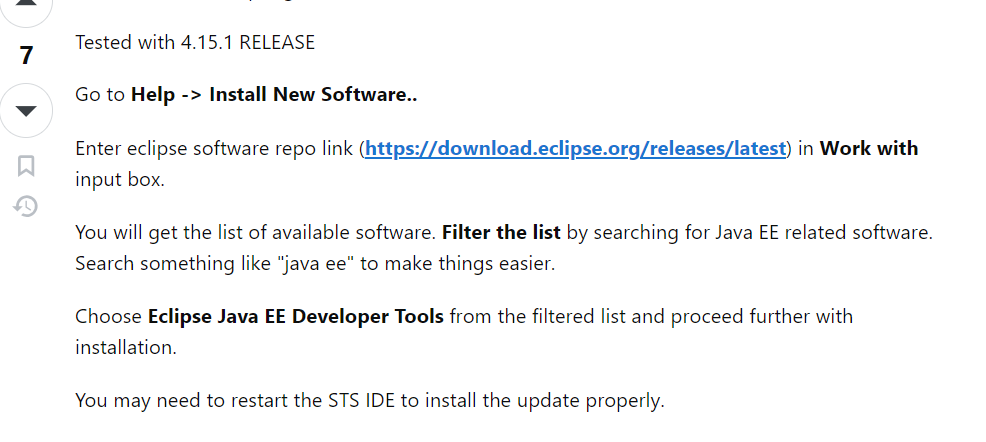
**Servlets**

https://stackoverflow.com/questions/56090961/dynamic-web-project-missing-in-sts



**Web application:**

The application which is developed only by using web related technologies (like html, xml, java script, angular, servlets, jsps etc) is called web application.

Eg: Amazon.com

Web application provides services over the web to the end users.

**Web server:**

Once we developed web application, we required special environment to run that web application which is called web server. Web server provides environment to run web applications.

Eg: Tomcat, Nginx,

**Deployment:**

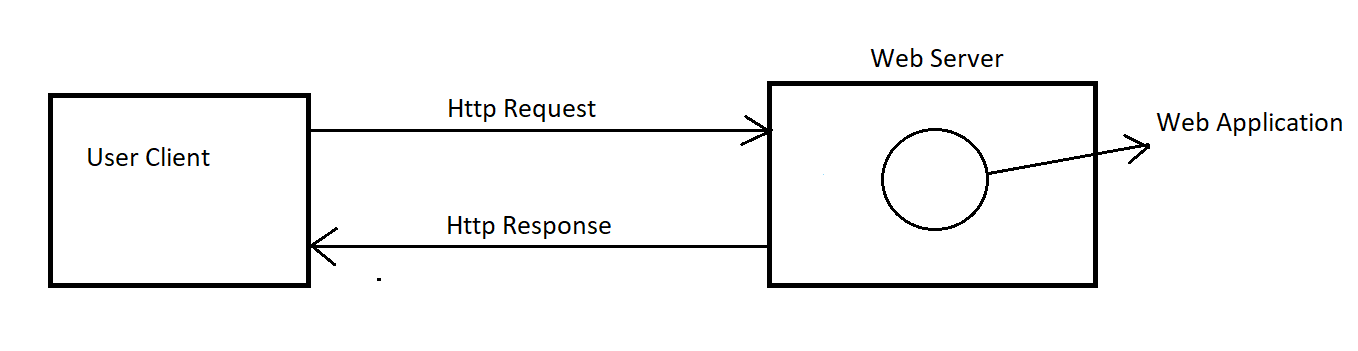
The process of placing web application inside web server is called deployment.

The process of removing web application from the web server is called un-deployment.

**Web client (User Agent):**

To send a request to web server we need a user interface to interact with web server which is called web client. i.e By using web client we can communicate with web server.

Eg: Chrome, Postman (very IMP) and CURL (in Linux env).



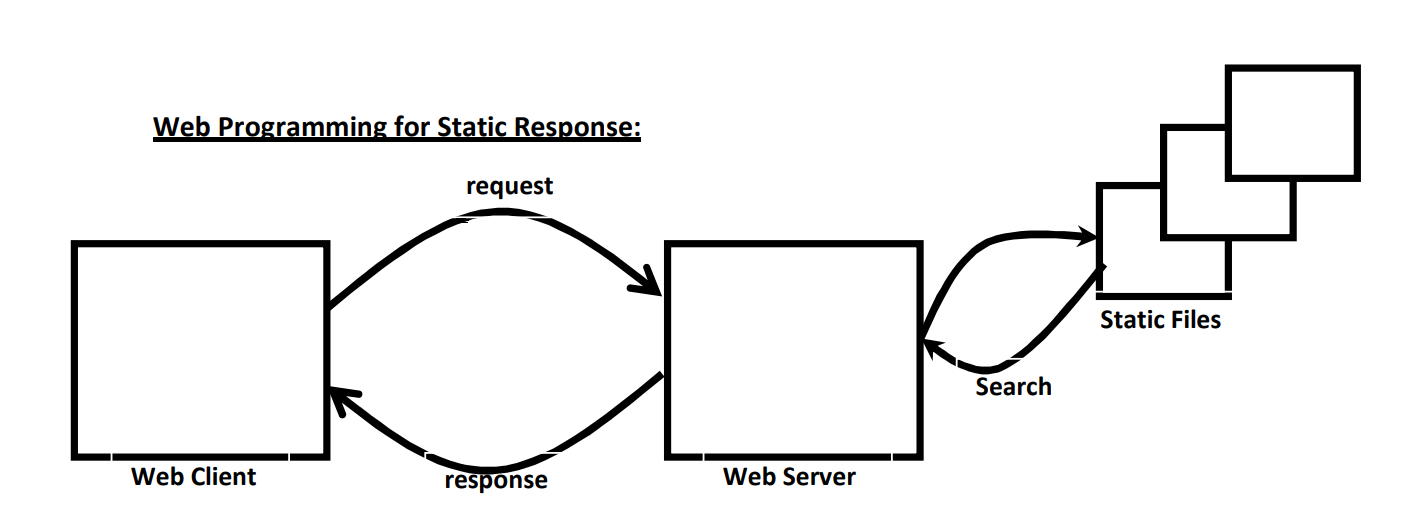
**Static Response vs Dynamic Response:**

The response which won't be changed from person to person at any given time, such type of response is called static response.

Eg: Home pages.

The response which is varied from person to person according to user input, such type of response is called dynamic response.

Eg: Gmail inbox.

****

**Steps:**

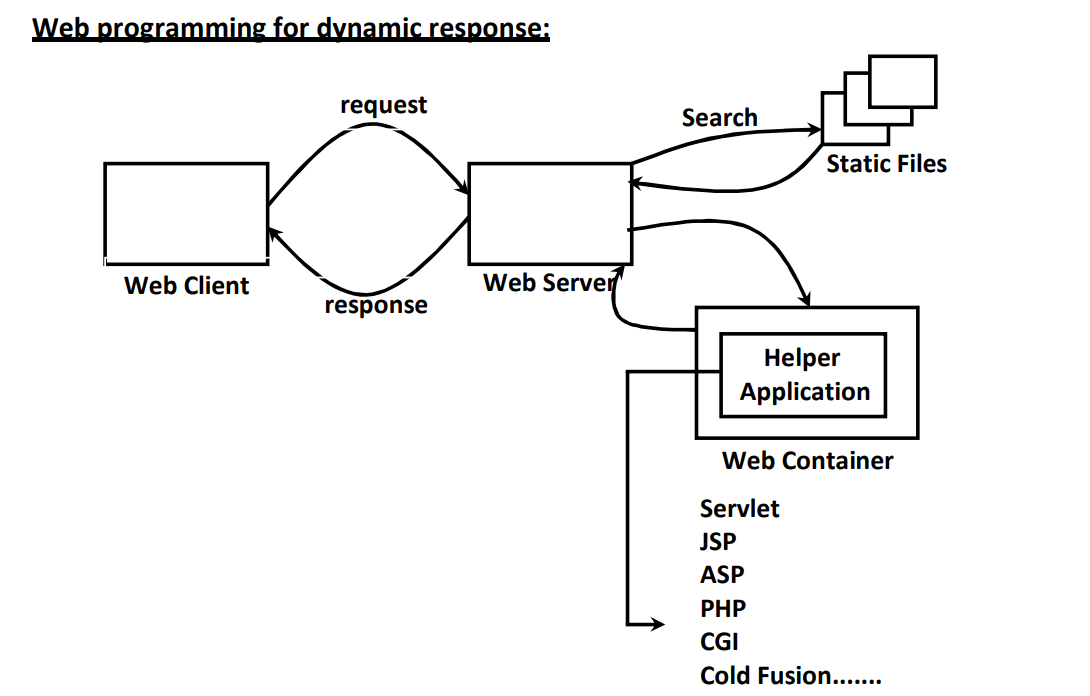
1. Client sends a request for static file to the server.

2. Server searches whether requested resource is available or not

3. If the requested resource is available then server will provide that file as response.

4.If the requested resource is not available then we will get 404 status code saying requested resource is not available.

Note: To serve static files, no processing is required at server side.



**Steps:**

For dynamic request, web server will forward the request to some helper application.

Helper application will analyze request, process request and generates required dynamic response.

Helper application forwards that response to web server and web server forwards that response to the client.

**THAT HELPER APPLICATION IS SERVLET/JSP.**

**Servlet:**

Servlet is a server-side web component which is responsible to generate dynamic responses.

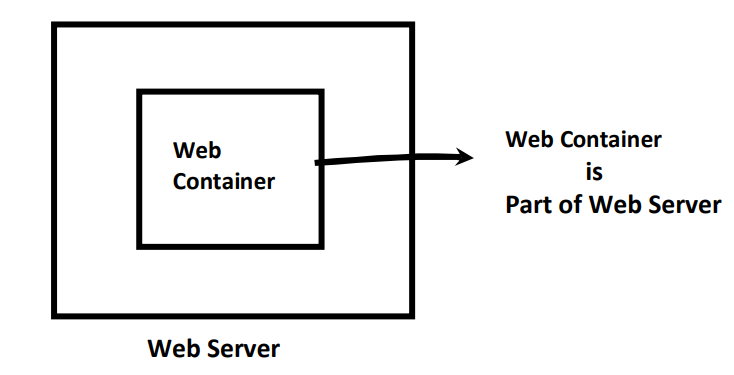
Total Servlet life cycle is managed by web container.

Web container manages the complete life cycle of servlet.

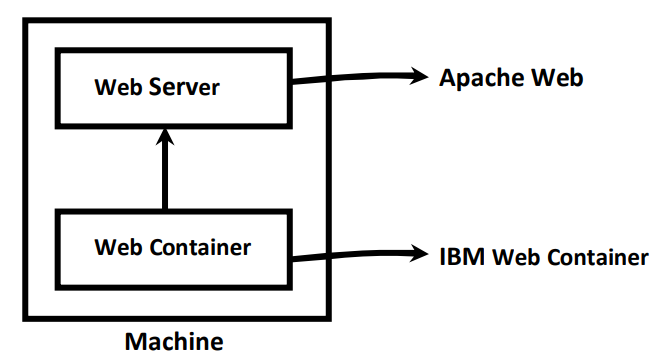
**Types of web containers:**

1. Standalone web containers.
2. In process web containers.
3. Out process web containers.

**Standalone web containers:** Both web server and web container are available as a single integrated component, such type of web containers are called standalone web containers.



**In-process web container:**



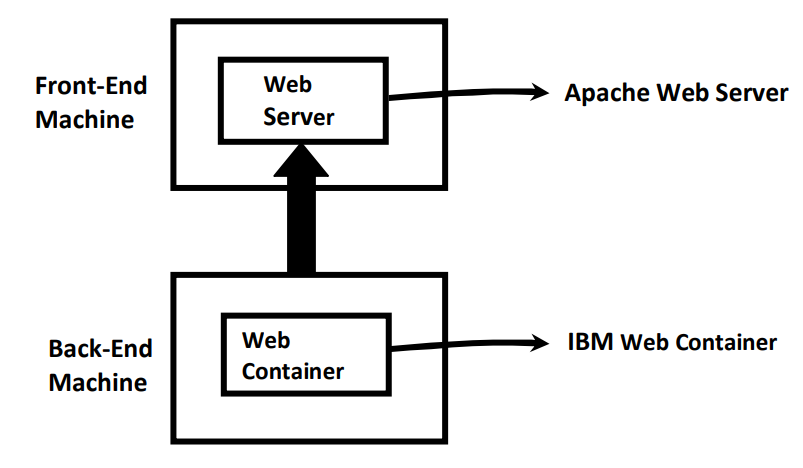
web container is connected to the web server as plug-in and both web server and web container are running in the same address space(same machine),such type of web containers are called in process web containers.

The benefit of this kind of web container is both web server and web container need not be from the same vendor. We can use Apache web server and WebLogic web container.

