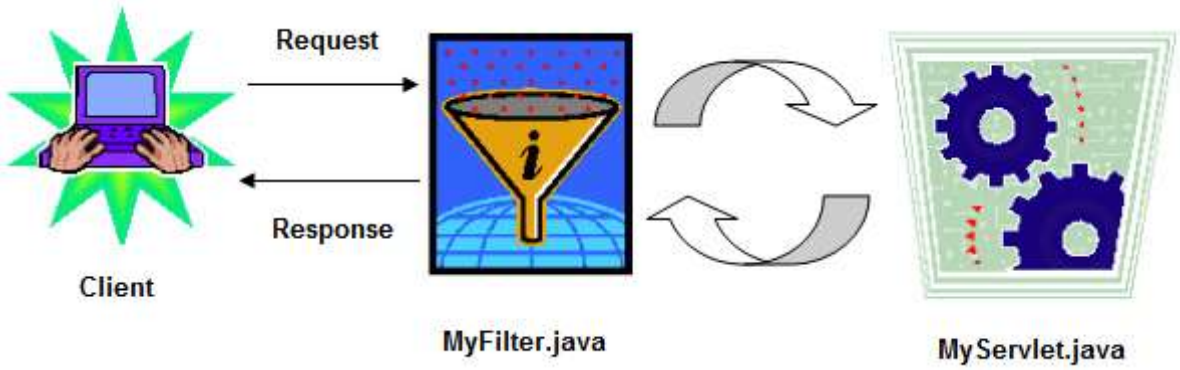
```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
```

```
<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

SERVLET FILTER

A **Filter** is an object that is invoked at the pre-processing and post-processing of a request. It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc.

The Servlet filter is plug-gable, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the Servlet. So maintenance cost will be less.



Usage of Filter

- Recording all detail about incoming requests
- Logs the IP addresses of the computers from which the requests originate
- Conversion
- Data compression
- Encryption and decryption
- Input validation etc.

Advantage of Filter

- Filter is plug-gable.
- One filter don't have dependency onto another resource.
- Less Maintenance

Filter API

Like Servlet filter have its own API. There are three interfaces in Servlet API to implement filter concept in a java web application. The javax.servlet package contains the three interfaces of Filter API.

1) Filter
2) FilterChain
3) FilterConfig

1) Filter interface

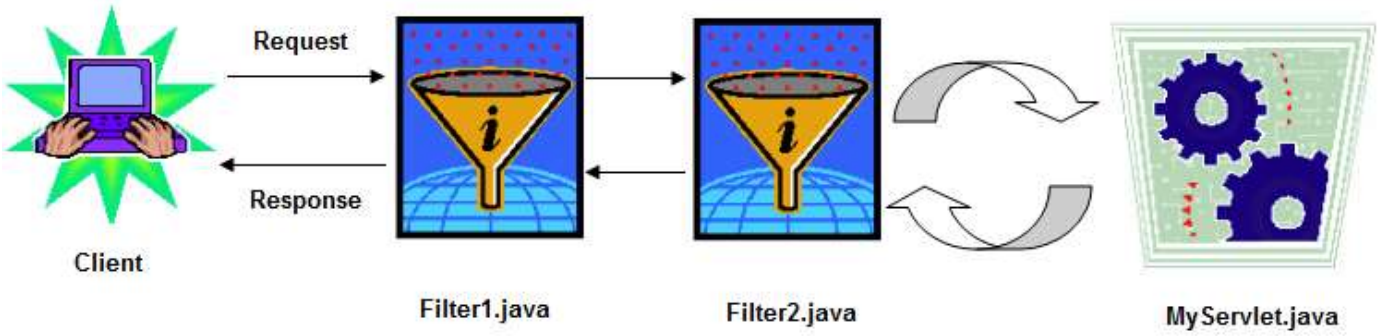
For creating any filter, we must implement the Filter interface. Filter interface have three life cycle

Method	Description
public void init(FilterConfig config)	init() method is invoked only once. It is used to initialize the filter.
public void doFilter(HttpServletRequest request,HttpServletResponse response, FilterChain chain)	doFilter() method is invoked every time when user request to any resource, to which the filter is mapped. It is used to perform filtering tasks.
public void destroy()	This is invoked only once when filter is taken out of the service.

2) FilterChain interface

whenever more than one filter are used in Servlet then it called as Filter Chain. The object of FilterChain is responsible to invoke the next filter or resource in the chain. This object is passed in the doFilter method of Filter interface. The FilterChain interface contains only one method:

- **public void doFilter(HttpServletRequest request, HttpServletResponse response):** it passes the control to the next filter or resource.



How to apply a Filter to Servlet

We can define filter same as Servlet. The elements of filter and filter-mapping.

Syntax

```
<web-app>

<filter>
<filter-name>...</filter-name>
<filter-class>...</filter-class>
</filter>

<filter-mapping>
<filter-name>...</filter-name>
<url-pattern>...</url-pattern>
</filter-mapping>

