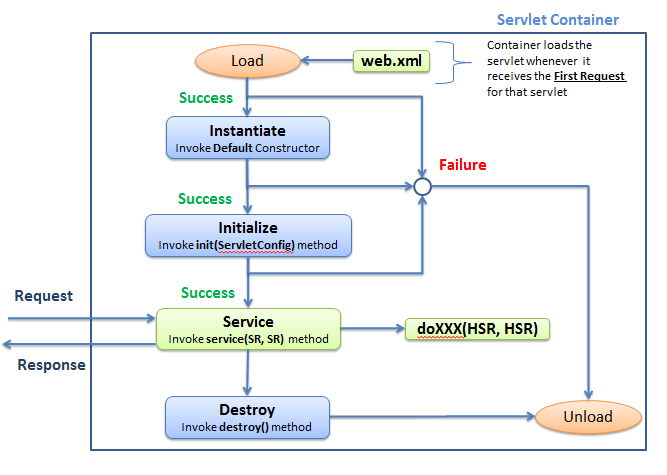
**Servlet Lifecycle:**



The lifecycle of a servlet is controlled by the Servlet Container. Servlet Lifecycle has following stages.

1. Instantiation Phase
2. Initialization Phase
3. Service Phase
4. Destruction Phase

**Note:**

1. Any Java class which extends either HttpServlet or GenericServlet class is called as Servlets
2. **Servlets are protocol independent in nature**, but are most often used with the HTTP & HTTPS protocols
3. If a class extends either HttpServlet or GenericServlet & a subclass of that class can also be called as Servlet
4. Servlets can have local variables, it’s own methods (static / non-static), block of code (static / non-static), inner class etc., but they cannot have abstract methods
5. Servlets must have public default (i.e. no argument) constructor or combination of any other constructor along with public default constructor
6. We can also have main() method in servlet but it’s of no use
7. There is only one instance exist for any particular servlet class. i.e. **Servlets are Singleton in nature**
8. Every request to a servlet runs in a separate thread.
9. **Instantiation Phase:-**

* Whenever the request comes to servlet, container using the URL information from request & referring the web.xml, it comes to know the Servlet configured for that URL.
* If there is NO servlet is configured, then it return the 404 error response
* If the servlet is configured for that URL in web.xml then it looks out for servlet instance in its cache. If not present (for the first request) then Load the Servlet class and **executes the static block of the servlet (if any).** Then container creates an instance of the servlet by invoking the public default constructor

1. **Initialization Phase:-**

Version 1: **public** **void** init(ServletConfig config)

**throws** ServletException

{

**super**.init(config);

//Initialization Code goes here

}

Version 2: **public** **void** init()

**throws** ServletException

{

//Initialization Code goes here

}

* Once Container successfully creates the servlets instance then it automatically invokes init(ServletConfig config) method
  + - Init() method gives us a chance to initialize the servlet before handling any client requests. Like, read data from property file etc.
    - **During initialization servlet has access to two key objects**
      * ServletConfig
      * servletContext
* This method is called **ONLY Once** in the lifecycle of a servlet
* **We may possibly override this method** & If we don’t override init() method then one from the GenericServlet is called
* If an error occurs during the initialization phase, servlet throws a ServletException. Then, servlet container unloads the servlet & put it for garbage collection. **In this case destroy() method is not called**
* The first line of version 1 init() method **Should** be “**super**.init(config)”. This allows GenericServlet to save a reference to ServletConfig & make it available to methods outside of the init()
* Once the instantiation & initialization is successful, container caches the servlet instance i.e. **servlets are singleton in nature**

**Note:**

* We can use constructor in servlet for initialization purpose, but this type of approach is not so common. Also init() method has access to the two key objects such ServletConfig & ServletContext whereas constructor don’t . Hence we should make use of init() method for initialization purpose.