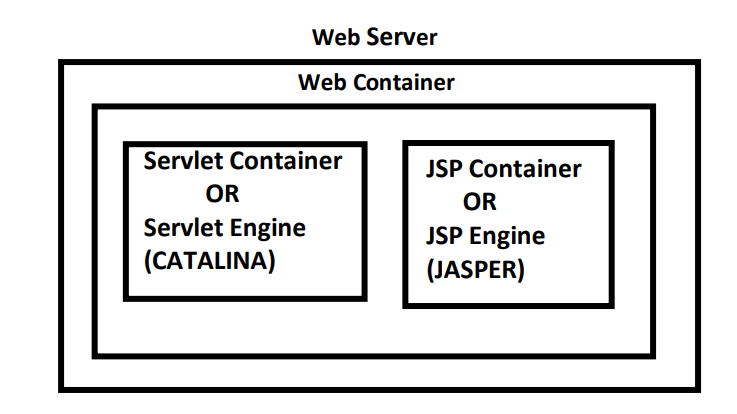
Best suitable of small applications.

**Out process web container:** Web server and web container are running on different machines. Web container is attached to the web server externally. We can configure Front end Apache web server to forward request to the back end IBM web container.

Best suitable for **real time applications.**



**Tomcat components:** web container is responsible to manage and execute servlets and jsps.



Internally web container contains two components.

1. Servlet Container (Servlet Engine):

It is responsible to manage and execute Servlet components.

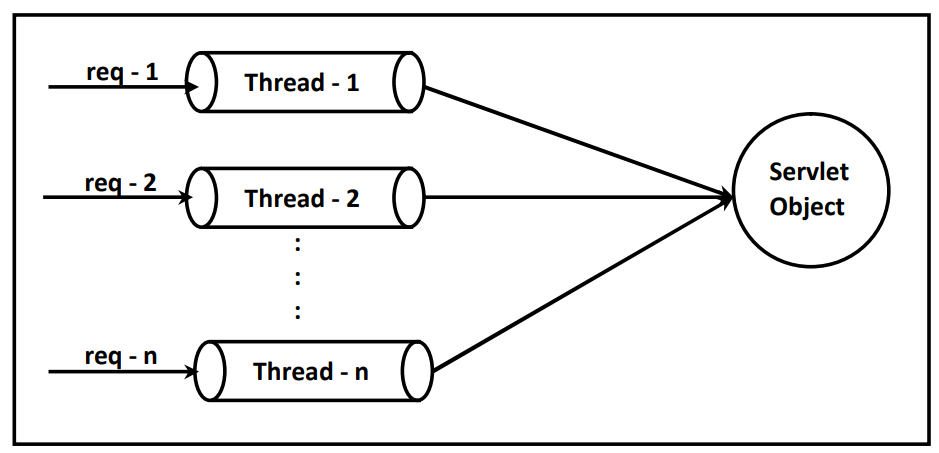
In Tomcat servlet container's name is CATALINA.

1. JSP container (JSP Engine):

It is responsible to manage and execute JSP Components.

In Tomcat JSP Container's name is JASPER.

**How Servlet Technology works:**

1. For every Servlet, web container will create **only one object.**
2. For every request, web container will allocate separate thread, which is responsible to process that request. Hence servlet follows "**Single Instance Multi-Threaded Model**". 

**Servlet API:**

Servlet API defines the following 2 packages.

* + Javax.servlet
  + Javax.servlet.http

**Javax.servlet:** This package defines several **classes** and **interfaces** used for developing protocol independent servlets (Generic Servlets).

**Javax.servlet.http:** This package defines several **classes** and **interfaces** which can be used to develop Http protocol based servlets.

**Important *interfaces* of javax.servlet package:**

1. Servlet
2. ServletRequest
3. ServletResponse
4. ServletConfig
5. ServletContext
6. RequestDispatcher
7. SingleThreadModel

**Servlet (I):**

* Every Servlet in Java should implement Servlet Interface either directly OR indirectly.
* This Interface defines the most common Methods which are applicable for any Servlet Object.
* The Life Cycle Methods of Servlet are defined in this Interface.

public class FirstServlet implements Servlet { }

**ServletRequest(I):**

* For **every request** web container creates one request object.
* **ServletRequest** object holds end user provided information.
* Servlet can use this request object to get end user's provided information.

**ServletResponse(I):**

* For **every request** web container creates one response object.
* Servlet can use response object to prepare and send response to end user.

**ServletConfig(I):**

* For **every Servlet**, web container creates a separate config object to hold its configuration information.
* In our programming Servlet can use this config object to get its configuration information.

**ServletContext(I):**

* For **every web application,** web container creates a separate **only one** context object to hold application-level configuration information.
* Servlet can use this context object to get application-level configuration information.

**NOTE**: ServletConfig is per Servlet whereas ServletContext is per web application.

**RequestDispatcher(I):**

We can use RequestDispatcher to dispatch request from one servlet to another servlet.

**SingleThreadModel(I):**

Single servlet object can be accessed by multiple threads simultaneously, hence there may be a chance of data inconsistency problems. **Which means servlet is not thread safe.**

SingleTheadModel solves this problem.

If our servlet class implements SingleThreadModel then our servlet object can be accessed by only one thread at a time.

public class FirstServlet implements Servlet,SingleThreadModel { }

**Benefit**: The main advantage of SingleThreadModel is threads will be executed one by one and hence data inconsistency problems will be resolved.

**Disadvantage:** The main disadvantage of SingleThreadModel is, it increases waiting time of Threads and creates performance problems. Hence it is not recommended to use SingleThreadModel.

Instead of SingleThreadModel it is recommended to use synchronized keyword at required places.

**Important *classes* of javax.servlet package:**

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletException

**GenericServlet:**

* GenericServlet implements Servlet interface.
* GenericServlet acts as base class to develop protocol **independent** servlets.

public class FirstServlet extends GenericServlet { }

**ServletInputStream:**

We can use ServletInputStream to read binary data send by end user.

**ServletOutputStream:**

We can use ServletOutputStream to write binary data to the response.

**ServletException:**

while processing user request if servlet faces any problem then it throws ServletException.

**Servlet(I):**

Every servlet in java should implements Servlet interface either directly or indirectly.

Servlet interface defines the most common methods which are applicable for any servlet.

Servlet interface defines the following 5 methods.

1. init()
2. service()
3. destroy()
4. getServletConfig()
5. getServletInfo()

**init() method:**

**syntax:** public void init(ServletConfig config) throws ServletException

* This method will be **called automatically** by web container to perform **initialization** activities after servlet **instantiation** immediately.
* Once init() method completes then only servlet is in a position to provide service.

**service() method:**

**syntax:** public void service(ServletRequest req,ServletResponse resp)throws ServletException,IOException

* This method will be **executed automatically** by web container for every request to provide required response.
* **Complete Service logic should be written in this method only.**

**destroy() method:**

**syntax:** public void destroy()

* This method will be executed only once by the web container to perform cleanup activities just before taking servlet object from out of service.
* Once destroy() method completes automatically web container destroys that servlet object.
* This is usually happens at the time of server shutdown or at the time of application undeployment.

**Note:** init(),service() and destroy() methods are called life cycle methods of servlet.

**getServletConfig():**

**syntax:** public ServletConfig getServletConfig()

This method returns ServletConfig object. By using this object servlet can get its configuration information.

**getServletInfo():**

**syntax:** public String getServletInfo()

This method returns information about our servlet like author, version, copyright information etc.

**Note:** getServletConfig() and getServletInfo() methods should be called explicitly by the programmer based on our requirement.

Execute First Servlet program.

**For the First Request:**

1. servlet loading
2. Servlet Instantiation (Object creation done servlet engine)
3. Init Method called
4. Service Method called

**From the Second Request onwards:**

Only Service Method will be called.

**Note:** For the time first request servlet class will be loaded and servlet object will be created followed by init() method execution. Finally service() method will be called. But for second request onwards only service() method will be called. Because of this the processing time of first request is more when compared with other requests.

To overcome this problem we should configure <load-on-startup>. Once we configured then servlet class loading, servlet instantiation and execution of init() method will be performed **at the time of server start up** or at the time of application deployment. For the first request also only service() method will be called.

We can configure in web.xml as below:

<web-app>

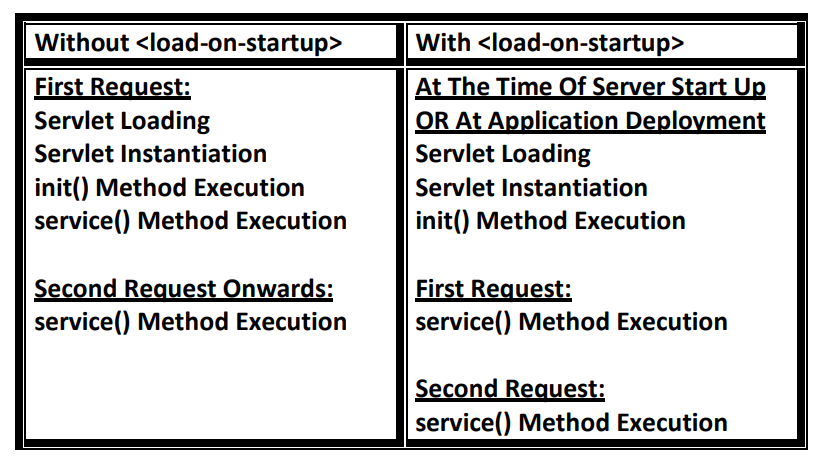
<servlet>

…………..

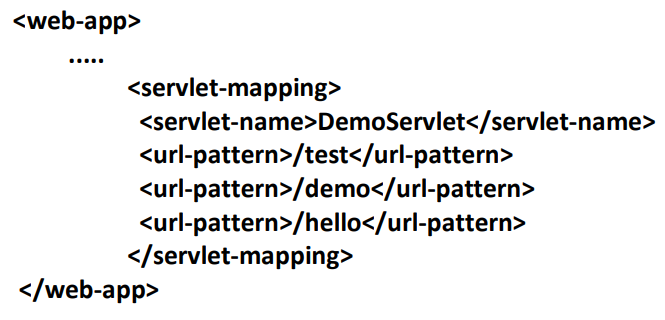
<load-on-startup>10</load-on-startup>

</servlet>

</web-app>



**Note:** we can define multiple url-patterns for the same servlet in web.xml as follows.



**Life Cycle of Servlet that implements Servlet interface:**

1. Client(Browser) sends a request to the server.
2. Server will check whether the request is for static or for dynamic information based on URL.
3. If the request is for static information, then the server will search for the required static file.If the static file is available then web server provides that static file as response. If the required static file is not available, then server will send 404 status code saying requested resource is not available.
4. If the request is for dynamic information, then web server forwards the request to the web container. 5. Web container will identify the corresponding servlet class based on url pattern and with the help of web.xml
5. web container will check whether servlet object is available or not If it is not already available, then web container will load servlet class,and create object for that servlet class and execute init() method. servlet class loading(by Class.forName() method) Servlet Instantiation( by uisng newInstance() method)
6. web container will call service() method which is responsible to provide required response.
7. Finally web container will call destroy() method to perform cleanup activities. This is usually happened at the time of application undeployment or at the time of server shutdown.

**Note:**

Whenever we are using annotations, compulsory we should import the below package.

import javax.servlet.annotation.\*;

we can replace web.xml configurations with annotations. Sometimes we can remove web.xml file also.

We can define url-pattern by using annotation as follows:

@WebServlet("/test")

We can define multiple url patterns as follows:

@WebServlet({"/test","/demo","/hello"})

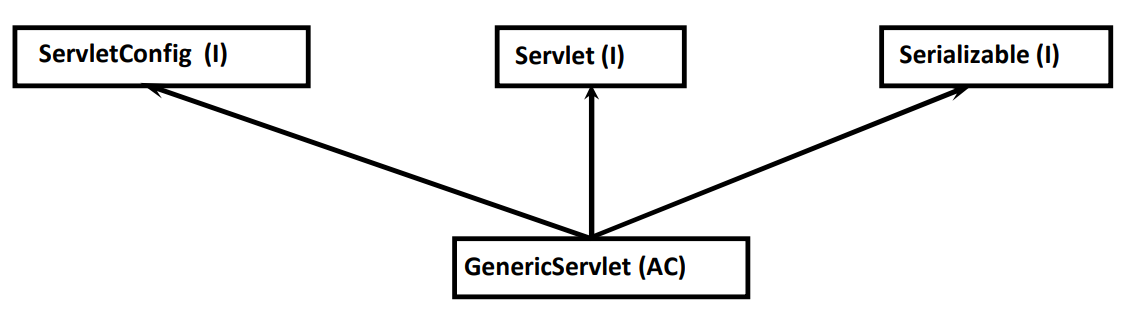
Execute FirstAnnotationServlet program.

**GenericServlet:**

We can develop a Servlet by implementing Servlet interface directly, but we should provide implementations for all 5 methods of Servlet interface whether required or not. Because of this length of the code will be increases and reduces readability.

GenericServlet provides solution to this, It already implemented Servlet interface and provides implementation for all methods of Servlet interface **except service()** method.