</web-app>
```

For mapping filter we can use, either url-pattern or Servlet-name. The url-pattern elements has an advantage over Servlet-name element i.e. it can be applied on servlet, JSP or HTML.

FilterConfig

An object of FilterConfig is created by the web container. This object can be used to get the configuration information from the web.xml file.

Methods of FilterConfig interface

There are following 4 methods in the FilterConfig interface.

- **public void init(FilterConfig config):** init() method is invoked only once it is used to initialize the filter.
- **public String getInitParameter(String parameterName):** Returns the parameter value for the specified parameter name.
- **public java.util.Enumeration getInitParameterNames():** Returns an enumeration containing all the parameter names.
- **public ServletContext getServletContext():** Returns the ServletContext object.

Difference between Forward and Redirect

Forward	Redirect
In forwarding, the destination resource must be java enabled resource only.	In redirection, the destination resource can be either java or non-java resource also.
In forwarding, both source and destination resource must run within the same server. It is not possible to communicate across the server.	In redirection, it is possible to communicate, either within the serve or even across the server.
In forwarding, both the data and control are forwarded to destination (by default).	In redirection, only control is redirected, but not the data (by default).

JAVA SERVER PAGES(JSP)

JSP technology is used to create web application just like Servlet technology. It can be thought of as an extension to servlet because it provides more functionality than servlet such as expression language, jstl etc. A JSP page consists of HTML tags and JSP tags. The jsp pages are easier to maintain than servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tag etc.

Features of JSP

JSP are tag based approach to develop dynamic web application. JSP have all the features of Servlet because it is internally Servlet. It has following features :

- **Extension to Servlet:** JSP is Extension to Servlet, it have all the features of servlet and it have also implicit objects, predefined tags, expression language and Custom tags in JSP, that makes JSP easy to develop any application.
- **Powerful:** These are internally Servlet, means consists byte code, so that all java features are applicable in case of jsp like robust, dynamic, secure, platform independent.
- **Portable:** JSP tags will process and execute by the server side web container, So that these are browser independent and j2ee server independent.
- **Flexible:** Allows to defined custom tags, the developer can fill conferrable to use any kind, framework based markup tags in JSP.
- **Easy:** JSP is easy to learn, easy to understand and easy to develop. JSPs are more convenient to write than Servlets because they allow you to embed Java code directly into your HTML pages, in case of servlets you embed HTML inside of Java code.
- **Less code than Servlet:** In JSP, we can use a lot of tags such as action tags, jstl, custom tags etc. that reduces the code.

JSP TRANSLATION

Each JSP page is internally translated into an equivalent Servlet. Every JSP tag written in the page will be internally converted into an equivalent java code by the containers this process is called **translation**.
There are two phases are occurs in JSP;

- **Translation phase:** It is process of converting jsp into an equivalent Servlet and then generating class file of the Servlet.

- **Request processing phase:** It is process of executing service() of jsp and generating response data on to the browser.

When first time a request to a jsp received the translation phase occurs first and then after service phase will be executed. From next request to the jsp, only request processing phase is executed but translation phase is not because jsp is already translated.

In following two cases only translation occurs;

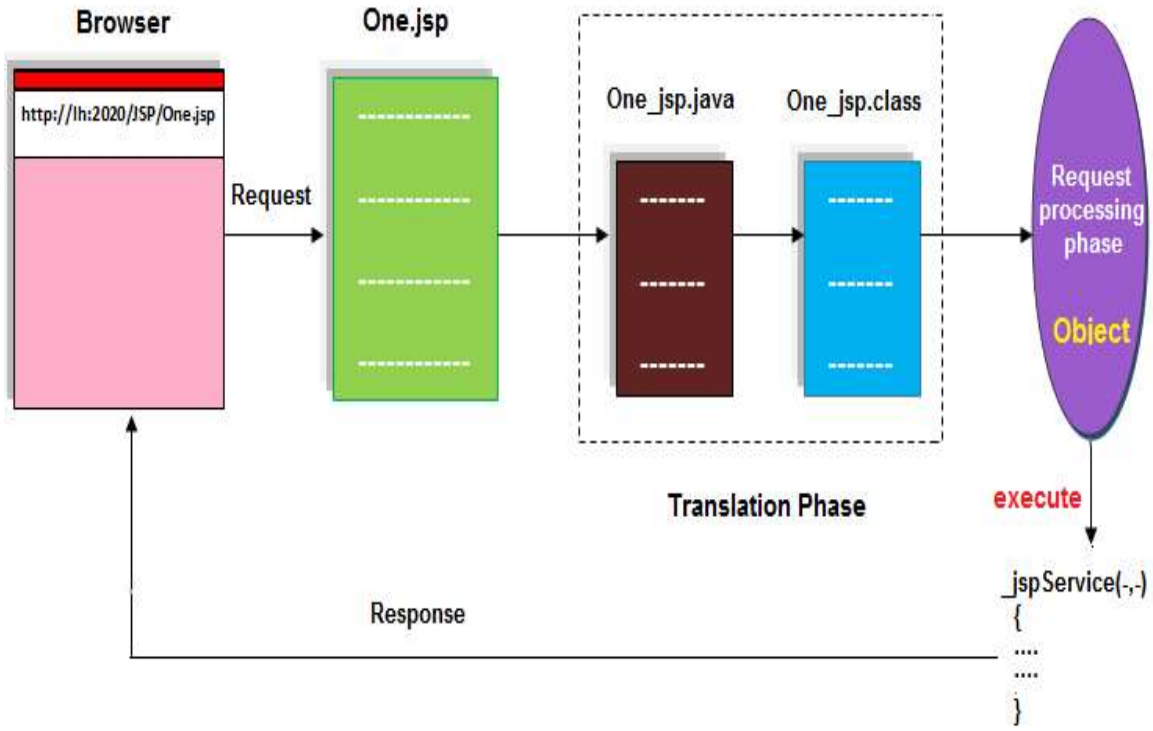
- When first request is a given to the jsp.
- When a jsp is modified.

If a request is given to the jsp after it has modified, then again translation phase is executed first and after that request processing phase will be executed.

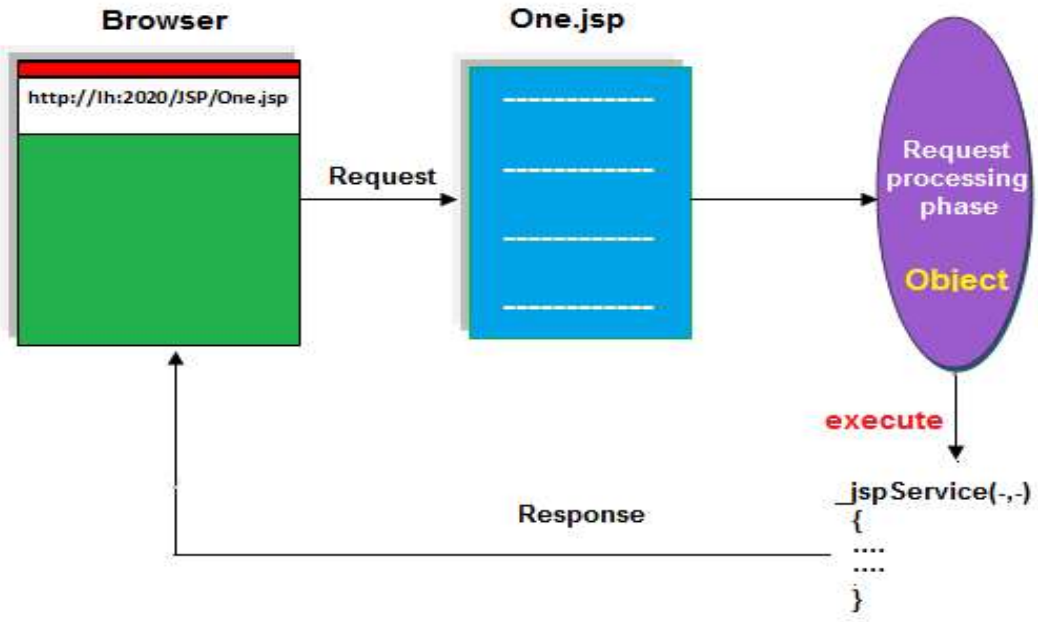
Note:

- If we shutdown and restart the Servet then only request processing phase will be executed, Because when we shutdown the server java and class files are not deleted from the server.
- If we remove the class file from server then partial translation occurs..
- If we remove both java and class file from server then again translation occurs.
- When a jsp is translated into an equivalent Servlet then that Servlet extends some base class provided by the server, but internally it extends HttpServlet.

When first request is come or modify jsp page.



From next request



Configuring JSP File

Configuring a JSP into a web.xml file is optional because JSP is a public file to the web application. A JSP called a public file and servlet is called a private file of the web application. Because JSP files stored in root directory of the web application and Servlet class file stored in sub directory of the web application. Because JSP is the public file, we can directly send a request from a browser by using its file name.

If we want to configure a JSP in web.xml file the xml elements are same as Servlet-configuration. We need to replace <servlet-class> element with <jsp-file>

Syntax

```
<web-app>
<servlet>
<servlet-name>test</servlet-name>
<jsp-file>/One</jsp-file>
</servlet>
<servlet-mapping>
<servlet-name>test</>
<url-pattern>/srv1</url-pattern>
</servlet-mapping>
</web-app>
```

Note: If we configure a jsp in web.xml then we can send the request to the jsp either by using jsp filename or by using its url-pattern.

Example

```
httpp://localhost:2014/root/One.jsp
                        or
httpp://localhost:2014/root/srv1
```

Difference Between Servlet And JSP

When we need Servlet and JSP ?

With less request processing logic with more response generation logic we need JSP.

Problem with Servlet

If any changes in static html code of Servlet, the entire Servlet need recompilation, redeployment, needs server restart. Any modification in Servlet needs recompilation because both request processing logic and response generation logic are tight coupled.

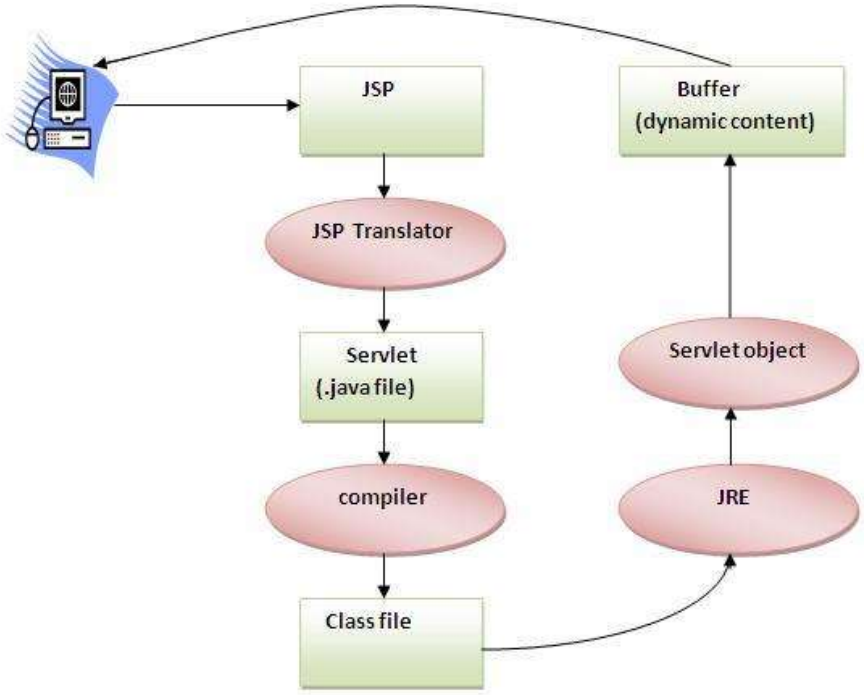
	Servlet	JSP
1	Servlet is faster than jsp	JSP is slower than Servlet because it first translate into java code then compile.
2	In Servlet, if we modify the code then we need recompilation, reloading, restarting the server> It means it is time consuming process.	In JSP, if we do any modifications then just we need to click on refresh button and recompilation, reloading, restart the server is not required.
3	Servlet is a java code.	JSP is tag based approach.
4	In Servlet, there is no such method for running JavaScript at client side.	In JSP, we can use the client side validations using running the JavaScript at client side.
5	To run a Servlet you have to make an entry of Servlet mapping into the deployment descriptor file i.e. web.xml file externally.	For running a JSP there is no need to make an entry of Servlet mapping into the web.xml file externally, you may or not make an entry for JSP file as welcome file list.
6	Coding of Servlet is harden than jsp.	Coding of jsp is easier than Servlet because it is tag based.
7	In MVC pattern, Servlet plays a controller role.	In MVC pattern, JSP is used for showing output data i.e. in MVC it is a view.
8	Servlet accept all protocol request.	JSP will accept only http protocol request.
9	In Servlet, service() method need to override.	In JSP no need to override service() method.
10	In Servlet, by default session management is not enabled we need to enable explicitly.	In JSP, session management is automatically enabled.
11	In Servlet we do not have implicit object. It means if we want to use an object then we need to get object explicitly form the servlet.	In JSP, we have implicit object support.
12	In Servlet, we need to implement business logic, presentation logic combined.	In JSP, we can separate the business logic from the presentation logic by uses javaBean technology.
13	In Servlet, all package must be imported on top of the servlet.	In JSP, package imported anywhere top, middle and bottom.

LIFE CYCLE OF A JSP PAGE

The JSP pages follows the below phases:

- Translation of JSP Page
- Compilation of JSP Page
- Classloading (class file is loaded by the classloader)
- Instantiation (Object of the Generated Servlet is created).
- Initialization (jspInit() method is invoked by the container).
- Request processing (_jspService() method is invoked by the container).
- Destroy (jspDestroy() method is invoked by the container).

Note: jspInit(), _jspService() and jspDestroy() are the life cycle methods of JSP.



As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator. The JSP translator is a part of web server that is responsible to translate the JSP page into servlet. After that Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happens in servlet is performed on JSP later like initialization, committing response to the browser and destroy.

Creating a simple JSP Page

To create the first jsp page, write some html code as given below, and save it by .jsp extension. Save this file as index.jsp. Put it in a folder and paste the folder in the web-apps directory in apache tomcat to run the jsp page.

index.jsp

The simple example of JSP using the scriptlet tag to put java code in the JSP page. .