1. **Service Phase:-**

**public** **void** service(ServletRequest req, ServletResponse res)

**throws** ServletException, IOException

{

//Service Code goes here

}

* Once the instantiation & initialization phase is successfully completed, container creates Request & Response objects based on the type of request it receives. i.e
  + for Non-Http requests it creates Objects of type ServletRequest & ServletResponse
  + for Http requests it creates Objects of type HttpServletRequest & HttpServletResponse
* Creates a thread for the incoming request & instructs the thread to run the servlets **service(ServletRequest req, ServletResponse resp)** method by passing request and response objects
* Depending on the HTTP Method in the request, the service() method calls relative doXXX() methods. Let say in our case it calls doGet() method
* Service method is called whenever client request come in. i.e. this method is called **one / more times** in servlet lifecycle. Most of the servlet’s life is spent running a service() method for a client request
* We **SHOULD NOT** override the service method. Our job is to override doXXX() methods & let service() method implementation from HttpServlet worry about calling the right doXXX() method
* Once the execution of doXXX() method is over, the thread dies, container converts the response object into an HTTP Response & sends it back to the webserver.
* Container then **garbage collects the request & response objects**. i.e. container creates new request & response objects for every request & response cycle

1. **Destruction Phase:-**

**public** **void** destroy()

{

//Clean-up Code goes here

}

* Whenever container wants to unload the servlet it calls the destroy() method before unloading the servlet from service.
* The decision of when to destroy a servlet instance rests on the shoulders of servlet engine. Developers should not be concerned with these details, but instead focus on what should be done when the time comes.
* Once the destroy() method has completed the servlet engine releases the servlet from service & makes it eligible for garbage collection
* This method is called **ONLY Once** in the lifecycle of a servlet
* **We may possibly override this method** & If you don’t override destroy(), then one from the GenericServlet is called
* destroy() method is similar to finalize() method in garbage collection

**About <load-on-startup> Tag:**

* By default, servlets are only instantiated & initialized at first request. Therefore, time taken to generate the response of the first request to the servlet is more
* This can be avoided by preloading the servlet at deployment time by including <load-on-startup> tag specifying a value >= 0. For example,

**<servlet>**

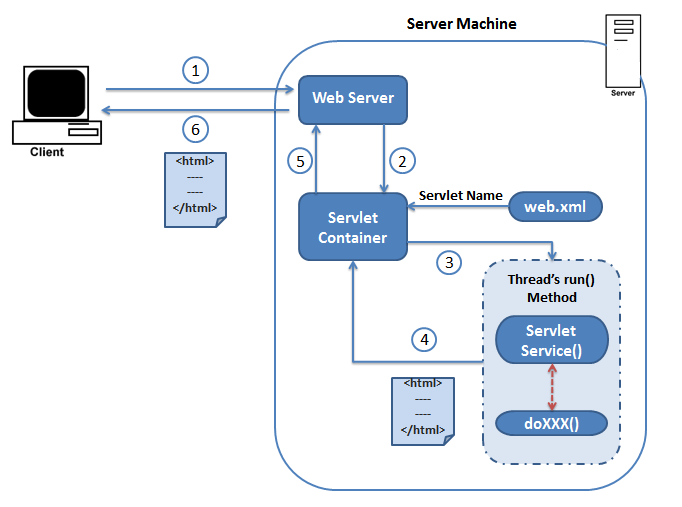
**<servlet-name>MyServlet</servlet-name>**

**<servlet-class>myPackage.MyServlet</servlet-class>**

**<load-on-startup>1</load-on-startup>**

**</servlet>**

Order in which servlets are preloaded depends on the value of the <load-on-startup>

**How Dynamic Request is handled in Web Server:-**

Step 1: Client requests for a dynamic resource. Let say HTML page with current date & time

Step 2: Webserver does not find the resource with in itself & it handover that request to Servlet Container

Step 3:

1. Container finds the correct servlet based on the request URL by using Web.xml
2. It tries to find the instance of that servlet in its cache
3. If the instance is not found (i.e. request come for the first time) then container creates the instance of the servlet by **invoking default constructor** & once instantiation is successful it initializes the servlet by **invoking init(ServletConfig config) method**
4. Once the instantiation & initialization is successful, container caches the servlet instance
5. Servlet Container creates Request & Response objects based on the type of request it receives. i.e
   * + for Non-Http requests it creates Objects of type ServletRequest & ServletResponse
     + for Http requests it creates Objects of type HttpServletRequest & HttpServletResponse
6. Creates a thread for the incoming request & instructs the thread to run the servlets **service(ServletRequest req, ServletResponse resp)** method by passing request and response objects
7. Depending on the HTTP Method in the request, the service() method calls relative doXXX() methods. Let say in our case it calls doGet() method

Step 4:

1. doGet() method constructs the brand new HTML page along with current date time & stuff that page into the response object.
2. The thread completes, container converts the response object into an HTTP Response & sends it back to the webserver.
3. Container then **garbage collects the request & response objects**.