Instead of implementing Servlet interface if we extend GenericServlet then we required to provide implementation only for service() method, so that length of the code will be reduced and readability will be improved.



GenericServlet is an **abstract** class and implements Servlet, ServletConfig and Serializable interfaces.

Execute GenericServletTest example.

**Q1.** what happens If our servlet class does not contain any init() method?

**Q2.** what happens if we override init(SC) in our servlet class ?

**Q3.** what happens If we override no-arg init() method ?

Q4. why GenericServlet class contains 2 init() methods?

Q5. Why config variable declared as transient in GenericServlet ?

**javax.servlet.http:**

This package contains classes and interfaces which can be used for developing **Http** based servlets.

**Important interface of javax.servlet.http package**:

1. HttpServletRequest :

It is the child interface of ServletRequest.

We can use request object to get end user provided information.

1. HttpServletResponse:

It is the child interface of ServletResponse.

We can use response object to prepare and send response to the end user.

1. HttpSession :

We can use HttpSession object to implement session management.

**Important classes of javax.servlet.http package:**

1. HttpServlet:

It is the child class of GenericServlet.

HttpServlet acts as **base class** to develop Http based servlets.

1. Cookie:

We can use Cookies in the session management.

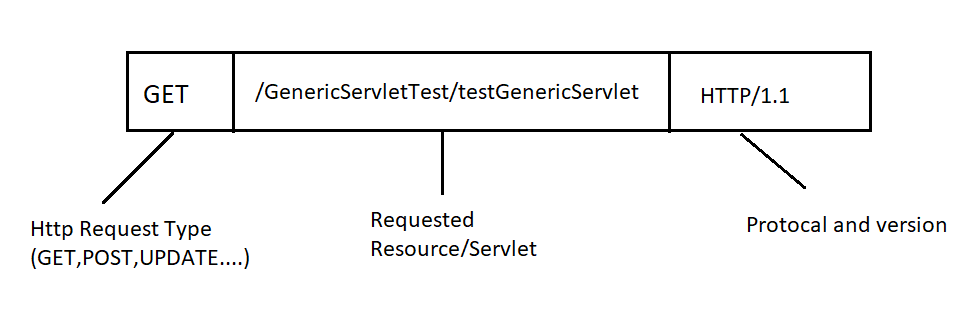
**Note:** web client and web server communicate with each other over the network HTTP protocols.

HTTP defines standard structures for request and response.

**Structure of Http request:**

* Request Line
* Request Headers
* Request Body

**Request Line:**



**Request Headers:**

Request headers will provide configuration information **of the browser** like media types accepted by browser, languages accepted by browser, encoding types supported by browser etc...

Web server will use these request headers while preparing proper response.

Ex: we will get user authentication details in encrypted form.

**Request Body: (TOO IMP)**

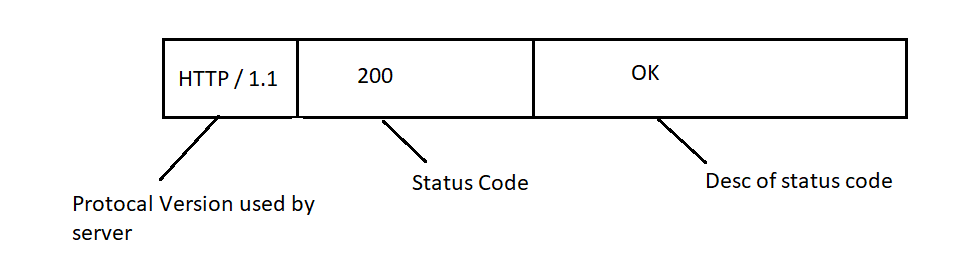
It contains end user provided information basically kind of object.

For the GET request, request body is optional and for POST request, request body is mandatory.

**Structure of Http Response:**

* Status Line
* Response Headers
* Response Body

**Status Line:**

****

**Status Code and Desc:**

1XX ==>Informational

2XX ===>Successful

3XX ===>Redirection

4XX ===>Client Error

5XX ===>Server Error

**Response Headers:**

These provide configuration information of server and information about our response like content type, content length, last modified date etc…

Browser will use these response headers to display response properly to the end user.

**Response Body:**

It contains original response provided by server. This response body will be displayed to the end user.

**Note:** Request headers send by browser and used by server whereas Response headers send by server and used by browser.

**Types of HTTP Request Methods:**

Based on operations performed by client, http methods are divided as follows:

1. **GET**: Requests using GET should only retrieve data.
2. **POST**: It submits data to the specified resource, which change in state of the server.
3. **PUT**: For update requests.
4. **DELETE**: delete specified resource.
5. **HEAD**: Identical to a GET request, but without the response body.
6. **OPTIONS**: describes the communication options for the target resource.
7. **TRACE**: performs a message loop-back test along the path to the target resource.
8. **PATCH**: The PATCH method applies partial modifications to a resource.
9. **CONNECT**: It establishes a tunnel to the server identified by the target resource.

In most the real time web applications we work with only GET, POST and PUT (sometimes).

**GET Method:**

1. To get information from the server then we can use GET Request.
2. GET Requests are Read Only and at server side no update operation performed.
3. In GET Request end user's provided information will be **appended** to the URL as the part of **Query String.**

**Eg :** <http://localhost:8080/FirstServlet/test?empName=abc>&empNo=123

1. In the GET request end user's provided information is visible to the outside hence we shouldn’t send sensitive information (like username, pwd etc) to server by using GET Request.
2. We can send only text data but we cannot send binary data(like images, video files etc.…) by using GET Request.
3. Caching of GET Request is possible and hence performance will be improved.

**Ex: GET** repuest to fetch state names of a country.

**Q. What is Idempotent request?**

By repeating the request multiple times if there is no change in the response, such type of requests are called Idempotent Requests.

Ex: GET is idempotent whereas POST is not idempotent.

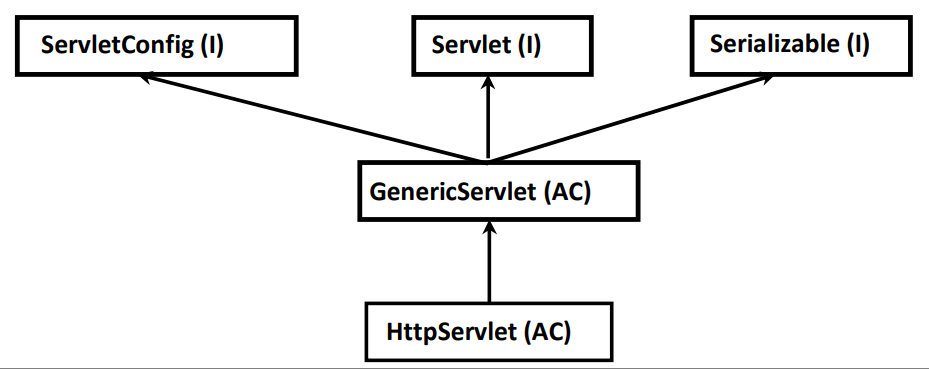
**POST Method:**

1. We can use POST method to send huge amount of information to the server Eg: uploading a file.
2. usually POST requests are WRITE only. i.e., at server-side update operation will be performed.
3. For the POST requests, end user provided information will be encapsulated in the request body instead of appending to the URL.
4. End user’s data is not visible in the URL and hence we can send sensitive information by using POST request.
5. There is no limit to the size of request body and hence we can send huge amount of information by using POST request.
6. By using POST request, we can send both text and binary data.
7. Caching is not applicable for POST requests.
8. Post requests are not idempotent.

Note: For GET requests body is optional whereas for POST **Requests body** is mandatory.

**HttpServlet:**

* HttpServlet acts as base class to develop Http Based servlets.
* It is the child class of GenericServlet.



For every Http Method XXX, HttpServlet class contains the corresponding doXxx() method.

**Syntax:**

Protected void doXxx(HttpServletRequest req,HttpServletResponse resp)throws ServletException,IOException.

Eg: protected void doGet(HttpServletRequest req,HttpServletResponse resp)throws ServletException,IOException.

**Note:**

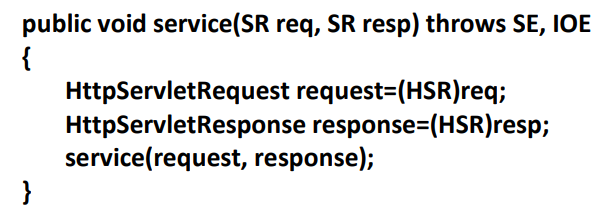
HttpServlet contains two Service methods:

1. public void service(ServletRequest req,ServletResponse resp)
2. protected void service(HttpServletRequest req,HttpServletResponse resp)

Execute http servlet programs.

**Life cycle of HttpServlet:**

1. Whenever we submit form, browser prepares HttpRequest and send to the server.
2. Web server checks whether request is for static or for dynamic information.
3. If the request is for dynamic information, then web server forwards the request to web container.
4. web container will identify corresponding servlet class based on URL pattern and with the help of web.xml.
5. web container will check whether servlet object is available or not.
6. If the servlet object is not available then web container loads servlet class, instantiate servlet and execute init() method.
7. web container creates ServletRequest and ServletResponse objects and invokes public service() method by passing these as arguments.
8. web container will check whether our servlet class contains public service(SR,SR) method or not. If our servlet class contains public service(SR,SR) method then it will be executed and provides required response.
9. If our servlet class does not contain public service() method then parent class(HttpServlet) public service(SR,SR) method will be executed.
10. If our servlet class does not contain public service() method then parent class(HttpServlet) public service(SR,SR) method will be executed.



1. Inside HttpServlet public service () method, req and resp objects will be type casted to HttpServletRequest and HttpServletResponse and then invoke protected service(HSR,HSR) method.
2. web container will check whether our servlet class contains protected service(HSR,HSR) method or not.
3. If our servlet class contains protected service(HSR,HSR) method then it will be executed and provide required response to the end user.
4. If our servlet class does not contain protected service() method then web container will execute HttpServlet protected service() method.
5. HttpServlet protected service() method will identify request method(like get,post etc) and invoke corresponding doXxx() method.
6. web container will check whether our servlet class contains the corresponding doXxx() method or not. If our servlet class contains doXxx() method then it will be executed and provide required response.
7. If our servlet class does not contain doXxx() method then parent class HttpServlet doXxx() method will be executed.

**Conclusion:**

Web Container will always calls the methods in the following order.

* public service(SR,SR)
* protected service(HSR,HSR)
* public doXxx(HSR,HSR)

**Try to answer these questions:**

1. Why GenericServlet class is declared as abstract?

2. HttpServlet class does not contain any asbtract method, still it is declared as abstract.What is the reason?

3. Internally how web container will create servlet object?

4. Why every servlet should contains public no-arg constructor?

5. The main purpose of constructor is to perform initialization and every servlet compulsory contains public no-arg constructor. Then what is need of init() method?

6. Is it possible to call destroy() method explicitly from init() and service() methods?

7. In How many ways we can develop servlet? 8. In Http based servlets , is it recommended to override service() method?

**HttpServletRequest:**

By using HttpServletRequest, we can do below things:

* Retrieving form parameters
* Retrieving Request headers
* Retrieving Cookies
* Retrieving Client and Server information

**Retrieving Form Parameters:**

* Form Parameters are key-value pairs where both key and value are String objects.
* Form parameter can be associated with single value or multiple values.

ServletRequest interface defines the following methods to retrieve form parameters from the request object.

1. **public String getParameter(String pname):**
2. returns the value associated with specified parameter.
3. If parameter associated with multiple values, then it returns only first value.
4. If the specified parameter is not available, then this method returns null.
5. Argument is case sensitive.

**Syntax:** String user=req.getParameter("uname");

1. **public String[] getParameterValues(String pname):**
2. It returns all values associated with the specified parameter.
3. If the specified parameter is not available, then this method returns null.
4. Argument is case sensitive.

**Syntax:** : String[] s = req.getParameterValues("course");

1. **public Enumeration getParmeterNames():**

returns all form parameter names.

1. **public Map getParameterMap():**
   1. returns Map object which contains all parameter names and values.

**Retrieving Request Headers:**

For every request browser sends its configuration information in the form of headers.

These may include the media types, encoding types, accepted languages, the type of browser etc...

Server can use these request headers to send proper response to the browser.

Ex:

1. accept: media types accepted by browser (like html,pdf,images etc..)
2. accept-encoding: encoding types accepted by browser like gzip,dflate etc..
3. accept-language: language types accepted by browser like un-en.
4. user-agent: It represents the type of browser.
5. cookie: used to send cookies for session management.

String encoding=req.getHeader("accept-encoding");

String userAgent=req.getHeader("user-agent");

**HttpServletRequest defines the following methods to retrieve *header information* at server side:**

1. **public String getHeader(String hname)**
2. returns value associated with specified request header.
3. If the specified request header is not already available then we will get null.
4. If the header associated with multiple values then this method returns only first value.
5. argument is case insensitive.