```
1. <html>
2. <body>
3. <% out.print(2*5); %>
4. </body>
5. </html>
```

It will print **10** on the browser.

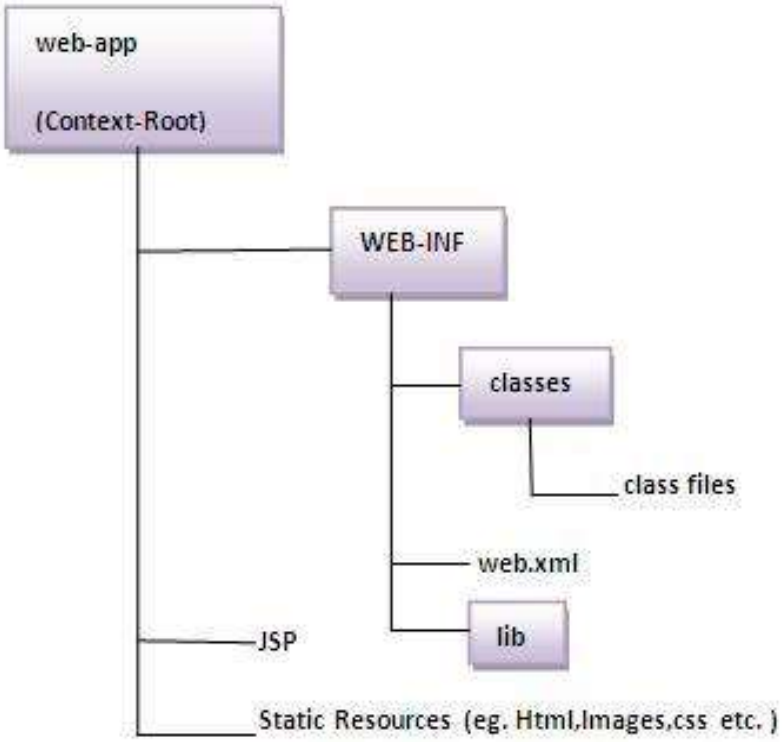
Running a simple JSP Page

Follow the following steps to execute this JSP page:

- Start the server
- put the jsp file in a folder and deploy on the server
- visit the browser by the url http://localhost:portno/contextRoot/jspfile
  e.g. http://localhost:8888/myapplication/index.jsp

DIRECTORY STRUCTURE OF JSP

The directory structure of JSP page is same as servlet. It contains the jsp page outside the WEB-INF folder or in any directory.



The JSP API
The JSP API consists of two packages:

1. javax.servlet.jsp
2. javax.servlet.jsp.tagext

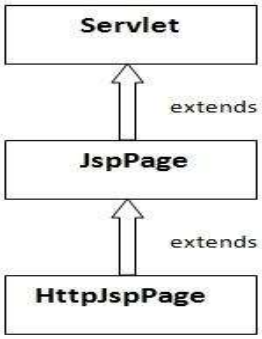
javax.servlet.jsp package
The javax.servlet.jsp package has two interfaces and classes. The two interfaces are as follows:

1. JspPage
2. HttpJspPage

The classes are as follows:

- JspWriter
- PageContext
- JspFactory
- JspEngineInfo
- JspException
- JspError

The JspPage interface: According to the JSP specification, all the generated servlet classes must implement the JspPage interface. It extends the Servlet interface. It provides two life cycle methods.



Methods of JspPage interface

1. **public void jspInit():** It is invoked only once during the life cycle of the JSP when JSP page is requested firstly. It is used to perform initialization. It is same as the init() method of Servlet interface.
2. **public void jspDestroy():** It is invoked only once during the life cycle of the JSP before the JSP page is destroyed. It can be used to perform some clean up operation.

The HttpJspPage interface: The HttpJspPage interface provides the one life cycle method of JSP. It extends the JspPage interface.

Method of HttpJspPage interface:

1. **public void _jspService():** It is invoked each time when request for the JSP page comes to the container. It is used to process the request. The underscore _ signifies that you cannot override this method.

Scripting Element

In JSP, java code can be written inside the jsp page using the scriptlet tag. JSP Scripting element are written inside **<% %>** tags. These code inside <% %> tags are processed by the JSP engine during translation of the JSP page. Jsp scripting elements are classified into two types are;

1. Language Based Scripting Element
2. Advance Scripting Elements (Expression Language)

1. **Language Based Scripting Element:** These are used to defined script in jsp page, this is traditional approach to define the script in jsp page. Language Based Scripting Element are classified into 4 types, they are;
    1. Comment Tag
    2. Declaration Tag
    3. Expression Tag
    4. Scriptlet Tag

Scripting Element Detail

	Start tag	Purpose	End tag	Inside class scope	Inside JSP Scope	Session
JSP Declaration	<%!	Declaration variable and method	%>	yes	no	yes
JSP Expression	<%=	Display response on browser	%>	no	yes	no
JSP Scriptlet	<%	Defining java code	%>	no	no	yes
JSP Comment	<%--	Comment discription	--%>	yes	yes	not applicable

1. **JSP Comment:** This is used to define comment description in jsp page. In jsp page we can insert three types of comments they are
    - Jsp comment
    - Html comment
    - Java comment

- **JSP Comment:** This type of comment is called as hidden comment, because it is invisible when a jsp is translated into a servlet internally.

Syntax

```
<%-- Comment description %>
```

- **Html Comment:** Html comment tag is common for Html, xml and jsp. In a jsp page if we write html comment these comment are visible in the _jspService(-,-) of the internal Servlet.

Syntax

```
<!--  Comment description -->
```

- **Java Comment:** We can write a single or multiline comment of java in a scriptlet tag. Java comment are allowed only inside scriptlet tags because we can insert java code in a jsp only at a scriptlet tags.

Syntax

```
//  Single line comment
/*
Multiline comment*/
```

2. **JSP Declaration Tag:** This is used to declare variable and methods in jsp page will be translate and define as class scope declaration in .java file.
   The variable and methods will become global to that jsp. It means in that jsp we can use those variables anywhere and we can call those method anywhere.

   At the time of translation container inserts the declaration tag into the class. So the variables become instants variables and methods will become instants methods.

   The code written inside the jsp declaration tag is placed outside the service() method of auto-generated Servlet, so it does not get memory at each request.

Syntax

```
<%! variable declaration or method declaration %>
```

3. **JSP Expression Tag:** Expression tag is used, to find the result of the given expression and send that result back to the client browser. In other words; These are used to show or express the result or response on browser.
   We can use this as a display result of variable and invoking method. Each Expression tag of jsp will be internally converted into **out.print()** statement. In jsp out is an implicit object.

   The expression tags are converted into out.print() statement and insert into the _jspService(-,-) of the servlet class by the container.

   The expression tags written in a jsp are executed for each request, because _jspService(-,-) is called for each request.

   One expression tag should contains one java expression only. In a jsp we can use expression tag for any number of times.

Syntax