Step 5: Webserver gets the response back from Servlet Container. **As far as Webserver is concerned; the HTML from the servlet container is a static page**

Step 6: Client get the requested resource with the current date & time

**Explain the below concepts using the below Programs**

**public** **interface** ServletRequest

{

//Some Methods

}

**public** **class** NonHttpReq **implements** ServletRequest

{

//Implement methods of ServletRequest

}

**public** **interface** HttpServletRequest **extends** ServletRequest

{

//Some More Methods

//i.e. Methods of ServletRequest + Methods of HttpServletRequest

}

**public** **class** HttpReq **implements** HttpServletRequest

{

//Implement Method of HttpServletRequest

}

**public** **class** **MyGenericServlet**

{

**public** **void** service(ServletRequest sr)

{

System.*out*.println("Inside service 1");

}

}

**public** **class** **MyHttpServlet**

{

**public** **void** service(ServletRequest sr)

{

System.*out*.println("Inside service 1");

HttpServletRequest request;

request = (HttpServletRequest) sr;

service(request);

}

**public** **void** service(HttpServletRequest request)

{

System.*out*.println("Inside service 2");

doXXX(request);

}

**public** **void** doXXX(HttpServletRequest request)

{

System.*out*.println("Inside doXXX Method");

}

}

**public** **class** ServletContainer

{

**public** **static** **void** main(String[] args)

{

/\*

\* Non-HTTP Request

\*/

MyGenericServlet genericServlet = **new** MyGenericServlet();

NonHttpReq request = **new** NonHttpReq();

ServletRequest request = **new** NonHttpReq();

genericServlet.service(request);

/\*

\* HTTP Request

\*/

MyHttpServlet httpServlet = **new** MyHttpServlet();

HttpReq request = **new** HttpReq();

HttpServletRequest request = **new** HttpReq();

ServletRequest request = **new** HttpReq(); //Important & Works

//Important; You get java.lang.ClassCastException @ Runtime

//Upcast & Downcast are decided at Runtime

ServletRequest request = **new** HttpReq();

httpServlet.service(request);

}

}

NOTE :

Servlet Container creates Request & Response objects based on the type of request it receives. i.e

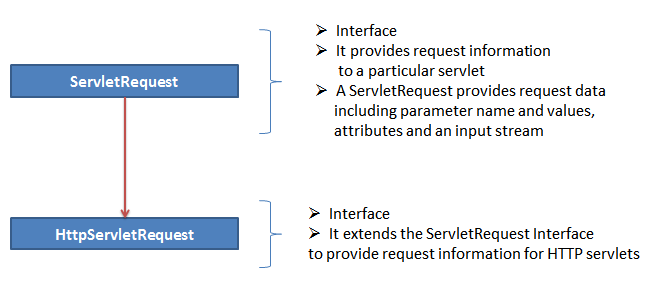
* + for Non-Http requests it creates Objects of type ServletRequest & ServletResponse
  + for Http requests it creates Objects of type HttpServletRequest & HttpServletResponse

i.e something similar to

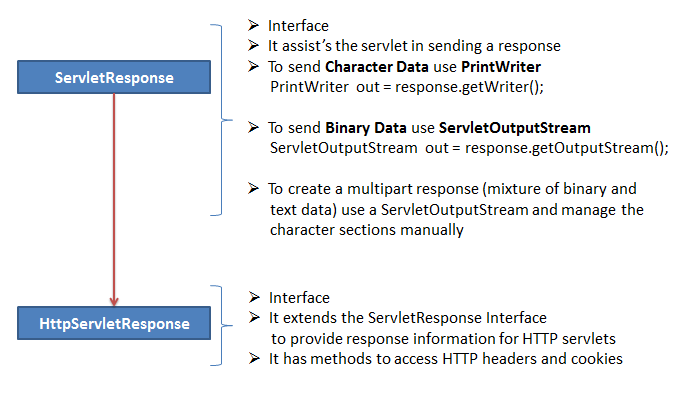
ServletRequest request = **new** NonHttpReq();

ServletRequest request = **new** HttpReq();