Ex: String userAgent=req.getHeader("USER-AGENT");

String userAgent=req.getHeader("user-agent");

1. **public Enumeration getHeaders(String hname):**
   1. returns all values associated with specified header.
   2. If the specified header is not available then this method returns empty Enumeration object but not null.
   3. argument is not case sensitive.
2. **public Enumeration getHeaderNames():**
   1. returns all header names associated with the request.
3. **public int getIntHeader(String hname):**
   1. We can use this method if header associated with int value.

Ex: String length=req.getHeader("content-length");

int l = Integer.parseInt(length);

**Retrieving Cookies from the request:**

HttpServletRequest interface defines the following methods to retrieve cookies from the request.

Cookie[] c = req.getCookies();

**Retrieving Client and Server information from the request:**

We can get client and server information from the request by using the following methods of ServletRequest.

1. **public String getRemoteHost():**

Returns fully qualified name of the client which sends the request.

1. **public String getRemoteAddr():**

returns IP address of the client.

1. **public int getRemotePort():**

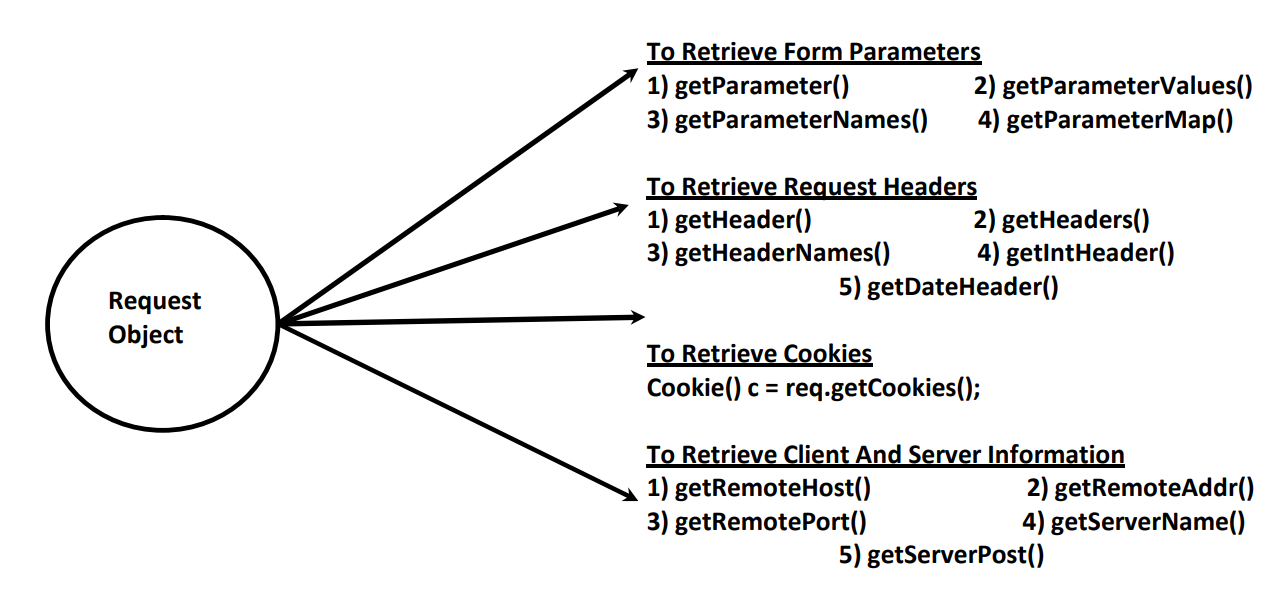
returns the port number on which client is running.

1. **public String getServerName():**

returns the server’s name to which request sent.

1. **public int getServerPort():**

returns port number at which server is running.



**HttpServletResponse:**

By using HttpServletResponse interface, we can:

1. set Response headers.
2. set Content type of Response.
3. acquire Text Stream for response.
4. acquire Binary Stream for response.
5. redirect request to another URL.
6. add cookies to the response.

**set Response headers:**

Http Response headers provides configuration information of the server and information about the response like content-type, content-length etc.

Browser will use these response headers to display response body properly to the end user.

HttpServletResponse interface defines the following methods to add headers to the response.

1. **public void setHeader(String hname,String hvalue):**

If the specified header is already available then old value will be replaced with new value.

1. **public void addHeader(String hname,String hvalue):**

If the specified header is already available then this new value also will be added to existing list of values.

**set content type of response:**

ContentType header represents MIME type(multi purpose internet mail extension)of the response.

Common MIME Types are:

* text/html==>Html text as response
* application/pdf==>pdf file as response
* application/msword==>msword file as response
* application/vnd.ms-excel==>excel file as response
* image/jpeg==>jpeg image file as response
* video/mp4===>mp4 video file as response

We can set content Type of response in the following 2 ways:

1. By ServletResponse:

ServletResponse interface defines the following method for this.

Ex: resp.setContentType("text/html");

1. By HttpServletResponse:

HttpServletResponse interface defines the following method for this.

Ex: resp.setHeader("content-type","text/html");

**Redirecting Http Request to another url:**

Sometimes we have to redirect Http request to another URL. We can implement this by using sendRedirect() method HttpServletResponse interface.

Ex: public void sendRedirect(String targetPath) throws IOE

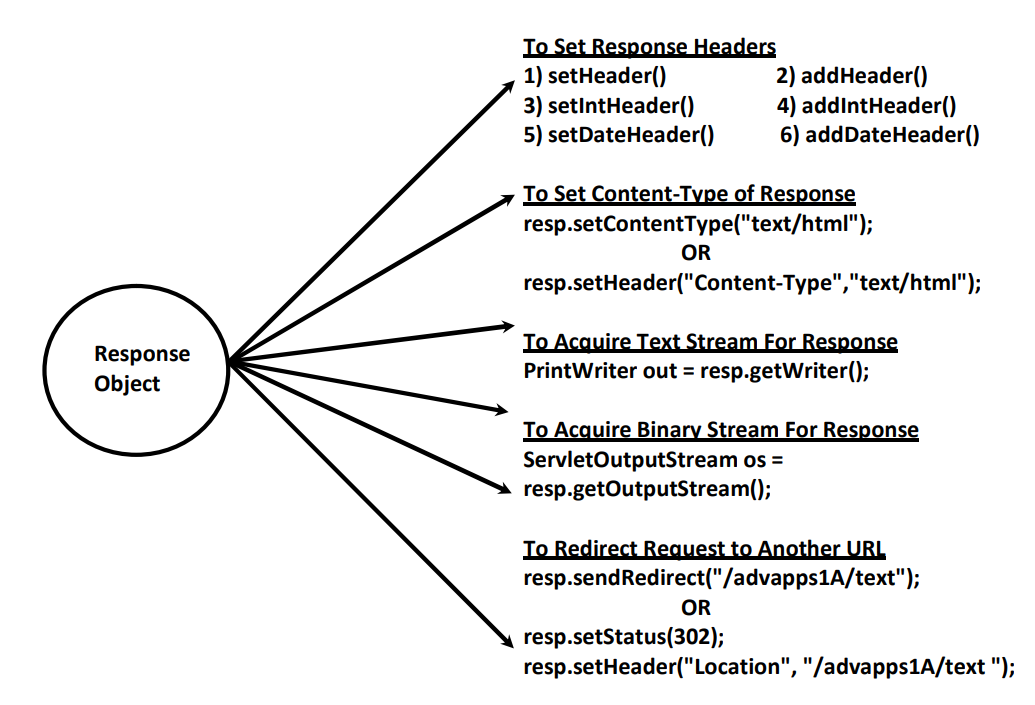
**Adding Cookies to Response:**

Cookie is key-value pair. We can use Cookies in session management.

Cookie c = new Cookie("org","cognovision");

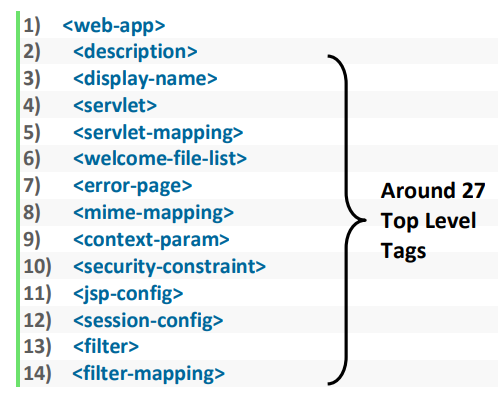
We can add Cookie to the response by using addCookie() method of HttpServletResponse.

resp.addCookie(c);



**Deployment Descriptor: (web.xml):**

Deployment Descriptor (DD) is an XML File named with web.xml.



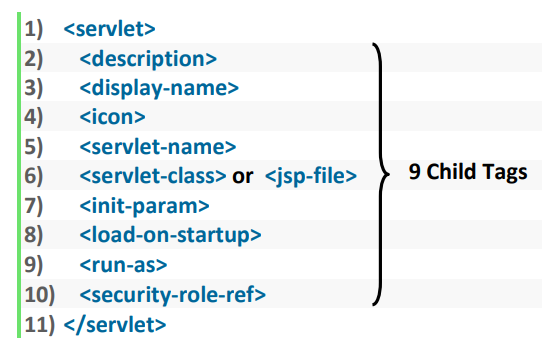
* Web container uses this web.xml to get web application deployment information. i.e., web.xml acts as GUIDE to web container.
* web.xml provides declarative mechanism for customizing our web application.
* For every web application we can maintain only one web.xml.
* Among these 27 top level tags, no tag is mandatory.

1. **<servlet>:**

We can use this tag to declare servlets in web.xml.

<servlet> tag contains the following 9 child tags.

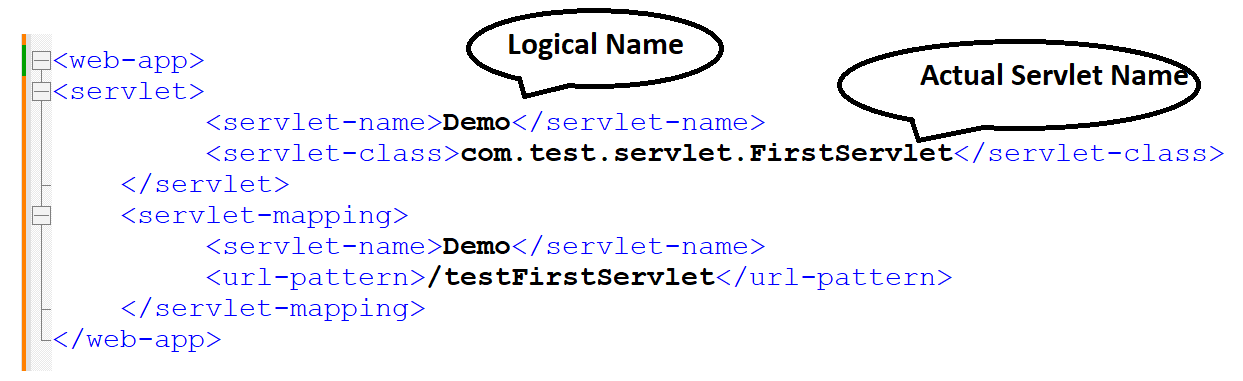
The order of these 9 tags is important and we should not change.



Among these 9 tags the following 2 tags are mandatory:

<servlet-name>

<servlet-class>



**Explanation:**

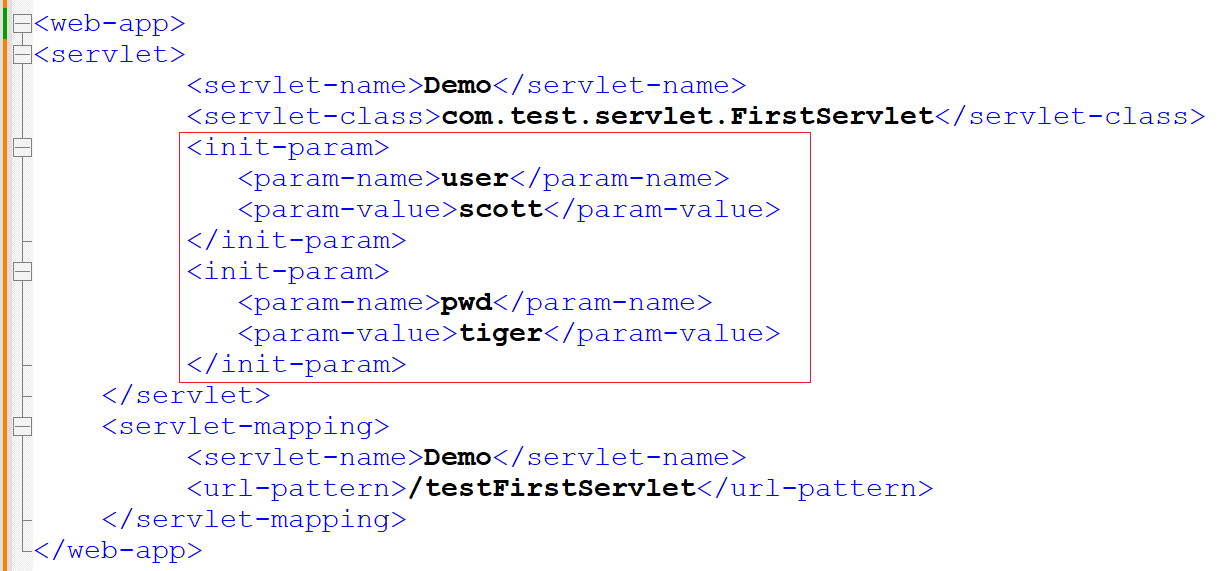
* Actual Name OR Developer's Name specified by <servlet-class>.
* Logical Name specified by <servlet-name> tag.
* End User's Name specified by <url-pattern> tag.
* We can map same servlet with different url-patterns.
* Within web.xml Logical Name is always unique across the servlets.
* Within the servlet we can get Logical Name by using getServletName() method of ServletConfig interface.