```
<%= expression %>
```

Note: In jsp, the implicit object are allowed into the tags, which are inserted into _jspService(-,-). An expression tag goes to _jspService(-,-) only. So implicit object are allowed in the expression tag.

4. **JSP Scriptlet Tag:** It use to define or insert java code in a jsp. Every java statement should followed by semicolon (;). It will be place into jspService() method.

   The scriptlet tags goes to _jspService(-,-), during translation. It means scriptlet tags are executed for each request. Because _jspService(-,-) is called for each request.

   We can use implicit object of jsp in a scriptlet tag also because scriptlet tag goes to _jspService(-,-) only. In a jsp implicit object are allowed in expression tags and scriptlet tags but not in the declaration tags.

Syntax

```
<% java code %>
```

JSP IMPLICIT OBJECTS

JSP provides standard or predefined implicit objects, which can use directly in JSP page using **JSP Scriptlet**. The implicit objects are Servlet API class type and created by JSP containers

There are **9 JSP implicit objects**. These objects are created by the web container (JSP containers) that are available to all the jsp pages.

The available implicit objects are out, request, response, page, pageContext, exception, config, session and application.

List of all 9 implicit object are

	Class type	Object
1	HttpServletRequest	request
2	HttpServletResponse	response
3	ServletConfig	config
4	ServletContex	application
5	HttpSession	session
6	JspWriter/PrintWriter	out
7	Exception	exception
8	PageContext	pagecontext
9	Object	page

All implicit objects of jsp are accessible within the expression and scriptlet of the jsp, but not accessible in the declaration tags of the jsp.

In a jsp session and exception object are not available always. Because session object depends on session attribute of the page directive and exception depends on isErrorPage attribute of the page directive. In a jsp, session object is available, if session equals to true in page directive. In a jsp an exception object is available, if isErrorPage is equal to true in page directive.

1. **Request Implicit Object:** The JSP request is an implicit object of type HttpServletRequest i.e. created for each jsp request by the web container. It can be used to get request information such as parameter, header information, remote address, server name, server port, content type, character encoding etc. It can also be used to set, get and remove attributes from the jsp request scope.

   Example of request implicit object where we are printing the name of the user with welcome message.

Example of JSP request implicit object

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);  %>
```

2. **Response Implicit Object:** In JSP, response is an implicit object of type HttpServletResponse. The instance of HttpServletResponse is created by the web container for each jsp request. It can be used to add or manipulate response such as redirect response to another resource, send error etc.

   Example of response implicit object where we are redirecting the response to the Google.

Example of response implicit object

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
response.sendRedirect("http://www.google.com");
%>
```

3. **Config Implicit Object:** config object is an implicit object of type ServletConfig and it is created by the container, whenever servlet object is created. This object can be used to get initialization parameter for a particular JSP page. config object is created by the web container for every jsp page. It means if a web application has three jsp pages then three config object are created.

   In jsp, it is not mandatory to configure in web.xml. If we configure a jsp in web.xml then the logical name and init parameter given in web.xml file are stored into config object by the container.
   config object is accessible, only if the request is given to the jsp by using its url pattern, but not with name of the jsp.

   config object is accessible, only if the request is given to the jsp by using its url pattern, but not with name of the jsp.

Example of config implicit object

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

web.xml file

```
<web-app>

<servlet>
<servlet-name>Home</servlet-name>
<jsp-file>/welcome.jsp</jsp-file>

<init-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>
</servlet>

<servlet-mapping>
<servlet-name>Home</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>

</web-app>
```

welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=config.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

4. **Page Implicit Object:** In JSP, page is an implicit object of type Object class. When a jsp is translated to an internal Servlet, we can find the following statement in the service() method of servlet. Object page=this. For using this object it must be cast to Servlet type. For example:

Example

```
<% (HttpServlet)page.log("message"); %>
```

Since, it is of type Object it is less used because you can use this object directly in jsp. For example:

Example

```
<% this.log("message"); %>
```

5. **Session Implicit Object:** In JSP, session is an implicit object of type HttpSession. This object used to set, get or remove attribute or to get session information.

Example:

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<% String name=request.getParameter("uname");
out.print("Welcome "+name);
session.setAttribute("user",name);
<a href="second.jsp">second jsp page</a> %>
```

One.jsp

```
<%
String name=(String)session.getAttribute("user");
out.print("Hello "+name);
%>
```

6. **Exception Implicit Object:** JSP, exception is an implicit object of type java.lang.Throwable class. This object can be used to print the exception. But it can only be used in error pages.

Example:

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
response.sendRedirect("http://www.google.com");
%>
```

7. **Application Implicit Object:** In JSP, application is an implicit object of type ServletContext. This application object in the Servlet programming is ServletContext. For all jsp in a web application, there must be a single application object with application object we can share the data from one JSP to any other JSP in the web application.

   The instance of ServletContext is created only once by the web container when application or project is deployed on the server. This object can be used to get initialization parameter from configuration file (web.xml). It can also be used to get, set or remove attribute from the application scope.

Example:

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
```

```
</form>
```

welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=application.getInitParameter("dname");
out.print("driver name is="+driver);
 %>
```

web.xml

```
<web-app>
<servlet>
<servlet-name>One</servlet-name>
<jsp-file>/welcome.jsp</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>One</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>

<context-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>
</web-app>
```

8. **PageContext Implicit Object:** In JSP, pageContext is an implicit object of type PageContext class. The pageContext object can be used to set, get or remove attribute from one of the following scopes.
   - page
   - request
   - session
   - application

   In JSP, page scope is the default scope.

Example of pageContext implicit object

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("Welcome "+name);
pageContext.setAttribute("user",name,PageContext.SESSION_SCOPE);
<a href="One.jsp">One jsp page</a> %>
```

One.jsp

```
String name=(String)pageContext.getAttribute("user",
                                        PageContext.SESSION_SCOPE);
out.print("Hello "+name);
```

JSP DIRECTIVE ELEMENTS

The directive elements are used to do page related operation during page translation time. JSP supports 3 types of directive elements, they are:

1. Page directive
2. Include directive
3. Taglib directive

Syntax of directive elements

```
<@ directive attribute="value" %>
```

1. **Page Directive:** The page directive defines attributes that apply to an entire JSP page. This element is used to define default of explicit operation to the jsp.

Syntax

```
<@ page attribute="value" %>
```

Attributes of JSP page directive

- import
- contentType
- extends
- language
- buffer
- info
- isELIgnored
- isThreadSafe
- autoFlush
- session
- pageEncoding
- errorPage
- isErrorPage

2. **Include Directive:** It is used to include some other pages into the JSP document, it may be jsp file, html file or text file. This is known as static include because the target page is located and included at the time of page compilation. The jsp page is translated only once so it will be better to include static resource.

The advantage of Include directive is Code Re-usability

Syntax

```
<%@ include file="resourceName"%>
```

Example

```
<html>
<body>

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

</body>
</html>
```

Note: The include directive includes the original content, so the actual page size grows at run-time.

3. **Taglib Directive:** This is used to use custom tags in JSP page, here the custom tag may be user defined ot JSTL tag or struct, JSF,..... etc.

Syntax

```
<@ taglib uri="uri of the taglibrary" prefix="prefix of taglibrary"
%>
```

- The attribute of taglib directive is uri

Example

```
<html>
<body>

<%@ taglib uri="http://www.tutorial4us.com/customtag/tags"
prefix="mytag" %>

<mytag:currentDate/>

</body>
</html>
```

JSP Action Element

Action elements are used to performs specific operation using JSP container, like setting values into java class, getting values from java class. The JSP action elements classified into two types are;

1. JSP Standard Action Element
2. JSP Custom Action Element

1. **Standard Action Element:** Standard Action are given in JSP to separate the presentation logic and the business logic of the JSP but partial.
The name is given as a standard Action, because each action as a pre-defined meaning and as a programmer it is not possible to change the meaning of the tag.

Each Standard Action follows xml syntax, we do not have any equivalent html syntax. The standard action element followed by "JSP" prifix.

Syntax of Standard Action Element

```
<jsp: standard action name>
```

Standard Action are given by JSP are

- <jsp: include>
- <jsp: forward>
- <jsp: param>
- <jsp: params>
- <jsp: plugin>
- <jsp: useBean>
- <jsp: setProperty>
- <jsp: getProperty>
- <jsp: fallback>

- **Jsp Forward:** This action tag is used for forwarding a request from a jsp to another jsp or a servlet or a html.