**ServletRequest**



**ServletResponse**

****

**Talk about Inheritance; Create a Superclass & Subclass & ask them what happens when we create an instance of subclass. Explain how the dynamic request is handled at Webserver side & explain why we need to override doXXX() methods**

**public** **abstract** **class** SuperClass

{

**public** **abstract** **void** method1(Object obj1, Object obj2);

}//End of Class

**public** **class** SubClass **extends** SuperClass

{

@Override

**public** **void** method1(Object obj1, Object obj2)

{

// Some Business Logic

System.*out*.println("Overriding Super Class 'Class Method' ");

method1("", "");

}

**public** **void** method1(String str1, String str2)

{

// Some More Business Logic

System.*out*.println("Sub Class 'classMethod'");

method2("", "");

}

**public** **void** method2(String str1, String str2)

{

// Something Else

System.*out*.println("Sub Class 'myMethod' ");

}

}//End of Class

**public** **class** MyClass **extends** SubClass

{

@Override

**public** **void** method2(String str1, String str2)

{

System.*out*.println("My Own Logic");

}

@Override

**public** **void** method1(String str1, String str2)

{

System.*out*.println("My Own Logic 2 ");

}

@Override

**public** **void** method1(Object obj1, Object obj2)

{

System.*out*.println("My Own Logic 2 ");

}

}//End of Class

**public** **class** MainMethodClass

{

**public** **static** **void** main(String[] args)

{

Object obj1 = **new** Object();

Object obj2 = **new** Object();

MyClass myClass = **new** MyClass();

myClass.method1(obj1, obj2);

}

}//End of Class

**Correlate**

SuperClass = GenericServlet

SubClass = HttpServlet

MyClass = MyServlet

MainMethodClass = ServletContainer

SuperClass.method1() = GenericServlet.service(ServletRequest,

ServletResponse)

SubClass.method1() = HttpServlet.service(HttpServletRequest,

HttpServletResponse)

SubClass.method2() = HttpServlet.doGet(HttpServletRequest,

HttpServletResponse)