**<servlet-class>:**

* By using this tag, we can specify the original name of servlet (Developer's name).
* Web container will create object for this class only. Hence compulsory it should be concrete class but not abstract class or interface.

**Servlet Initialization Parameters(<init-param>):**

* Defines a value available to a single specific servlet within a context.
* If the value of the variable will change frequently then those values are not recommended to hard-code within the servlet class.
* The problem in this approach is, If there is any change in the value, to reflect that change, we have to recompile, rebuild and redeploy application.
* Such type of variables we must configure in web.xml by using tag.
* The advantage in this approach is if there is any change in the value, to reflect that change just redeployment is enough, which won't create big business impact to the client.

****

We can declare any number of init parameters.

Within the servlet we can access servlet initialization parameters by using ServletConfig object.

ServletConfig interface defines the following methods to access these parameters inside servlet:

1. **public String getInitParameter(String pname) :**

Returns the value associated with specified parameter.

Returns null if the specified parameter is not available.

1. **public Enumeration getInitParameterNames():**

Returns all initialization parameter names.

If the specified servlet does not contain any initialization parameters then this method returns empty enumeration object.

Note: GenericServlet implements ServletConfig interface and hence GenericServlet provides

implementation for the above two methods.

**Programs to Execute:** 11InitializeParameterDemo and 12InitializeParameterDemo1

**ServletConfig (I):**

For every servlet web container creates one **ServletConfig** object to hold its configuration information.

By using ServletConfig object, servlet can get its configuration information.

ServletConfig defines the following 4 methods:

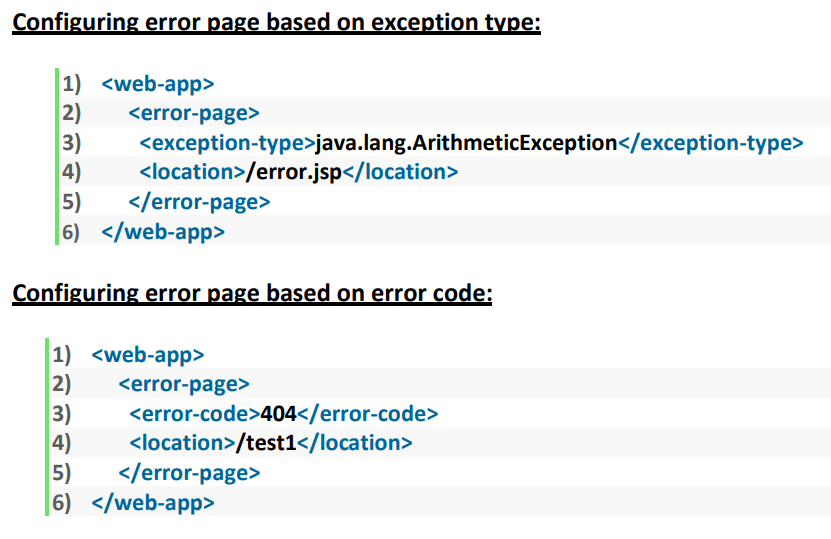
1. public String getServletName()
2. public String getInitParameter()
3. public Enumeration getInitParamterNames()
4. public ServletContext getServletContext()

**GenericServlet provides implementation for all these methods as GS implements ServletConfig Interface**

**Configuring Error Pages in web.xml:**

It is not recommended to send error information directly to the end user. We have to convert that java specific error information into end user understandable form.

We can achieve this by configuring error pages in web.xml. We can configure error page either based on exception type or based on error code.



Execute Program here

**The Web Container Model:**

1. ServletContext
2. Servlet Scopes and Attributes
3. RequestDispatcher
4. Filters
5. Wrappers

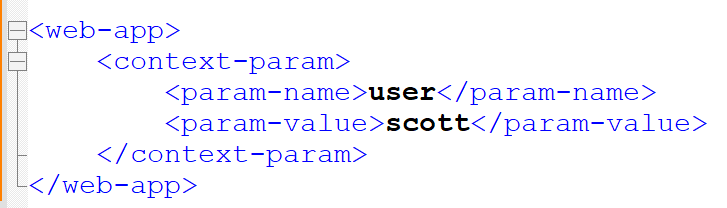
**ServletContext(I):**

For every servlet web container will create one ServletConfig object to hold servlet level configuration information. By using this config object servlet can get its configuration information like logical name of servlet, initialization parameters etc.

Similarly for every web application , web container creates one ServletContext object to maintain application level configuration information.By using ServletContext object, servlet can get application level configuration information like context parameters,RequestDispatcher etc..

**Conclusion**: ServletConfig is per Servlet whereas ServletContext is per web application.

Use <context-param> tag to declare any servlet context values.



We can declare any number of context parameters but separate tag for every parameter.

With in any servlet, we can access context initialization parameters by using ServletContext object.

We can get ServletContext object by using getServletContext() method of ServletConfig interface.

ServletContext context=getServletContext()

**OR**

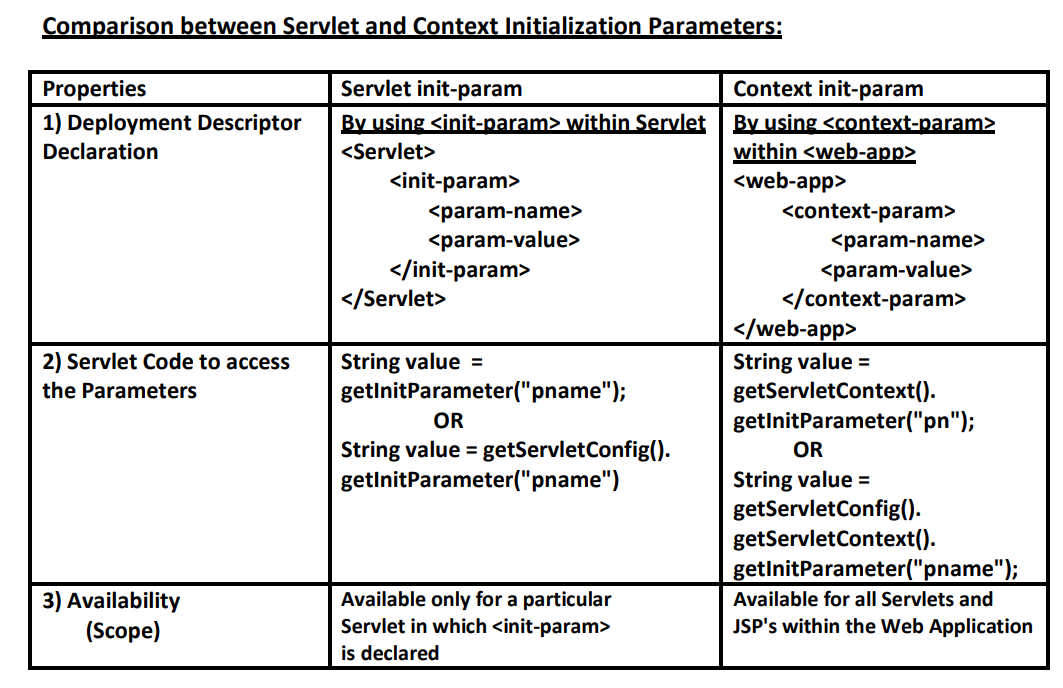
ServletConfig config=getServletConfig();

ServletContext context=config.getServletContext();

ServletContext interface defines the following methods for accessing context initialization parameters.

* public String getInitParameter(String pname)
* public Enumeration getInitParameterNames()

Execute Program here



**Differences between ServletConfig and ServletContext:**

**ServletConfig(I):**

1. For every Sevlet Web Container creates one ServletConfig Object.
2. ServletConfig Object will be created at the time of Servlet Object Creation and destroyed at the time of Servlet Object Destruction.
3. Web Container hand-over Config Object to the Servlet as an Argument to init() Method.
4. Within the Servlet we can get its Config Object as follows:

ServletConfig config = getServletConfig();

1. By using Config Object, Servlet can get its Configuration Information like Logical Name of the Servlet, Initialization Parameters etc.. by using the following Methods.
   * + getServetName()
     + getInitParameter()
     + getInitParameterNames()
     + getServletContext()

**ServletContext:**

For every Web Application, Web Container creates one ServletContext Object to hold Application Level Configuration Information.

Context Object will be created at the time of Application Deployment and destroyed at the time of Application Undeployment.

Within the Servlet we can get Context Object as follows:

ServletContext context = config.getServletContext();

ServletContext context = getServletContext()

On the ServletContext Object we can call the following Methods:

* getRequestDispatcher()
* getInitParameter()
* getAttribute()
* getServletInfo()
* getResourcePaths()
* getContextPath()
* log()
* getMajorVersion()
* getMinorVersion()

**Servlet Scopes and Attributes:**

Servlets Scopes:

* + Request
  + Session
  + Context

**History:**

There are three Types of Parameters available:

1. Form parameters
2. Servlet Initialization parameters
3. Servlet Context Initialization parameters.

These parameters are read-only. i.e., from the servlet we can perform only read operation and we cannot modify, remove values based on our requirement. So, by using Parameters we can’t share data between components of web application.

The above issue can be addressed by using attributes.

**How ?**

Based on our requirement we can create a new attribute; we can modify the value and we can remove existing attribute. Hence **attributes concept is best suitable for sharing data between components of web application**. Based on our requirement, we can store attributes in the proper scope (request scope, session scope and application scope).

**Request Scope:**

* Request scope is maintained by either ServletRequest object or HttpSerlvetRequest object.
* Request scope will start at the time of request object creation(i.e just before calling service() method)and ends at the time of request object destruction(i.e just after completing service() method).
* The data stored in request scope is available for all components which are processing that request.
* For every request a separate new request object will be created, which can be accessed by only current thread. Other threads are not allowed to access request scoped attributes. Hence request scoped attributes are always thread safe.
* The most common application area where we can use Request scoped attributes is RequestDispatcher Mechanism.

ServletRequest interface defines the following methods for attribute management in the Request scope:

1. public void setAttribute(String name, Object value):

To add an attribute. If the specified attribute is already available, then the old value is replaced with new value.

1. public Object getAttribute(String name):

Returns the value associated with the specified attribute. If the specified attribute is not available then this method returns null.

1. public void removeAttribute(String name):
2. public Enumeration getAttributeNames():

**Session Scope:**

Session scope is maintained by HttpSession object.

Session Scope will start at the time of Session object creation.

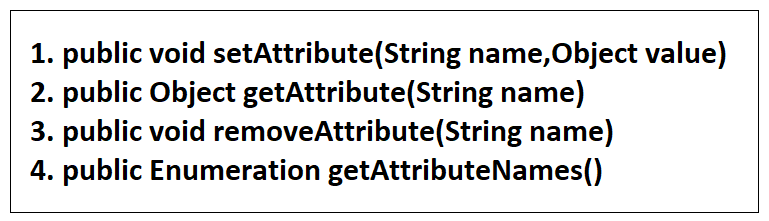
Session scope ends at the time of Session object destruction (i.e., at the time of either logout or timeout).

The information stored in the session scope is available for all the components which are participating in that session.

HttpSession session=req.getSession(); 🡪 Creates session.

Session.invalidate(); 🡪 Invalidate the session.

HttpSession interface defines the following methods for attribute management in session scope:



Once session expired, we are not allowed to call these methods otherwise we will get RuntimeException saying IllegalStateException.

**Ex:** login information should be available for total session. Hence, we must store this information in the session scope.

within the same session we can send multiple requests simultaneously by opening multiple tabs. Hence session object can be accessed by multiple threads simultaneously and hence **session scoped attributes are not Thread Safe**.

**Application Scope:**

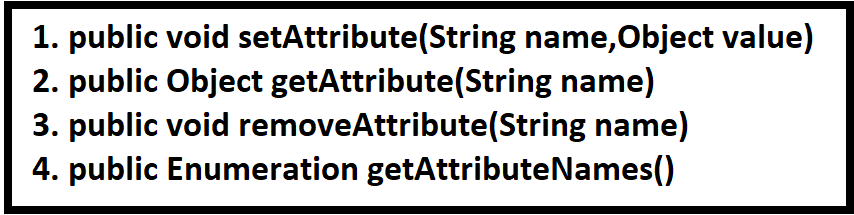
Application scope is maintained by ServletContext object.

This scope will start at the time of context object creation. i.e., at the time of application deployment or server startup.