When use JSP Forward tag
Generally, if a huge logic is required in a jsp then we divide that logic into multiple jsp pages and then we apply forwarding technique.

Note: If destination is a jsp or html then file name is required and if destination is a servlet then url pattern is required.

Save and Compile jsp program
JSP program must be save with the **.jsp** extension. And for compile jsp code simply follow step of compilation servlet code.

Example

```
<jsp:forward page="/srv1"/>
<jsp:forward page="home.jsp"/>
<jsp:forward page="index.html"/>
```

When a request is forwarded then along-with the request, automatically request parameters send from browser or also forwarded. If you want to attach additional parameters then we use <jsp:param> tag inside <jsp:forward> tag.

Example

```
<jsp:forward page="home.jsp">
<jsp:param name="p1" value="10"/>
<jsp:param name="p2" value="20"/>
</jsp:forward>
```

- **Jsp Include:** This action tag is used for including the response of one resource like a jsp or a servlet or a html into another jsp page. In jsp, we have two types of including,
  1. include directive
  2. include action

Main difference between include directive and include action

- Include directive is for static including and include action is for dynamic including.
- Internally a container uses RequestDispatcher's include method, for executing tag.

When use JSP Include Directive

We choose include directive when a destination is a static page like html and we choose include action, when a destination is a dynamic resource like a jsp or a servlet.

- **Jsp useBean: <jsp:useBean>** standard action tag is use to establish a connection between a jsp page and a java bean. In web applications of java, jsp and java bean communication is required in the following two cases:
  - In a real-time MVC project, a model class (business class) will set the data to a java bean and a jsp (view) will read the data from a bean and finally displays it on the browser. In this class jsp to a java bean communication is required.
  - If multiple jsp pages need common java logic then it separates that java code into a bean and then we call the bean from jsp. In this case also jsp to java bean communication is required.

  <jsp:useBean> tag is used for creating an object of a bean class in a jsp page.

  Every java class is not a java bean automatically. A class should have the following qualities to make it as a java bean.
  - Class must be public.
  - Class must contain default constructor.
  - Each private property of the class must have setter or getter or both methods.
  - A class can at most implement Serializable interface.

- **Jsp setProperty:** This action tag is to set an input value to set input value to a property on a variable of bean class by calling setter () method of the property.

  When use JSP setPriority tag

  When a request comes from browser, the protocol is http and it is unknown for the java bean. So, directly input values from browser can't be send to a java bean.

  The flow of setting input values is, a jsp page takes request from browser and it will set input values to bean using tag.
  <jsp:setProperty> must be used inside <jsp:useBean> tag.
  <jsp:setProperty> property contains four attributes, they are;

  Name and property attributes are mandatory. We shouldn't use param and value attribute at time. If request parameters names and variable name of java beans or matched then we can directly write <protperty="*">.

Attribute	Description
Name	Object name: The values of this attribute must be same as the value of id of the bean class
Property	variable name in bean class
Param	request parameter name
Value	static value

Name, property,value	allowed
Name ,property, param	allowed
Property, param	not allowed

Name, property,name, param	not allowed.

- **Jsp getProperty:** This action tag is used to read the value of a variable of bean class by calling its getter () method. This action tag must be written at outside of tag.

  Attribute of getProperty tag
  This action tag has only two attribute called name and property. In this tag property="*" is not allowed.

- **Jsp plugin: <jsp:plugIn>** action tag is given for integrating a jsp page with an applet.

  When use <jsp:plugIn> tag
  When a jsp wants to display response in a graphics format in a browser, then a jsp page will integrates with an applet. given for integrating a jsp page with an applet we use <jsp:plugin> tag. In applet class, we have a method called paint(). In this method we can create output in graphical format using methods of Graphics class.

  Example

```
<jsp:plugin type="applet" code="classsname" width="200" height="200">
<jsp:params>
<jsp:param ----------/>
<jsp:param------------/>
</jsp:params>
<jsp:fallback>Alternate Message</jsp:fallback>
</jsp:plugin>
```

- **Jsp param: <jsp:param>** tag is used to represent parameter value during jsp forward or include action this should be the sub tag of <jsp:forward> or <jsp:include>. When include or forward element is invoked, the original request object is provided to the target page. If you wish to provide additional data to that page, you can append parameters to the request object by using the jsp:param element.

  Syntax

```
<jsp:param name=" " value=" "/>
```

  Example

```
<jsp:include page="contact.jsp"/>
<jsp:param name="param1" value="value1"/>
</jsp:include>
```

  Example

```
<jsp:forward page="home.jsp"/>
<jsp:param name="param1" value="value1"/>
</jsp:forward>
```

- **Jsp fallback: <jsp:fallback>** tag is used to display alternative text when there is a problem to load the requested applet. This is sub element of plugin like as **alt** tag in html.

  When use <jsp:fallback> tag
  This tag is used to display alternative text when there is a problem to load the requested applet..

Example

```
<jsp:plugin type="applet" code="classsname" width="200" height="200">
<jsp:fallback>Unable to load Applet</jsp:fallback>
</jsp:plugin>
```

Difference between Include Directive and Include Action

	Include Directive	Include Action
1	In Include directive, the code of one jsp page is inserted into another jsp. It is called as compile time or static including.	Include Action, the response of one page will be inserted into another page. It is called as run-time or dynamic including.
2	In Include directive, for both source and destination pages, only a single Servlet will be generated internally so number of Servlet object are reduced.	In Include action, individually Servlet are generated for each jsp page. So it increases the number of servlet object.
3	In Include directive, the destination must be jsp page only.	In Include action, the destination resource can be a jsp or Servlet or html.
4	In jsp include directive, we have both html and xml syntax for the tag.	In include action, we have only xml syntax but there is one html syntax.
5	<%@include file=" "%>	<%JSP:include page=" "%>
6	In include directive, the file must be available in the within some application.	In include action the page may exits either within same application or another web application of server.

JSP Standard Tag Library (JSTL)

JSP Standard Tag Library(JSTL) is a standard library of custom tags. The JSTL contain several tag that can remove scriplet code from a JSP page. JSTL are divided into 5 categories.

1. **JSTL Core**: JSTL Core provide several core tags such as **if**, **forEach**, **import**, **out** etc to support some basic scripting task. Url to include JSTL Core Tag inside JSP page is:
   <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
2. **JSTL Formatting**: JSTL Formatting Tag provide tags to format text, date, number for Internationalized web sites. Url to include JSTL Formatting Tag inside JSP page is:
   <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
3. **JSTL sql**: JSTL SQL tag provide support for Relation Database Connection. Url to include JSTL SQL Tag inside JSP page is:
   <%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
4. **JSTL XML**: JSTL XML tag provide support for XML. It provide flow control, transformation etc. Url to include JSTL XML Tag inside JSP page is:
   <%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
5. **JSTL functions**: JSTL functions tag provide supports for string manipulation. Url to include JSTL Function Tag inside JSP page is:
   <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

1. JSTL Core: The JSTL core library contains several tags that are used to eliminate the basic scripting such as for loop, if condition etc from a JSP Page. Let's study some important tag of JSTL Core library**.**

- **JSTL if tag**: The if tag is a conditional tag used to evaluate conditional expression. When a body is supplied with if tag , the body is evaluated only when the expression is true.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>
<c:if test="${param.name == 'studytonight' }">
   <p>Welcome to ${param.name} </p>
</c:if>
</body>
</html>
```

- **JSTL out tag**: The out tag is used to evaluate a expression and write the result to **JspWriter**.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   <c:out value="${param.name}" default="StudyTonight" >

</body>
</html>
```

The value attribute specifies the expression to be written to the JspWriter. The default attribute specifies the value to be written if the expression evaluate null.