MainMethodClass.obj1 = ServletRequest

MainMethodClass.obj2 = ServletResponse

<servlet>

<servlet-name>nonHttp</servlet-name>

<servlet-class>com.qspiders.lms.servlets.NonHttpServlet</servlet-class>

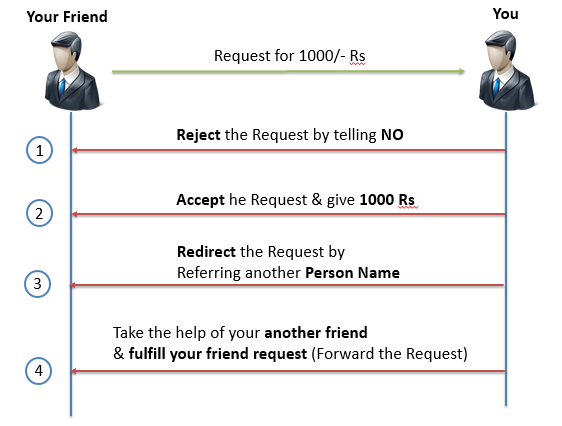
</servlet>

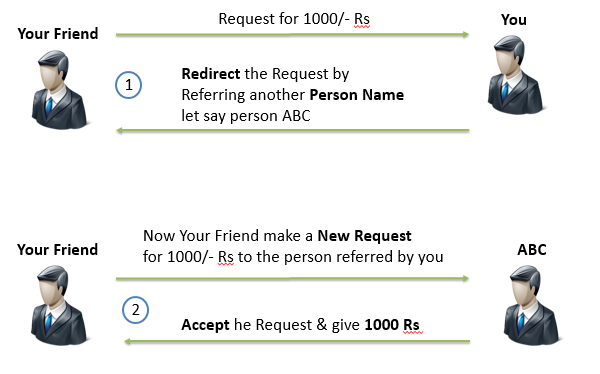
<servlet-mapping>

<servlet-name>nonHttp</servlet-name>

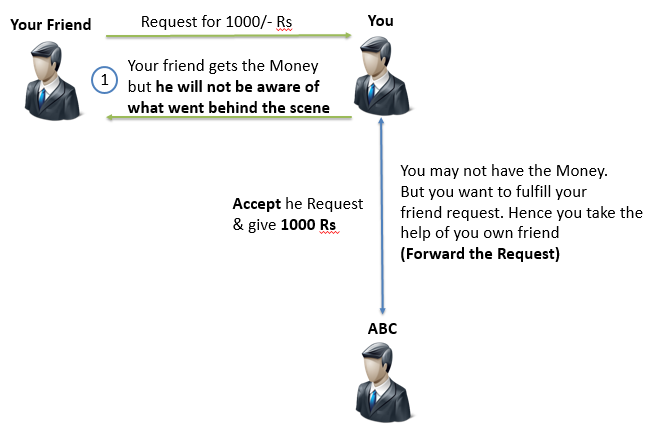
<url-pattern>/nonHttpRequest</url-pattern>

</servlet-mapping>

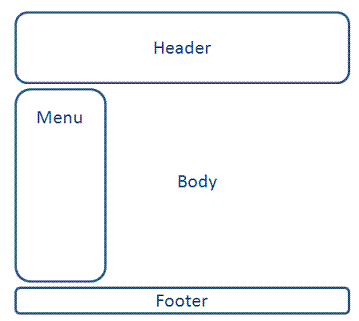


****

**Real Time Example of Send Redirect:** You are buying a HP laptop & at that time HP Shopping cart site will suggest the Cisco Router to buy for that laptop. If you click on that link then the request get **Redirected** to Cisco site (third party punch out) & you can buy that router over there & once done again the request will get **Redirected** back to HP Shopping cart site.

****

Using the below picture explain the Include Advantage



Recap the ServLetContext working ask students to go to ServletContext topic to see the first point which says about One context object per application & lifecycle of Context object = lifecycle of the application. Hence attributes leave as long as long application is alive.

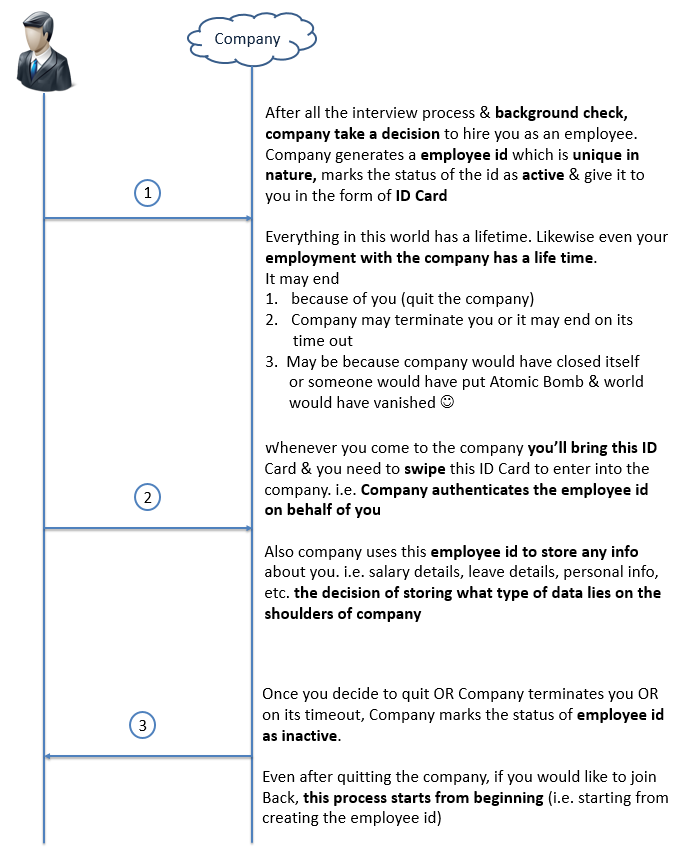
Recap the how dynamic request is handled section stress on the point which says request & response objects are deleted once the response sent back to the client. Hence attribute set to request scope will leave as long as request is severed. Once the response is sent back to client then these attributes will be deleted.

**Different ways to pass information to servlets:**

1. Using Query String & HTML form
2. Using ServletContext & ServletConfig init params
3. Using Attributes

**Difference between Attribute & Parameter**

|  |  |  |
| --- | --- | --- |
| **No.** | **Attribute** | **Parameter** |
| 1 | Return type is Object | Return type is String |
| 2 | We can set or remove an attribute | We Cannot set or remove the parameter |
| 3 | There are 3 types of Attributes   1. Context attributes 2. Request attributes 3. Session attributes | There are 3 types of parameters   1. Context Init Parameter 2. Servlet Init Parameter 3. Request Parameters (form data & query string) |
| 4 | We can get the attribute by using the method getAttribute(String name) | We can get the initialization parameters by using getInitParameter(String name) method & request parameters using getParameter(String name) method |

****

If the user's browser supports cookies, then servlet container makes use of cookies to handle session.