Application scope ends at the time of context object destruction. i.e., at the time of application un-deployment or server shutdown.

The data stored in application scope will be available to all the components of web application irrespective of request and end user.

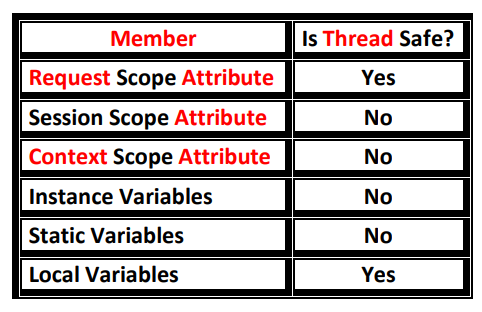
ServletContext interface defines the following methods for attribute management in the application scope...



**Note:** ServletContext object can be accessed simultaneously by multiple threads and hence context scoped attributes are not thread safe.

**In General , saying it** :

* Instance and static variables can be accessed by multiple threads simultaneously and hence these are not thread safe.
* For every Thread a separate copy of local variables will be created, and hence local variable can be accessed by only current thread. Hence local variables are Thread Safe.

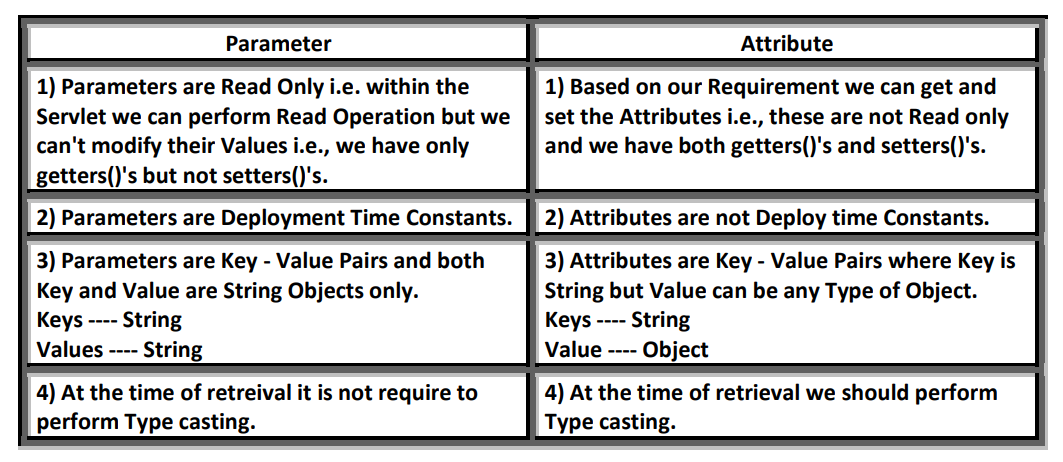


**Note:**

Parameters are key-value pairs where both key and value are String objects. Attributes are also key-value pairs. But keys are String type and **values can be any type**. Which means at the time of retrieval we should type cast the output value.

String user=(String)req.getAttribute("user") 🡪 getting attribute value.

String user= req.getParameter("user"); 🡪 getting parameter value.



Execute Programs:

15HitCountDemo

16CountUserLogin

17CountReqInSession

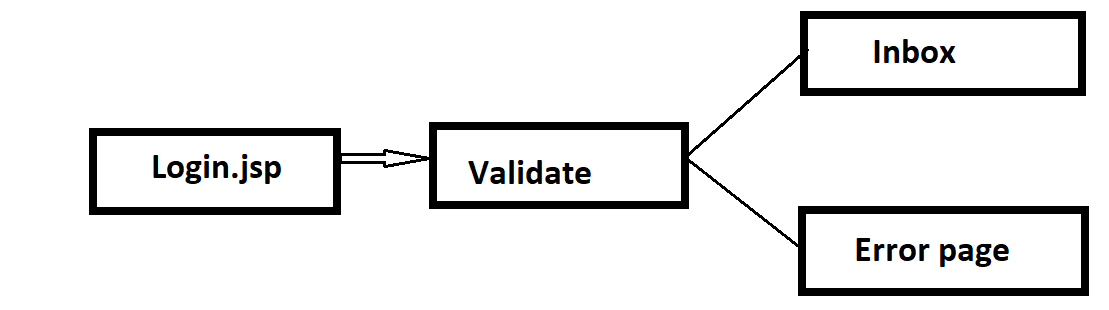
18DisplayAllAttribute

**RequestDispatcher:**

The main purpose of RequestDispatcher is to dispatch our request from one component to another component.

If total functionality is distributed across multiple components, then these components have to communicate with each other to provide response to end user. We can achieve this communication by using RequestDispatcher.

Hence the main purpose of RequestDispatcher is to dispatch our request from one component to another component.



We can get RequestDispatcher either by using ServletRequest object or by using ServletContext object.

**By ServletRequest object:**

ServletRequest interface defines the following method for this purpose.

public RequestDispatcher getRequestDispatcher(String targetResouce)

ex: RD rd = req.getRequestDispatcher("/test2");

RD rd = req.getRequestDispatcher("test2");

**Note:** The target Resource can be specified either **by absolute path or by relative path**.

If the target resource is not available, then we will get 404 status code saying requested resource is not available.

**By ServletContext Object:**

1. **public RequestDispatcher getRequestDispatcher(String targetResource)**

ServletContext interface defines the following methods for getting RequestDispatcher.

ex: RD rd = context.getRequestDispatcher("/test2");

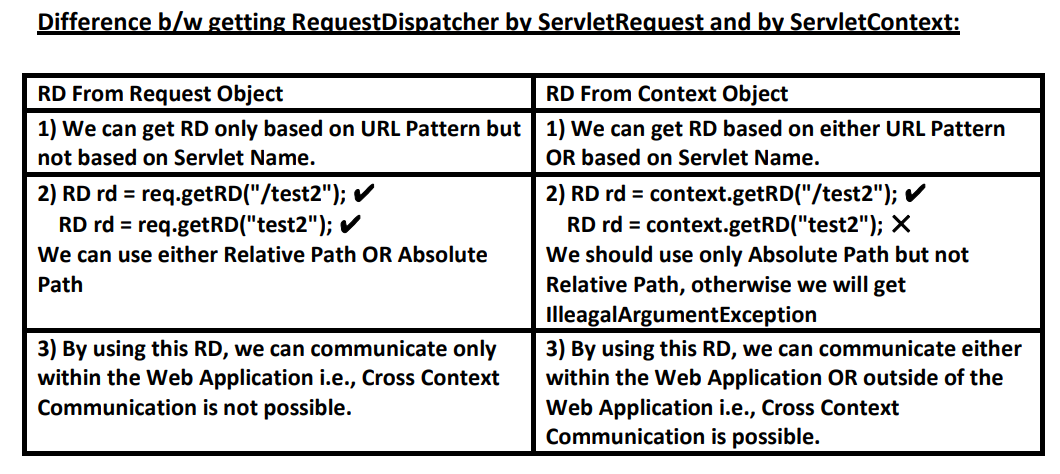
**Note:** The target Resource should be specified by only absolute path.

If we are using relative path (i.e., the path does not start with "/") then we will get Runtime Exception saying IllegalArgumentException.

If the target resource is not available then we will get 404 status code saying requested resource is not available.

1. **public RequestDispatcher getNamedDispatcher(String servletName):**

* The argument represents the value associated with the tag in web.xml. i.e., it represents logical name of the servlet.
* If url-pattern is not available, then we can use this method.
* If the specified servlet is not available, then we will get null. On that null if we are trying to call any method then we will get NullPointerException.
* Ex: RD rd = context.getNamedDispatcher("FirstServlet");

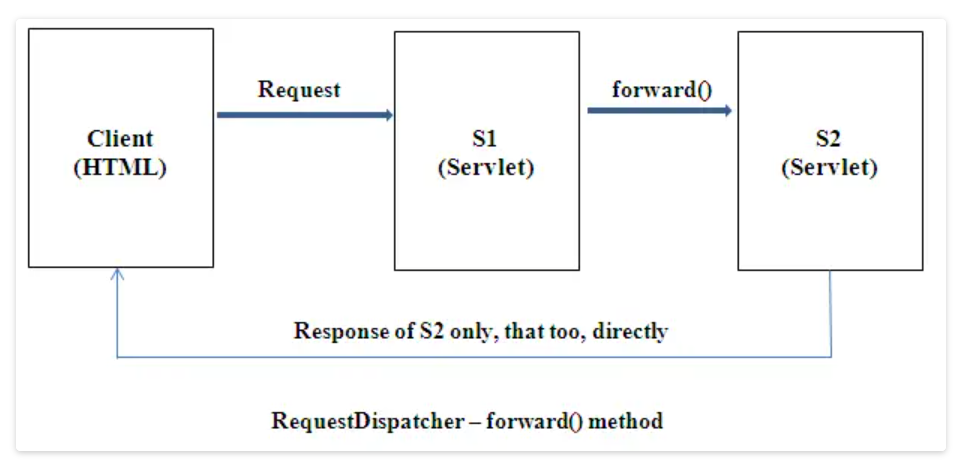


**Methods of RequestDispatcher:**

Once we got RD, we can call the following 2 methods on this object.

1. public void forward(SR req,SR resp)throws SE,IOE
2. public void include(SR req,SR resp)throws SE,IOE

**Forward Mechanism:** In forward mechanism, SecondServlet has complete control on the response . object.



**Execute Examples : 1. 19RDForwardDemo 2. 20RDForwardDemo2**

**Note:**

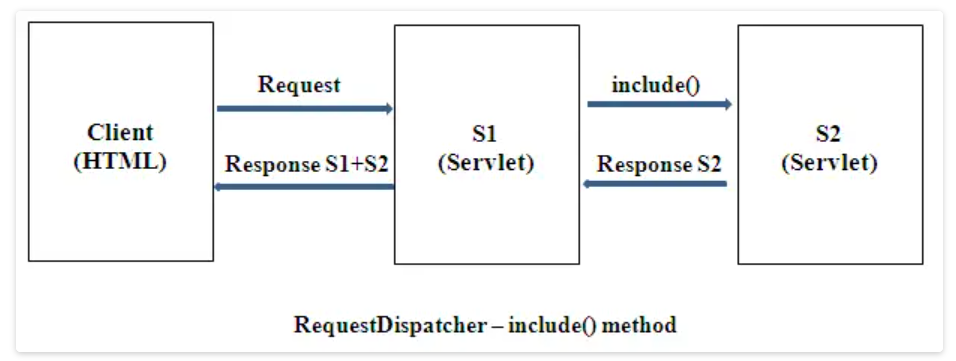
1. Just before forwarding the request, the response object will be cleared automatically by web container. Hence if any response added by FirstServlet won't be displayed to the end user.
2. In Forward Mechanism the same request object will be forwarded to the SecondServlet. Hence information sharing b/w the components is possible in the form of request scoped attributes.

Ex: req.setAttribute("count",10); --🡪 Set value Inside FirstServlet.

Object o = req.getAttribute("count"); 🡪 Retrieve value in SecondServlet like this.

1. After forward call the control comes back to execute remaining statements. In the remaining statements if we are trying to write anything to response, then these statements will be ignored by web container. Only logger statements will be printed. (see example 19RDForwardDemo)

**Include Mechanism:**



* We can use include mechanism to include the response of other resources in the current response.
* The servlet which is getting request initially is responsible to provide response.
* After committing, the response we can perform include() call, but we cannot perform forward call.
* In the include SecondServlet does not have complete control on the response object. It is not allowed to change response headers.

Ex: Execute 22RDIncludeDemo1 program.

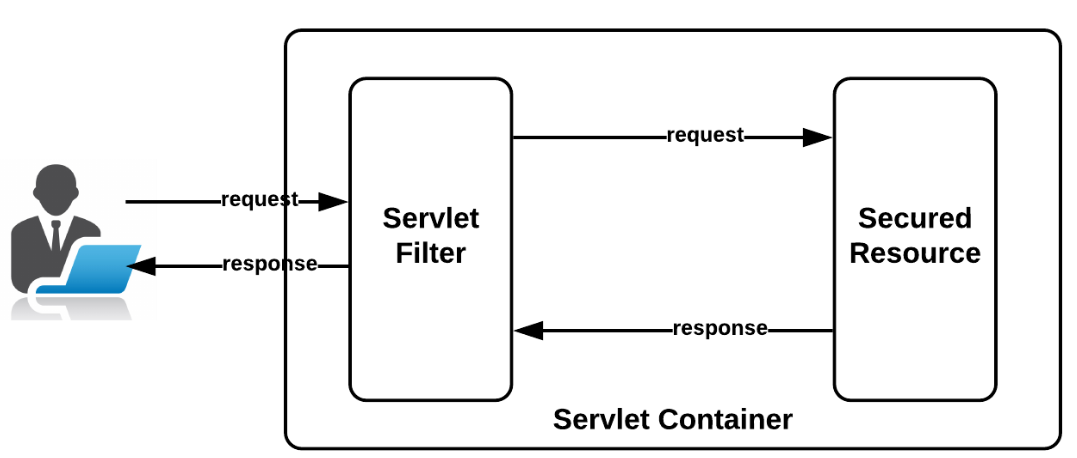
Q1. What is the difference b/w forward() and include(). ?