- **JSTL forEach tag**: The **forEach** tag provide a mechanism to iteration within a JSP page. JSTL forEach tag works similar to **enhanced for** loop of Java Technology. You can use this tag to iterate over a existing collection of item.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   <c:forEach var="message" items="${errorMsgs}" >
    <li>${message}</li>
    </c:forEach >

</body>
</html>
```

Here the attribute **items** has its value an EL expression which is a collection of error messages. Each item in the iteration will be stored in a variable called **message** which will available in the body of the **forEach**tag.

- **JSTL choose, when, otherwise tag**: These are conditional tags used to implement conditional

evaluated otherwise the content within the **otherwise tag** is evaluated. We can also implement **if-else-if construct** by using multiple **when tags**. These when tags are mutually exclusive that means the first when tag whose test evaluates to true is evaluated and then, the control exits the choose block. If none of the when condition evaluates to true , the otherwise condition is evaluated.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>
<c:forEach var="tutorial" items="${MyTutorialMap}" begin="0" end="5"
varStatus="status" >
<c:choose>
<c:when test="${status.count %2 == 0 }">
    <p style="color:red;"> ${tutorial.key} </p><br/>
</c:when>
<c:when test="${status.count %5 == 0 }">
    <p style="color:green;"> ${tutorial.key} </p><br/>
</c:when>
<c:otherwise>
    <p> ${tutorial.key} </p><br/>
</c:otherwise>
</c:choose>
</c:forEach>

</body>
</html>
```

- **JSTL import tag**: `< c:import>` tag is used to dynamically add the contents from the provided URL to the current page, at request time. The URL resource used in the `< c:import>` url attribute can be from outside the web Container.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   < c:import url=”http://www.example.com/hello.html” >
    < c:param name="showproducts" value="true"/ >
    </c:import>

</body>
</html>
```

- **JSTL url tag**: The JSTL url tag is used to store a url in a variable and also perform url rewriting when necessary.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   < a href="<c:url value="/home.jsp" />" > Go Home </a>

</body>
</html>
```

- **JSTL set tag**: The JSTL set tag is used to store a variable in specified scope or update the property of JavaBean instance.

Example of setting the name property of a Student bean

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   < c:set target= "student" property="name" value="${param.name}" / >

</body>
</html>
```

- **JSTL catch tag**: The JSTL catch tag is used to handle exception and doesn't forward the page to the error page.

Example:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>

   < c:catch  >
      <% int a =0;
         int b =10;
         int c= b/a;
      %>
     < / c:catch  >

</body>
</html>
```

2. **JSTL Formatting**: JSTL Formatting Tag provide tags to format text, date, number for Internationalized web sites. Url to include JSTL Formatting Tag inside JSP page is:

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

Following is the list of Formatting JSTL Tags:

Tag	Description
<fmt:formatNumber>	To render numerical value with specific precision or format.
<fmt:parseNumber>	Parses the string representation of a number, currency, or percentage.
<fmt:formatDate>	Formats a date and/or time using the supplied styles and pattern
<fmt:parseDate>	Parses the string representation of a date and/or time
<fmt:bundle>	Loads a resource bundle to be used by its tag body.
<fmt:setLocale>	Stores the given locale in the locale configuration variable.
<fmt:setBundle>	Loads a resource bundle and stores it in the named scoped variable or the bundle configuration variable.
<fmt:timeZone>	Specifies the time zone for any time formatting or parsing actions nested in its body.
<fmt:setTimeZone>	Stores the given time zone in the time zone configuration variable
<fmt:message>	To display an internationalized message.
<fmt:requestEncoding>	Sets the request character encoding

3. **JSTL sql**: JSTL SQL tag provide support for Relation Database Connection. Url to include JSTL SQL Tag inside JSP page is:
```
<%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>
```
Following is the list of SQL JSTL Tags:

Tag	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.
<sql:dateParam>	Sets a parameter in an SQL statement to the specified java.util.Date value.
<sql:transaction >	Provides nested database action elements with a shared Connection, set up to execute all statements as one transaction.