If the browser does not support cookies or the user explicitly disabled it, then servlet container reverts to URL-rewriting. All these details are handled by HttpSession automatically and it happens behind the scene. Developer need not to worry about handling these things.

1. HttpSession HttpServletRequest.getSession()

Returns the current session associated with the current request. If the request does not have a session, this method creates a new session.

1. HttpSession HttpServletRequest.getSession(boolean create)

* If create = true, returns the current HttpSession associated with the current request or **if there is no session exists, returns a new session.**
* If create = false, returns the current HttpSession associated with the current request or **return null if there's no session exists**

1. String HttpServletResponse.encodeURL(String url)

Encodes the specified URL by appending the session ID to it. We have to use this method to encode Forward, Hyperlink & Form URL’s

1. String HttpServletResponse.encodeRedirectURL(String url)

Encodes the specified URL by appending the session ID to it. We have to use this method to encode Redirect URL’s

**Why two methods to encode URL?**

Because the rules for making the determination can differ from those used to decide whether to encode a normal link & hence encodeRedirectURL method is separated from the encodeURL method.

1. boolean isNew()

This returns true if the session is newly created and returns false for pre-existing sessions.

When client does not choose to join a session (when cookies are intentionally turned off), **getSession()** method (present in HttpServletRequest) will return a different session on each request and **isNew()** method always returns true (until the client joins the session).

1. long getCreationTime()

Returns the time when this session was created, measured in milliseconds

1. long getLastAccessedTime()

Returns the last time the client sent a request associated with this session, as the number of milliseconds

1. int getMaxInactiveInterval()

Returns the session time in seconds. After this interval, the servlet container will invalidate the session.

* + After this, below things happens behind the scene.

1. Container creates the Unique Session ID & sends it to the user browser (using cookie where Name=JSESSIONID & value= Unique Session ID) along with the **response**.
2. Whenever user makes any future requests, browser sends this cookie to the container along with the **request**.
3. Container stores any on-going transaction related information (if any, using session attributes) at **server side** against this ID

Once the session is created, for the subsequent requests validate the session (where ever necessary, it depends on the application for example, for Gmail & Facebook kind of applications you have to validate for every request & for Flipkart kind of application validate only the certain requests)

**Logic written in req.getSession(true) OR req.getSession() methods**

**if**(JSESSIONID is present in Cookie)

{

**return** the existing Session Object;

}**else** **if**(JSESSIONID is present in URL){

**return** the existing Session Object;

}**else**{

**return** **null**; // In case of req.getSession(true)

**return** new Session Object; // In case of req.getSession()

}

**URL Rewriting:**

* In handling user session, URL Rewriting comes into picture **ONLY if we tell server to encode URL’s**
* If we encode our URL’s (Redirect / Forward / Hyperlink), the container will first attempt to use cookies for session management & fall back to URL rewriting only if cookie approach fails
* We cannot use URL Rewriting methods inside static pages. So if our application is dependent on session we must carefully review our decision of using static pages. In this case we should not go for static pages which have links to Dynamic pages. Using only static page which has some content init is fine.
* We have to use HttpServletResponse.encodeURL() method to encode Forward & Hyperlink URL’s and

we have to use HttpServletResponse.encodeRedirectURL() method to encode Redirect URL.

**Filters**

When the servlet container calls a method in a servlet on behalf of the client, the HTTP request that the client sent is, by default, passed directly to the servlet. The response that the servlet generates is, by default, passed directly back to the client, with its content unmodified by the container. In this scenario, the servlet must process the request and generate as much of the response as the application requires.

But there are many cases in which some pre-processing of the request for servlets would be useful. In addition, it is sometimes useful to modify the response from a class of servlets. One example is encryption. A servlet, or a group of servlets in an application, may generate response data that is sensitive and should not go out over the network in clear-text form, especially when the connection has been made using a non-secure protocol such as HTTP. A filter can encrypt the responses. Of course, in this case the client must be able to decrypt the responses.