Q2. What is the difference b/w forward() and sendRedirect(). ?

**Filters:**

Filters can be used for **preprocessing** of request and **post processing** of request before they reach target resource in the web application.

The most common application areas of Filters are logging, security checks, altering request information, compressing response, encryption of response, authentication etc...



**Filter API:**

Filter API contains the following 3 interfaces:

1. Filter
2. FilterConfig
3. FilterChain

**Filter(I):**

Every Filter in Java must implement Filter interface either directly or indirectly.

Filter interface defines the following 3 method:

1. **init():**

**syntax:** public void init(FilterConfig config)throws ServletException

This method will be executed only once to perform initialization activities.

1. **destroy():**

**Syntax:** public void destroy()

This method will be executed only once to perform cleanup activities just before taking the filter from out of service.

1. **doFilter():**

**Syntax:** public void doFilter(ServletRequest req, ServletResponse resp, FilterChain fc)throws SE,IOE

Total **filtering logic** we must define in this method only.

This method will be executed for every request.

By using FilterChain object we can forward the request to the next level. (it may be Servlet or another Filter).

**FilterConfig(I):**

For every Filter, web container creates a FilterConfig object and handover to the filter as argument to init() method.

By using FilterConfig object,Filter can get its configuration information.

FilterConfig defines the following methods...

1. public String getFilterName() ; returns the logical name of the filter which is configured in web.xml.
2. public String getInitParameter(String name)
3. public Enumeration getInitParameterNames()
4. public ServletContext getServletContext()

**FilterChain:**

We can use FilterChain object to forward request to the next level. (It may be another Filter or Servlet).

FilterChain interface contains the following method.

**Syntax:** public void doFilter(SR req,SR resp) throws SE,IOE

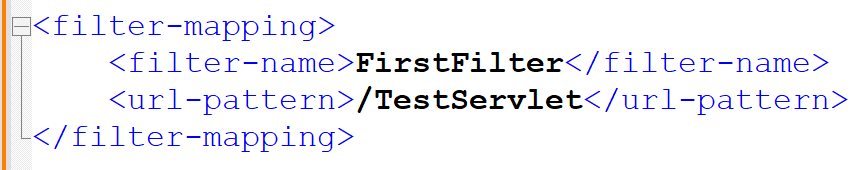
EXECUTE PROGRAM 23FilterDemo HERE.

**Flow Analysis:**

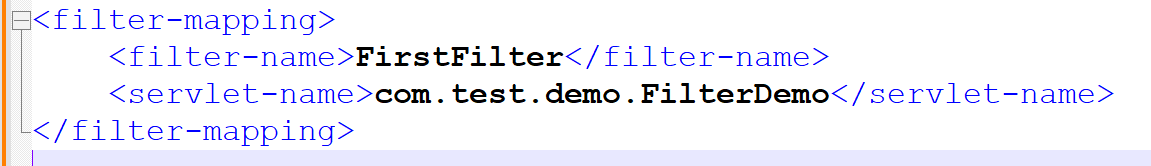
* Whenever we are sending the request to TargetServlet, web container checks whether there any filter configured for this servlet or not.
* If any Filter configured, web container forwards the request to the Filter instead of Servlet.
* After completing Filtering logic, Filter forwards the request to the TargetServlet.
* After processing that request by TargetServlet, the response will be forwarded to the Filter instead of browser.
* After executing Filtering logic, Filter forwards total response to the browser.

**Note:** We can map Filter either for a **Particular url-pattern** or to a **particular Servlet** or to **the total web application.**

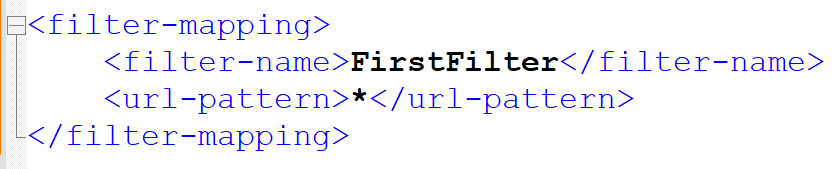
**Filter-Mapping to a Particular url-pattern:**



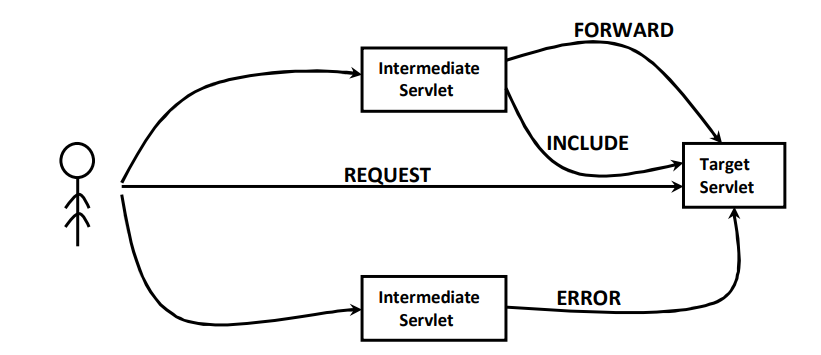
**Filter-mapping to a Particular Servlet:**



**Filter-Mapping for Total web application:**



A servlet can get the request in one of the **following 4** ways:



1. A Request Directly from Browser(REQUEST).
2. By RequestDispatcher's forward() call(FORWARD)
3. By RequestDispatcher's include() call(INCLUDE)
4. By Error Page Call(ERROR)

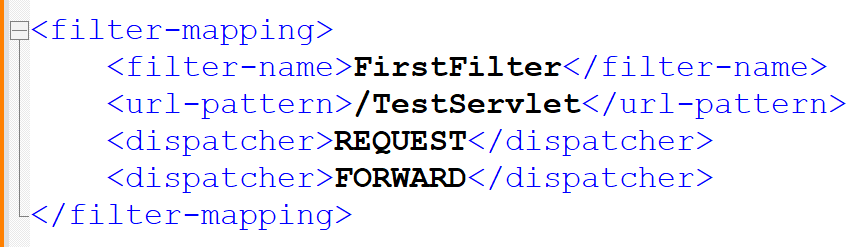
**Conclusion:** By default, Filter concept is applicable **only for direct end user REQUEST** and not applicable for RD's forward, include calls and error page calls.

So, if you want to extend for remaining three cases, then you should go for **<dispatcher>** tag.

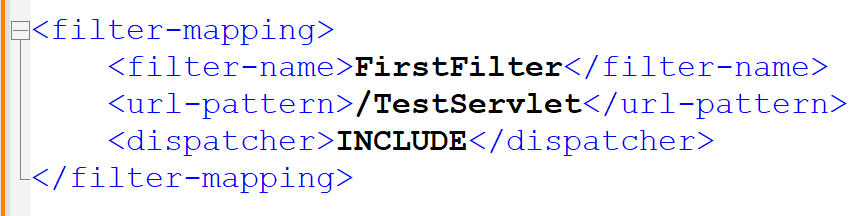
The allowed values for tag are:

1. **REQUEST**: Filter will be executed for direct end user request (default behavior).
2. **FORWARD**: Filter will be executed for RD's forward() call.
3. **INCLUDE**: Filter will be executed for RD's include() call.
4. **ERROR**: Filter will be executed for Error page call.

**Case 1:** If we want to execute Filter for direct end user's request and RD's forward() call, then we have to configured filter as below:

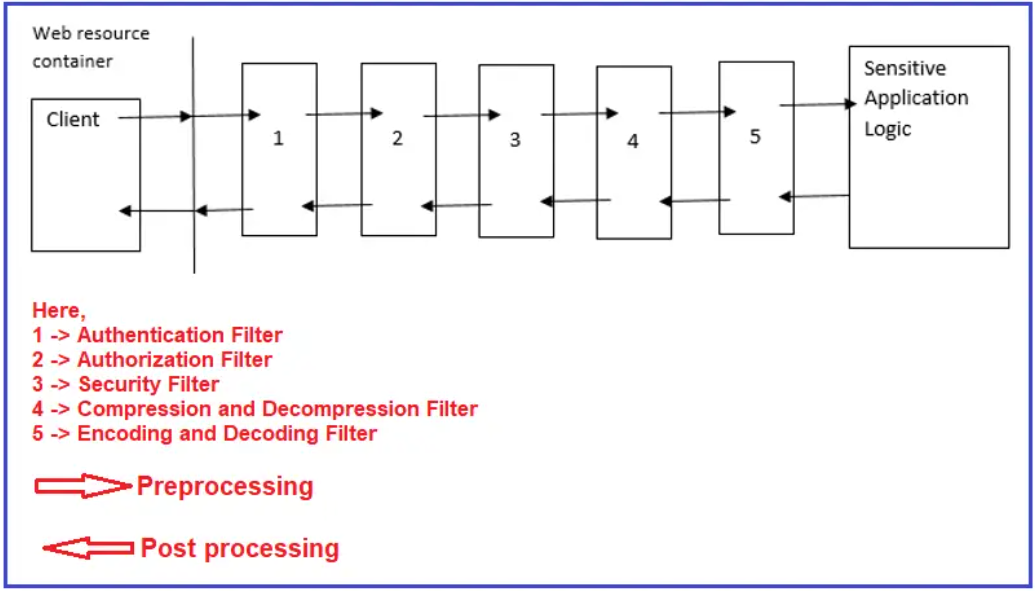


**Case 2:** In we want to execute Filter **only for RD's include** call, then we must configure filter as below:



**Filter Chain:**

We can configure more than one Filter for a Target Resource and all these filters will be executed one by one and forms Filter Chain.

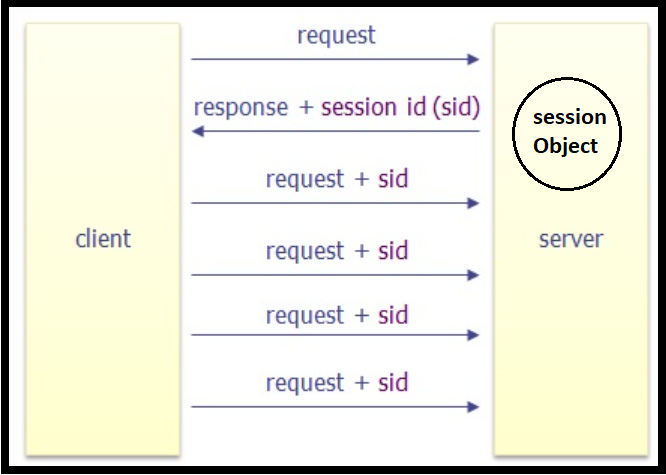


**Session Management:**

* Client and Server can communicate with some common language, which is nothing but HTTP.
* The basic limitation of HTTP is it is stateless protocol. i.e., it is unable to remember client information for future purpose across multiple requests. Every request to the server is treated as a new request.
* Hence some mechanism is required at server side to remember client information across multiple requests. This mechanism is nothing but **session management** mechanism.

The following are various session management mechanisms:

1. Session API
2. Cookies
3. URL Rewriting
4. Hidden Form Fields (Not suggestable to use … Deprecated…)



* Whenever client sends a request to the server, if server wants to remember client information for the future purpose, then server will create a session object and stores the required information in the form of attributes. Server sends the corresponding session id to the browser as the part of response.
* With every consecutive request, browser sends that session id. By accessing session id and the corresponding session object, server can be able to remember client info across multiple requests.
* Client information will be maintained at server side in session object in the form of attributes.

**Process of Creating Session Object:** HttpServletRequest interface defines the following methods for creating session object.

1. **public HttpSession getSession():**

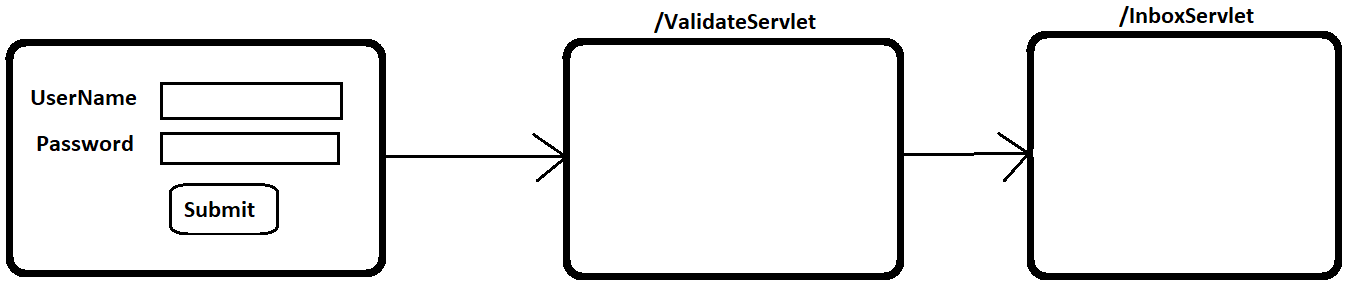
**Syntax:** HttpSession session = req.getSession();

* + First this method will check is there any session already associated with request object or not.
  + If the request does not associate with any session, then this method creates a new session object and returns it.
  + If the request already associated with session object, then existing session object will be returned. There is a guarantee that this method will always return session object. It may be newly created or already existing one.