4. **JSTL XML**: JSTL XML tag provide support for XML. It provide flow control, transformation etc. Url to include JSTL XML Tag inside JSP page is:
```
<%@ 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:if >	Evaluates a test XPath expression and if it is true, it processes its body. If the test condition is false, the body is ignored.
<x:forEach>	To loop over nodes in an XML document.
<x:choose>	Simple conditional tag that establishes a context for mutually exclusive conditional operations, marked by <when> and <otherwise>
<x:when >	Subtag of <choose> that includes its body if its expression evalutes to 'true'
<x:otherwise >	Subtag of <choose> that follows <when> tags and runs only if all of the prior conditions evaluated to 'false'
<x:transform >	Applies an XSL transformation on a XML document
<x:param >	Use along with the transform tag to set a parameter in the XSLT stylesheet

5. **JSTL functions**: JSTL functions tag provide supports for string manipulation. Url to include JSTL Function Tag inside JSP page is:
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
Following is the list of JSTL Functions:

Function	Description
fn:contains()	Tests if an input string contains the specified substring.
fn:containsIgnoreCase()	Tests if an input string contains the specified substring in a case insensitive way.
fn:endsWith()	Tests if an input string ends with the specified suffix.
fn:escapeXml()	Escapes characters that could be interpreted as XML markup.
fn:indexOf()	Returns the index withing a string of the first occurrence of a specified substring.
fn:join()	Joins all elements of an array into a string.
fn:length()	Returns the number of items in a collection, or the number of characters in a string.
fn:replace()	Returns a string resulting from replacing in an input string all occurrences with a given string.
fn:split()	Splits a string into an array of substrings

fn:startsWith()	Tests if an input string starts with the specified prefix.
fn:substring()	Returns a subset of a string.
fn:substringAfter()	Returns a subset of a string following a specific substring.
fn:substringBefore()	Returns a subset of a string before a specific substring.
fn:toLowerCase()	Converts all of the characters of a string to lower case.
fn:toUpperCase()	Converts all of the characters of a string to upper case.
fn:trim()	Removes white spaces from both ends of a string.

JSP TAG EXTENSIONS

Introduction:
Tag extensions are an important feature of JSP. They are also referred to as custom tags. Tag extensions looks like standard HTML or XML tags embedded in a JSP page. But they have a special meaning to the JSP Engine at translation time. They allow custom functionality to be invoked without having to write java code within scriptlets. This allows a JSP developer to invoke application functionality without having to know any of the underlying java code.

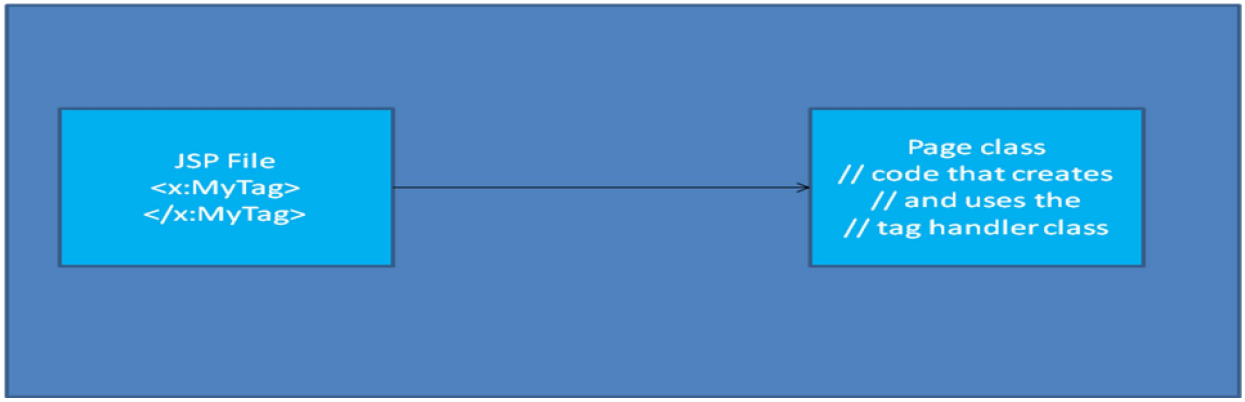
Tag libraries also offer portable runtime support, authoring / modification support and validation. Tag extensions allow a vast range of new functionality to be added to JSP pages which can be invoked just like standard actions.

The key concepts of tag extensions are:

- Tag name: A JSP tag is uniquely identified in a page by a combination of a prefix and a suffix which are separated by a colon <prefix:suffix>
  The prefix identifies a tag library and the suffix identifies a particular tag in the library.
- Attributes: Tags may have attributes. Attributes may be necessary or optional.
- Nesting: Tag extensions can detect nestings at runtime and cooperate. A tag directly enclosing another tag is the parent of the tag it encloses.
- Body content: The body content is anything between the start and end elements in a JSP tag excluding sub tags. A tag extension can access and manipulate its body contents
- Scripting variables: Tags can define variables that can be used within the JSP page in tag body content or after the tag has been closed.

USES OF TAG EXTENSIONS

- Conceal the complexity of access to a data source or enterprise object from the page author.
- Introduce a new scripting variable into the page.
- Filter or transform tag content
- Handle iteration without the need for scriptlets
- The actual behaviour of a custom action is provided at runtime by an instance of a java class
- This java class is also known as a **Tag Handler**



Tag Handlers

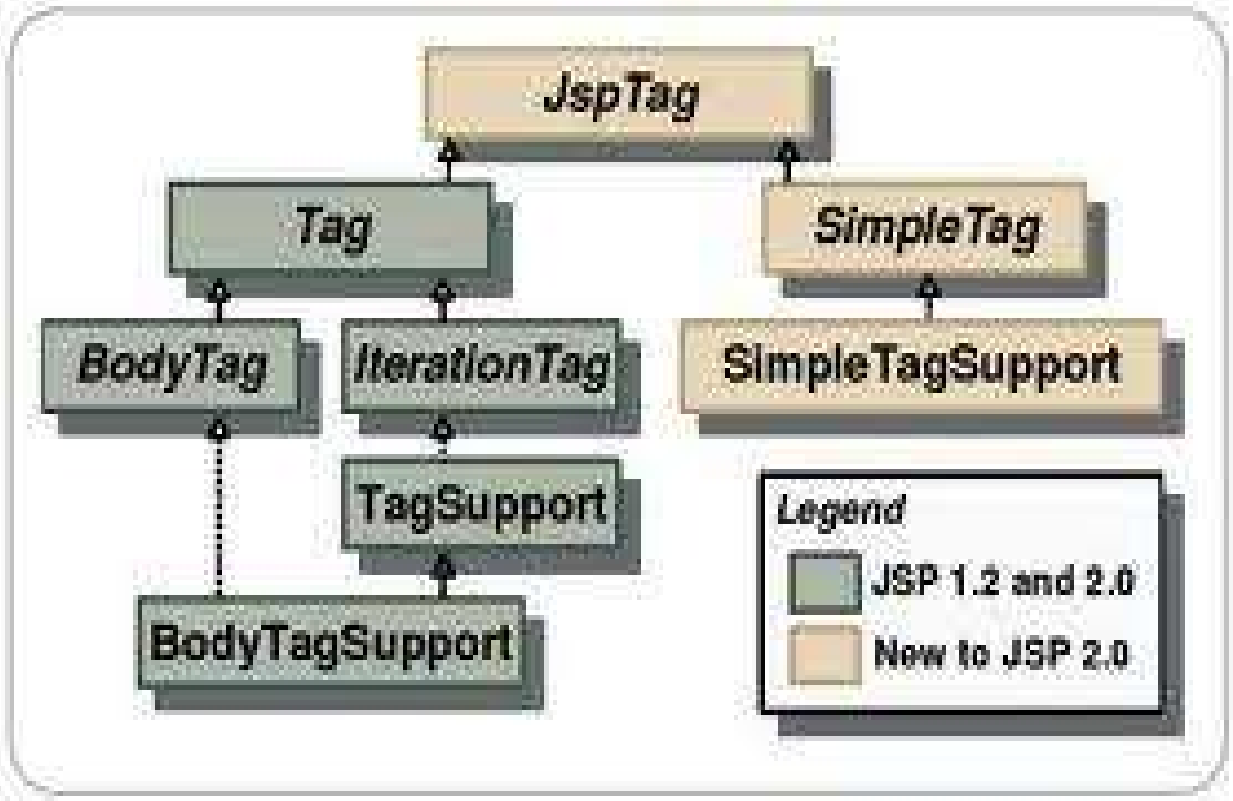
The tag handler is the java class that implements the behaviour of a custom action. It follows the requirements of a Java Bean and it implements one of the tag extension interfaces.

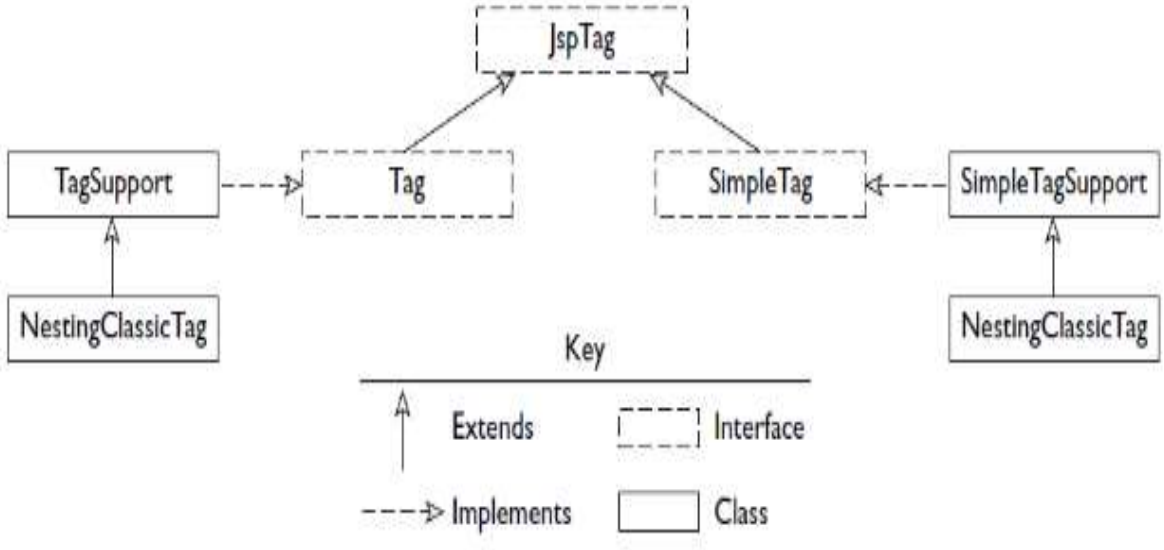
- **Tag:** This interface handles a simple action with no iteration and no need to process the body of the tag.
- **BodyTag:** This interface is used when the body of the tag is processed as part of the action.
- **IterationTag:** This interface is used to handle iterations
- **SimpleTag:** This interface is used to make tag handling easier
- **JSPFragment:** This tag interface is used to encpasulate the body content of a tag in an object.

Note:

- Tag, BodyTag and IterationTag are called Classic tag handlers
- SimpleTag and JSPFragment tags are called simple tag handlers.

TAG EXTENSION CLASS HIERARCHY





SimpleTag

JSP 2.0 introduces the *SimpleTag* interface and a base class *SimpleTagSupport* that implements this interface. To create a custom action, one would create a tag handler class that extends the *SimpleTagSupport* base class, overriding the methods as necessary to provide the behaviour of the custom action. Usually only the *doTag()* method is overridden.

Methods of the SimpleTagSupport

- doTag()
- setParent(parent:JspTag)
- getParent(): JspTag
- setJspContext(pc:JspContext)
- getJspContext(): JspContext
- setJspBody(jspBody:JspFragment)
- getJspBody():JspFragment

When the tag appears in a JSP file, the translator creates code that:

- Creates an instance of the tag handler
- Calls setJspContext() and setParent()
- Initializes the tag handler attributes
- Creates a JspFragment object and calls setJspBody()

Note:

- For the page implementation code to be able to create the tag handler instance and initialize its properties, all tag handlers follow the JavaBean conventions.
- After the tag handler class is created and initialized, the page calls the doTag(). This method is called only once for the tag.

JspFragment

JspFragment is also an interface, but its implementation is left entirely to the JSP container. If a SimpleTag tag handler needs to evaluate the body of the tag, it calls the invoke() method of the JspFragment.

Syntax:

```
public void invoke(java.io.Writer out, java.util. Map params)
```

Tag Library Descriptor

After creating one or more classes that implement a tag, you need to inform the container which tag handlers are available to the JSP pages in an application.This is done through a descriptor file called a Tag Library Descriptor (TLD). It is an XML-compliant document that contains information about the tag handler classes in a tag library.

A TLD will provide information about the tag library using a <taglib> element. The <taglib> element can have a number of sub elements. They are:

- <tlib-version>: The version number of the library
- <short-name>: A simple default name. It may be used as the preferred prefix value in taglib directives
- <tag>: Information about the tag handler

The <tag> element has several sub-elements. They are:

- <name>: The name of the tag handler
- <tag-class>: The fully qualified class name of the tag handler class
- <body-content>: Whether the body tag can have content. It can take values *tagdependent*, *JSP* or *empty*. *JSP* is default
- <variable>: Defining the scripting variables created by the tag handler
- <attribute>: Defines attributes for the tag. It has three sub elements: name, required, rtexprvalue

GENERAL STRUCTURE

```
<taglib>
<tlib-version>1.0</tlib-version>
<short-name>    </shortname>
<tag>
<name>…</name>
…
…
</tag>
</taglib>
```

PACKAGING THE TAG LIBRARIES

After creating the tag handler classes and the TLD, there are a few final steps that need to be accomplished to use the tags in a JSP.

- Application Structure:
  - Tag handler classes must be placed in the classes subdirectory of WEB-INF
  - TLD must be placed in the WEB-INF directory
- Deployment Descriptor:
  - Within the web.xml, a mapping from the URI to a TLD location should be created
  - This is done by adding a <taglib> element in web.xml

```
<taglib>
<taglib-uri>/examples</taglib-uri>
<taglib-location>/WEB-INF/tlds/descriptor.tld </taglib-location>
</taglib>
```

Importing the Taglib into a page:
To use a custom action, the tag library must be imported into the JSP. This is done using **taglib directive**.