* Filters are basically used for intercepting and modifying requests and response from server. Consider a scenario where you want to check session from the every users request and if it is valid then only you want to let the user access the page. You can achieve this by checking sessions on all the servlet pages (or JSP pages) which users queries or you can do this by using Filter.
* A common scenario for a filter is one in which you want to apply pre-processing or post processing to requests or responses **for a group of servlets, not just a single servlet.** If you need to modify the request or response for just one servlet, there is no need to create a filter—just do what is required directly in the servlet itself.
* Note that filters are not servlets. They do not implement and override HttpServlet methods such as doGet() or doPost(). Rather, a filter is simply a Java class that implements the javax.servlet.Filter interface. The javax.servlet.Filter interface defines three methods:

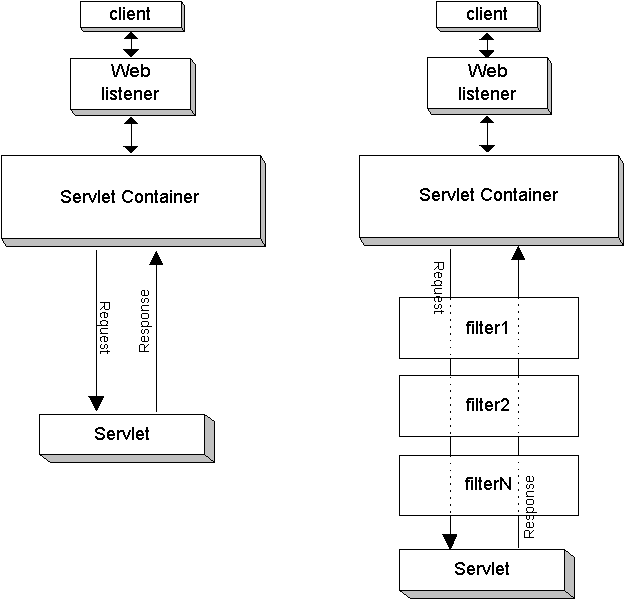
1. public void doFilter (ServletRequest request, ServletResponse response, FilterChain chain)
2. public FilterConfig getFilterConfig()
3. public void setFilterConfig (FilterConfig filterConfig)

It is the containers responsibility to create a javax.servlet.FilterConfig object and pass it to the filter during initialization. The javax.servlet.FilterConfig object can be used to

* Retrieve the filter name (as defined in the deployment descriptor)
* Retrieve the initial parameters (as defined in the deployment descriptor) and
* Get a reference to the ServletContext object the user is calling from.

How the Servlet Container Invokes Filters

* Following picture shows how the servlet container invokes filters. As shown, several filters (1, 2... N) have been configured in a chain to be invoked by the container before the servlet is called and after it has responded. The web.xml file specifies which servlets cause the container to invoke the filters.
* The order in which filters are invoked depends on the order in which they are configured in the web.xml file. The **first filter in web.xml is the first one invoked during the request, and the last filter in web.xml is the first one invoked during the response**. Note the reverse order during the response.



The order in which filters are invoked depends on the order in which they are configured in the web.xml file. The first filter in web.xml is the first one invoked during the request, and the last filter in web.xml is the first one invoked during the response. Note the reverse order during the response.

* Example :

Program

**public** **class** SessionFilter **implements** javax.servlet.Filter

{

**public** **void** init(FilterConfig fconfig)

**throws** ServletException

{

System.*out*.println("Filter Initialized");

System.*out*.println("Message : "+fconfig.getInitParameter("message"));

}

**public** **void** doFilter(ServletRequest sreq, ServletResponse sres, FilterChain chain)

**throws** IOException, ServletException

{

System.*out*.println("I am inside Filter");

chain.doFilter(sreq,sres);

}

**public** **void** destroy()

{

System.*out*.println("Filter Destroyed !!!");

}

} //End of Class

Web.xml

<filter>

<filter-name>validateSession</filter-name>

<filter-class>com.qspiders.lms.filters.SessionFilter</filter-class>

<init-param>

<param-name>message</param-name>

<param-value>A message for you!</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>validateSession</filter-name>

<!-- <url-pattern>/filter2.jsp</url-pattern> -->

<url-pattern>/\*</url-pattern>

</filter-mapping>

**Filtering of Forward or Include Targets**

In the OC4J 10.1.2 implementation, when a servlet is filtered, any servlets that are forwarded to or included from the filtered servlet are not filtered by default. You can change this behavior, however, through the following environment setting:

oracle.j2ee.filter.on.dispatch=true

This flag is set to false by default.