1. **public HttpSession getSession(boolean b):**

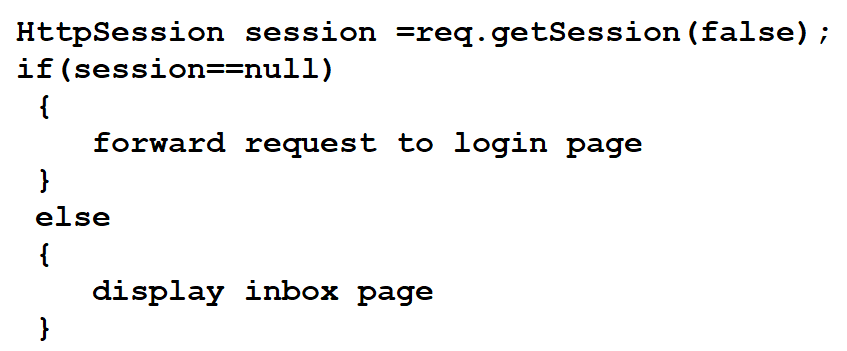
* If the argument is true then this method simply acts as getSession().
* If the argument is false, then this method first checks whether the request associated with any session or not.
* If the request already associated with session, then this method returns existing session object.
* If the request does not associate with any session, then this method returns null without creating any new session object.

**Example:**

****

* ValidateServlet will check whether credentials are valid or not. If valid then it is responsible to create session object. Hence inside ValidateServlet we have to use getSession() method.
* HttpSession session =req.getSession()
* After creating session object ValidateServlet forwards the request to InboxServlet.
* To Access InboxServlet compulsary the request should be associated with session. If the request does not associate with any session, then it is not responsible to create session object and it will forward the request to login page. Hence in this case we must use getSession(false).

**Sample code:**



**Invalidating session object:**

We can invalidate a session by using the following 2 ways :

1. By using **invalidate()** method

2. By Timeout mechanism

**invalidate () method:**

* HttpSession interface defines invalidate() method to invalidate session explicitly.
* public void invalidate()
* whenever we are clicking logout button internally this method will be executed.
* session.invalidate()

**Timeout mechanism:**

If we are not performing any operation on the session object for a pre-defined amount of time then the session will be expired automatically. This predefined amount of time is called session timeout.

We can configure session timeout either server level or web application level or a particular session object level.

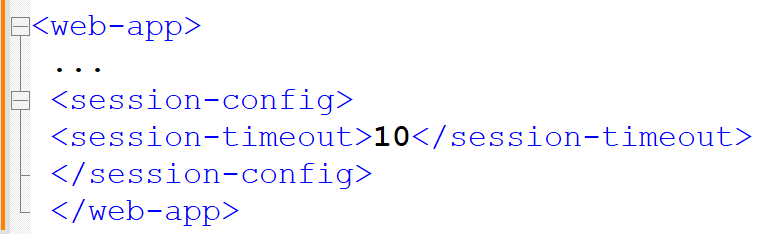
**Session Timeout at Server Level:**

Most of the web servers provide default support for session timeout. Mostly it is 30 minutes. We are allowed to change this server level session timeout based on our requirement. This session timeout applicable for all sessions created in that server of all web applications.

**Configuring session timeout at web application level:**

If we are not satisfied with server level session timeout, then we must configure at application level.

We can configure session time out at application level in web.xml as follows:



The unit to the <session-timeout> tag is **minutes**.

zero or -ve value indicates that session never expires.

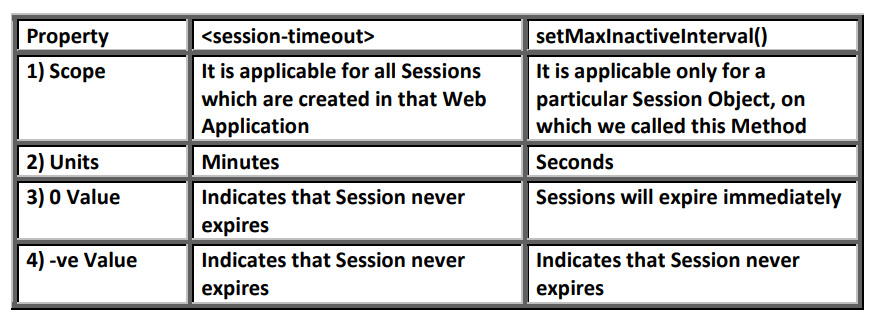
This session timeout is applicable for all the session s which are created in that web application.

**Setting session timeout for a particular session object:**

We can set session timeout for a particular session object by using the following method of HttpSession.

**public void setMaxInactiveInterval(int seconds)**

* The argument is in **seconds.**
* -ve value indicates that session never expires.
* zero value indicates that session will expire immediately.
* This session timeout is applicable only for a particular session object on which we call this method.



**Important methods in session Object:**

* public Boolean isNew() :
  + To check whether the session object is newly created or no
* public void invalidate() :
  + To expire a session forcefully.
* public void setMaxInactiveInterval(int seconds) :
  + To set session timeout for a particular session object.
* public int getMaxInactiveInterval():
  + Returns the session timeout value in seconds.
* public String getId():
  + Returns session id.
* public long getCreationTime():
  + Returns the time when the session was created in milli seconds since Jan 1st 1970.
* public long getLastAccessedTime():
  + Returns the time when the client accessed session recently in milli seconds since 1970 Jan 1st.

HttpSession interface defines the following methods to perform attribute management in session scope:

* public void setAttribute(String name,Object value)
* public Object getAttribute(String name)
* public void removeAttribute(String name)
* public Enumeration getAttributeNames()

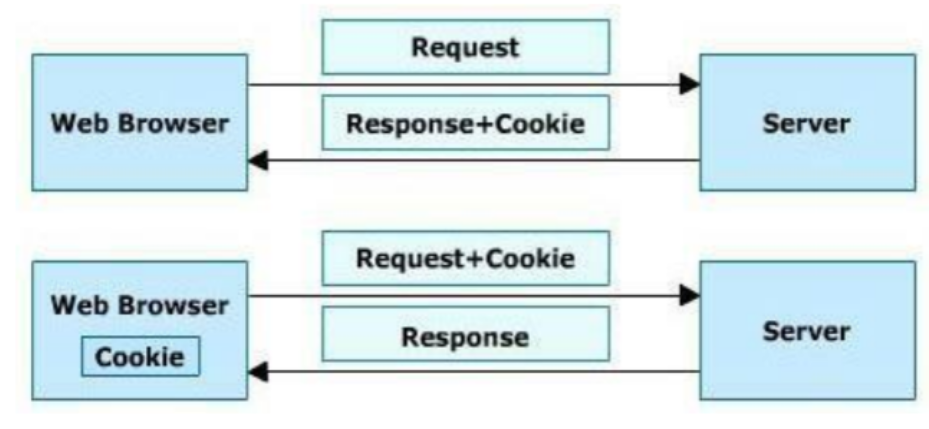
**Execute 26SessionDemo program.**

**Explanation:**

1. Whenever browser sends a request to server, if server wants to remember client information for the future purpose, then Server will create Session object and store required information in the form of attributes.
2. Server sends the corresponding sessionid as the part of response. For this server will use **setCookie response header.**
3. Browser will retrieve that session id and will send with every consecutive request to the server. For this browser will use **cookie request header.**
4. Hence session id exchanging b/w client and server with setCookie response header and cookie request header.

**Session Management by using Cookies:**

* If the required session information is very less, then creating a separate session object and maintaining that object at server side is not recommended because it creates performance problems.
* To resolve this, we should go for Cookies concept, where session information is **maintained at client side** & server is not responsible to maintain session info.
* Cookie is a small amount of information (key-value pair), which is created by server and maintained by client.
* Whenever browser sends first request, if server wants to remember client information for the future purpose, then server will creates Cookie object with the required information and sends to the browser as the part of response.
* Browser stores that cookie in the local file system and sends to the server with every consecutive request. By accessing that cookie server can able to remember client information.
* Server will use setCookie response header to send cookies to the client. Browser will use cookie request header to send cookies to the server.
* Hence by using setCookie response header and cookie request header cookies are exchanging b/w client and server. It is exactly same as exchanging sessionid b/w client and server.



We can create a Cookie object by using Cookie class Constructor:

Cookie c = new Cookie (String name, String value);

Eg: Cookie c = new Cookie (“training”,” Java”);

After creating Cookie object, we have to add that object to the response by using addCookie() method.

resp.addCookie(c);

At server side we can retrieve cookies send by the client from request object by using getCookies() method.

Cookie[] c = req.getCookies();

**Important methods of Cookie:**

1. public String getName() : returns the name of the Cookie
2. public String getValue() : returns the value of the Cookie
3. public int getMaxAge() : Returns the max age of the Cookie in seconds.
4. public void setMaxAge(int seconds) : To set max age of the cookie.
   1. setting max age as -1, then cookies will be expired automatically whenever browser window closed. **-1 is the default value.**

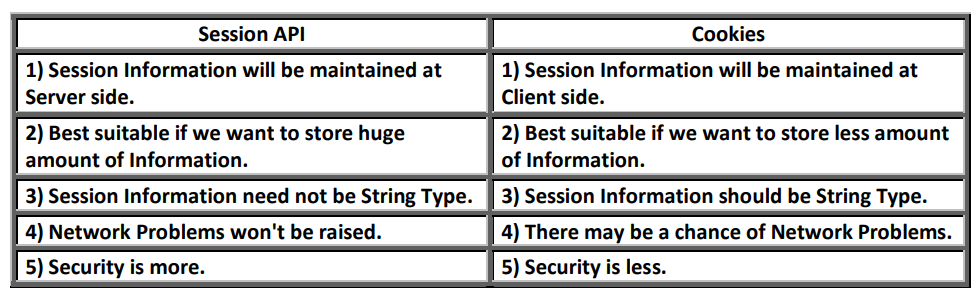
**Execute Program**

Advantages of Cookies:

* Very easy to implement.
* Persist across server restarts also.
* All browsers and servers provide automatic support for cookies.

Disadvantages of Cookies:

* Cookies can be enabled or disabled at client side to meet security constraints. If the cookies are disabled, then session management by using cookies is not possible.
* The number of cookies supported by any browser is always fixed.
* The max size of the cookie is also fixed. Hence, we cannot store huge amount of information by using Cookies.
* Cookie data is always String type.



**Session Management by URL REWRITING:**

**Why URL Rewriting?**

If the cookies are disabled at client side, then browser is unable to see Set-Cookie response header. Hence browser won’t get any cookies or session id send by server. If the cookies are disabled at client side, then browser unable to send Cookie request header. Hence server won’t get any cookies or session id from the request and every request is treated as new request. Due to this total session management fails.

To overcome this problem, we should go for: URL REWRITING.

**What is URL Rewriting?**

URLs can be re written or encoded to include session information. This technique is called url rewriting.

**URL Rewriting=URL+Session Info**

**methods to append session id to the url:**

1. public String encodeURL(String url) : Returns URL by appending JSESSIONID.
2. public String encodeRedirectURL(String url):
   1. Returns url by appending session id.
   2. This can be used as argument to sendRedirect() method.

The above 2 methods will append JSESSIONID to the URL if cookies are disabled at client side.

**Note**: If the cookies are enabled, these methods return the same URL without appending JSESSIONID.

At server side we can identify whether sessionid is coming as the part of url or from the Cookie request header by using the following methods of HttpServletRequest.

1. public boolean isRequestedSessionIdfromURL()
2. public boolean isRequestedSessionIdfromCookie()

By using these methods, we can identify underlying session management technique.

**Execute Program:**

Advantages of URL Rewriting:

There is no chance of disabling URL rewriting technique. Hence it will work always.

Limitations of URL Rewriting:

1. It is very difficult to rewrite all URLs to append session information. Hence it is the most painful technique.
2. 2.URL Rewriting will work only for dynamic documents.

**Listeners**

In the web application there may be a chance of occurring several events like

* Request object creation
* Request object Destruction
* Session object creation
* Session object Destruction
* Context object creation
* Context object destruction
* Attribute addition in request scope
* Attribute Removal in request scope
* Attribute Replacement in request scope
* ………… etc.

whenever these events occur, if we want to do particular operation automatically then we should go for listeners.

i.e., Listener listens the events and will perform certain operations automatically.

**All Listeners are divided into 3 groups:**

Request Listeners:

These listen events related to request.

1. ServletRequestListener
2. ServletRequestAttributeListener

Session Listeners:

These listen the events related to session.

1. HttpSessionListener
2. HttpSessionAttributeListener
3. HttpSessionBindingListener
4. HttpSessionActivationListener

Context Listeners:

These listen events related to context.

1. ServletContextListener
2. ServletContextAttributeListener.