```
<%@ taglib uri="URI_of_Library" prefix="tag-prefix"%>
```

JspTag interface
The JspTag is the root interface for all the interfaces and classes used in custom tag. It is a marker interface.

Tag interface: The Tag interface is the sub interface of JspTag interface. It provides methods to perform action at the start and end of the tag.

Fields of Tag interface: There are four fields defined in the Tag interface. They are:

Field Name	Description
public static int EVAL_BODY_INCLUDE	It evaluates the body content.
public static int EVAL_PAGE	It evaluates the JSP page content after the custom tag.
public static int SKIP_BODY	It skips the body content of the tag.
public static int SKIP_PAGE	It skips the JSP page content after the custom tag.

Methods of Tag interface: The methods of the Tag interface are as follows:

Method Name	Description
public void setPageContext(PageContext pc)	It sets the given pagecontext object.
public void setParent(Tag t)	It sets the parent of the tag handler.
public Tag getParent()	It returns the parent tag handler object.
public int doStartTag()throws JspException	It is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the start of the tag.
public int doEndTag()throws JspException	It is invoked by the JSP page implementation object. The JSP programmer should override this method and define the business logic to be performed at the end of the tag.
public void release()	It is invoked by the JSP page implementation object to release the state.

IterationTag interface: The IterationTag interface is the sub interface of the Tag interface. It provides an additional method to reevaluate the body.
After calling doStartTag() and after evaluating the body of the tag, the page class calls the doAfterBody(). This method allows the tag handler class to determine if the body should be evaluated again. It the doAfterBody() returns IterationTag.EVAL_BODY_AGAIN, the body is evaluated again else it returns Tag.SKIP_BODY. The page then calls doEndTag() and proceeds

Field of IterationTag interface: There is only one field defined in the IterationTag interface.

- **public static int EVAL_BODY_AGAIN** it reevaluates the body content.

Method of Tag interface: There is only one method defined in the IterationTag interface.

- **public int doAfterBody()throws JspException** it is invoked by the JSP page implementation object after the evaluation of the body. If this method returns EVAL_BODY_INCLUDE, body content will be reevaluated, if it returns SKIP_BODY, no more body cotent will be evaluated.

BodyTag interface: It is an Interface which has the following fields and Methods

- EVAL_BODY_TAG
- EVAL_BODY_BUFFERED
- setBodyContent(bc:BodyContent)
- doInitBody()

BodyTagSupport Class: It implements the BodyTag interface and extends the TagSupport class. It has the following methods.

- BodyTagSupport()
- getBodyContent(b:BodyContent)
- doInitBody()

In this scenario, the doStartTag() method works in the same way but it can return one more value which is BodyTag.EVAL_BODY_BUFFERED. If this value is returned, the page evaluates the tag body and stores the result in an instance of BodyContent. This instance is passed to the tag handler using the setBodyContent() method. The page then calls the doInitBody() using which the tag handler class can perform any initialization that depends on the body of the tag.The doEndTag() is called after this.

TagSupport class: TagSupport class implements the IterationTag interface**.** Its additional methods are:

- TagSupport()
- doStartTag()
- doEndTag()
- setId(id:String)
- getId()
- setValue(k:String, o:Object)
- getValue(k:String)
- removeValue(k:String)
- getValues()

When we extend the TagSupport class, you only need to override doStartTag() or doEndTag(). The doStartTag() is called by the page class at the point where the start tag appears in the JSP file.This method contains code that must be evaluated before the body of the tag is processed.If it returns Tag.SKIP_BODY, the body is not evaluated. If it returns Tag.EVAL_BODY_INCLUDE, the body tag is evaluated

The doEndTag() method is called by the page class at the point where the end of the tag appears. It contains code that must be executed after the body of the tag is evaluated. If it returns Tag.SKIP_PAGE, the remainder of the JSP page is not evaluated. It is returns Tag.EVAL_PAGE the remainder of the JSP page is evaluated.

Creating a Custom Tag

To create a Custom Tag the following component are required

1. The **Tag Handler** class
2. The **Tag Library Descriptor(TLD)** file
3. Use the Custom Tag in the jsp file

Tag Handler Class:

We can create a Tag Handler class in two different ways.

1. By implementing one of three interfaces **SimpleTag**, **Tag**, or **BodyTag** which define methods that are invoked during the life cycle of the tag
2. By extending an abstract base class that implements the **SimpleTag**, **Tag**, or **BodyTag** interfaces. The**SimpleTagSupport**, **TagSupport**, and **BodyTagSupport** classes implement the **SimpleTag**, **Tag**,

or**BodyTag** interfaces . Extending these classes relieves the tag handler class from having to implement all methods in the interfaces and also provides other convenient functionality.

Tag Library Descriptor:
A tag library descriptor is an XML document that contains information about a library as a whole and about each tag contained in the library. TLDs are used by a web container to validate the tags and by JSP page development tools.

Tag library descriptor file names must have the extension `.tld` and must be packaged in the **/WEB-INF/**directory or subdirectory of the WAR file or in the **/META-INF/** directory or subdirectory of a tag library packaged in a JAR.

EXAMPLE
In our example, we are creating a tag handler class that extends the TagSupport class. When we extend this class, we have to override the method `int doStartTag()`. There are two other methods of this class namely `int doEndTag()` and `void release()` that we can decide to override or not depending on the requirement.

CountMatches.java

```
package com.studytonight.taghandler;

import java.io.IOException;
import javax.servlet.jsp.*;
import org.apache.commons.lang.StringUtils;

public class CountMatches extends TagSupport {

            private String inputstring;
             private String lookupstring;

           public String getInputstring() {
                   return inputstring;
             }

             public void setInputstring(String inputstring) {
                   this.inputstring = inputstring;
             }

             public String getLookupstring() {
                   return lookupstring;
             }

             public void setLookupstring(String lookupstring) {
                   this.lookupstring = lookupstring;
             }

             @Override
           public int doStartTag() throws JspException {

               try {
               JspWriter out = pageContext.getOut();
       out.println(StringUtils.countMatches(inputstring, lookupstring));
```

```
                e.printStackTrace();
            }
            return SKIP_BODY;
        }
}
```

In the above code, we have an implementation of the doStartTag() method which is must if we are extending **TagSupport** class. We have declared two variables **inputstring** and **lookupstring**. These variables represents the attributes of the custom tag. We must provide getter and setter for these variables in order to set the values into these variables that will be provided at the time of using this custom tag. We can also specify whether these attributes are required.

CountMatchesDescriptor.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib>
<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>cntmtchs</shortname>
<info>Sample taglib for Substr operation</info>
<uri>http://studytonight.com/jsp/taglib/countmatches</uri>

<tag>

    <name>countmatches</name>
    <tagclass>com.studytonight.taghandler.CountMatches</tagclass>
    <info>String Utility</info>
    <attribute>
         <name>inputstring</name>
         <required>true</required>
     </attribute>
    <attribute>
         <name>lookupstring</name>
         <required>true</required>
      </attribute>
</tag>

</taglib>
```

The taglib element specifies the schema ,required JSP version and the tags within this tag library. Each tag element within the TLD represents an individual custom tag that exist in the library. Each of these tag should have a tag handler class associated with them.

The **"uri"** element represents a Uniform Resource Identifier that uniquely identifies the tag library. The two attribute elements within the tag element represents that the tag has two attributes and the **true** value provided to the required element represents that both of these attributes are required for the tag to function properly.

test.jsp

```
<%@taglib prefix="mytag" uri="/WEB-INF/CountMatchesDescriptor.tld"%>
<html>
<mytag:countmatches inputstring="StudyTonight" lookupstring="t">

</mytag:countmatches>
</html>
```

Custom tags are user-defined tags. They eliminates the possibility of scriptlet tag and separates the business logic from the JSP page. The same business logic can be used many times by the use of custom tag.

Advantages of Custom Tags

The key advantages of Custom tags are as follows:

1. **Eliminates the need of scriptlet tag** The custom tags eliminates the need of scriptlet tag which is considered bad programming approach in JSP.
2. **Separation of business logic from JSP** The custom tags separate the business logic from the JSP page so that it may be easy to maintain.
3. **Re-usability** The custom tags makes the possibility to reuse the same business logic again and again.

Syntax to use custom tag

There are two ways to use the custom tag. They are given below:

```
<prefix:tagname attr1=value1....attrn=valuen />
<prefix:tagname attr1=value1....attrn=valuen >
body code
</prefix:tagname>
```