<http://viralpatel.net/blogs/tutorial-java-servlet-filter-example-using-eclipse-apache-tomcat/>

<http://www.roseindia.net/servlets/logging-servlet.shtml>

Listeners

Filters

<http://docs.oracle.com/cd/B14099_19/web.1012/b14017/filters.htm> --- > Refer this link for Liesteners as well

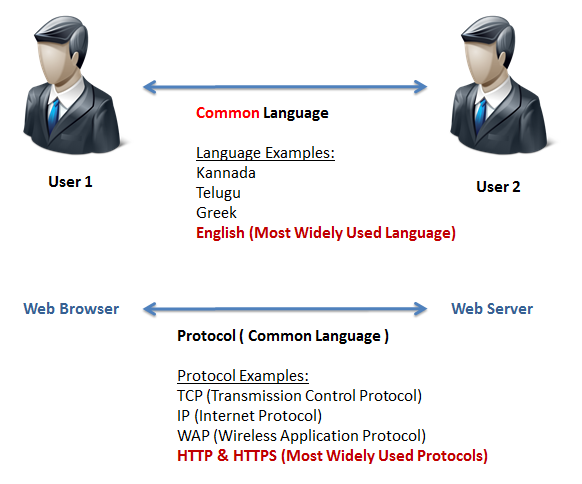
<http://www.javatips.net/blog/2011/12/servlet-listener-example>

<http://viralpatel.net/blogs/jsp-servlet-session-listener-tutorial-example-in-eclipse-tomcat/>

* Cookies were designed to be a reliable mechanism for websites to remember the state of the website or activity the user had taken in the past. This can include clicking particular buttons, logging in, or a record of which pages were visited by the user even months or years ago.
* The servlet sends cookies to the browser by using the HttpServletResponse.addCookie(javax.servlet.http.Cookie) method, which adds fields to HTTP response headers to send cookies to the browser, one at a time. The browser is expected to support 20 cookies for each Web server, 300 cookies total, and may limit cookie size to 4 KB each.
* The browser returns cookies to the servlet by adding fields to HTTP request headers. Cookies can be retrieved from a request by using the HttpServletRequest.getCookies() method. Several cookies might have the same name but different path attributes.
* User uses Browser to make request to server & every request contains the URL information. Also URL contains Domain information
* Browser is a software which is designed in such a way that, whenever user makes the request, it always looks out for Cookies information with in itself against the Domain. If found, it sends them to server along with the HTTP Request.
* The servlet is responsible for creating Cookies object & sends cookies to the browser along with **response object** & browser returns cookies to the servlet container by adding cookies to HTTP request & hence servlets gets the cookies from **request object**.

**How HTTP Works:**

* The browser sends a HTTP Request and server responds with a HTTP Response
* HTTP makes use of TCP/IP protocol
* Transmission Control Protocol (TCP) is responsible for making sure that data sent from one network node to another ends up as a complete data at destination, even though the data is split into multiple parts (data packets) when it’s sent
* Internet Protocol (IP) is the underlying protocol of TCP that routes the data packets from one network node to another
* HTTP is another network protocol that has web-specific features, but depends on TCP/IP to get the complete request & response



There are 216 -1 = 65, 536 ports available on a server (0 to 65, 535). Using one port / one application a webserver can run 65, 536 applications simultaneously. Although it is possible to run more than one application on the same port if the applications use different protocols

**Build Process:** It’s the process of compiling the source Java code & generating the standalone software artefact (WAR in case of Dynamic Web Project & JAR in case of Java Project). In real time we use tools such as Apache ANT, Apache Maven, etc. to do this job

**Deployment Process:** It’s the process of moving WAR file to the server deploy path.

**Web Archive (WAR)**

* It’s a Collection of source files (Java Classes, Servlets and JSP’s), Collection of resource files (XML, Properties, HTML, etc.) & Dependent JAR files
* WAR file represents a web application (like how .mp3 file represents a song & like .mpeg file represent a movie)