**Servlets**

The basic benefits of servlet are as follows:

1. better performance: because it creates a thread for each request not process.

2. Portability: because it uses java language.

3. Robust: Servlets are managed by JVM so no need to worry about momory leak, garbage collection etc.

4. Secure: because it uses java language..

Http is the protocol that allows web servers and browsers to exchange data over the web.

Servlet Life Cycle consists of three methods:

1. public void init(ServletConfig config) – This method is used by container to initialize the servlet, this method is invoked only once in the lifecycle of servlet.
2. public void service(ServletRequest request, ServletResponse response) – This method is called once for every request, container can’t invoke service() method until unless init() method is executed.
3. public void destroy() – This method is invoked once when servlet is unloaded from memory.

**Difference between GenericServlet and HttpServlet**

Generic Servlet:

1. GenericServlet class is direct subclass of Servlet interface.(GenericServlet is a abstract class)
2. Generic Servlet is protocol independent.It handles all types  of protocol  like http, smtp, ftp etc.
3. Generic Servlet only supports  service() method.It handles only simple request public void service(ServletRequest req,ServletResponse res ).
4. Generic Servlet only supports  service() method.

HttpServlet:

1. HttpServlet class is the direct subclass of Generic Servlet.(HttpServelt is a abstract class)
2. HttpServlet is protocol dependent. It handles only http protocol.
3. HttpServlet  supports public void service(ServletRequest req,ServletResponse res ) and protected void service(HttpServletRequest req,HttpServletResponse res).
4. HttpServlet supports also   doGet(),doPost(),doPut(),doDelete(),doHead(),doTrace(),doOptions()etc.

If you want your Servlet gives the functionality or you want provide functionality then you can extends GenericServlet and you can provide the same.

**Web.xml**

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app

PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app>

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>controlServlet</servlet-name>

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

</servlet-mapping>

</web-app>

**Servlet Init Parameters**

You can pass parameters to a servlet from the web.xml file. The init parameters of a servlet can only be accessed by that servlet. Here is how you configure them in the web.xml file:

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>

**<init-param>**

**<param-name>myParam</param-name>**

**<param-value>paramValue</param-value>**

**</init-param>**

</servlet>

Here is how you read the init parameters from inside your servlet - in the servlets init() method:

public class SimpleServlet extends GenericServlet {

protected String myParam = null;

public void init(ServletConfig servletConfig) throws ServletException{

**this.myParam = servletConfig.getInitParameter("myParam");**

}

## Servlet Load-on-Startup

The <servlet> element has a subelement called <load-on-startup> which you can use to control when the servlet container should load the servlet. If you do not specify a <load-on-startup> element, the servlet container will typically load your servlet when the first request arrives for it.

By setting a <load-on-startup> element, you can tell the servlet container to load the servlet as soon as the servlet container starts. Remember, the servlets init() method is called when the servlet is loaded.

Here is an example <load-on-startup> configuration:

<servlet>

<servlet-name>controlServlet</servlet-name>

<servlet-class>com.jenkov.webui.ControlServlet</servlet-class>

<init-param><param-name>container.script.static</param-name>

<param-value>/WEB-INF/container.script</param-value>

</init-param>

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

</servlet>

The number inside the **<load-on-startup>1</load-on-startup>** element tells the servlet container in what sequence the servlets should be loaded. The lower numbers are loaded first. If the value is negative, or unspecified, the servlet container can load the servlet at any time.

## Context Parameters

You can also set some context parameters which can be read from all servlets in your application. Here is how you configure a context parameter:

<context-param>

<param-name>myParam</param-name>

<param-value>the value</param-value>

</context-param>

Here is how you access the parameter from inside an HttpServlet subclass:

String myContextParam =

request.getSession()

.getServletContext()

.getInitParameter("myParam");

**ServletConfig and ServletContext**

**ServletConfig**

* ServletConfig available in javax.servlet.\*; package
* ServletConfig object is one per servlet class
* Object of ServletConfig will be created during initialization process of the servlet
* This Config object is public to a particular servlet only
* *Scope*: As long as a servlet is executing, ServletConfig object will be available, it will be destroyed once the servlet execution is completed.
* We should give request explicitly, in order to create ServletConfig object for the first time
* In web.xml – <*init-param*> tag will be appear under <*servlet-class*> tag
* To get the config init parameters ServletConfig conf = getServletConfig(); and then String s1= conf.getInitParametrs(“initparmName”);

**ServletContext**

* ServletContext available in javax.servlet.\*; package
* ServletContext object is global to entire web application
* Object of ServletContext will be created at the time of web application deployment
* *Scope*: As long as web application is executing, ServletContext object will be available, and it will be destroyed once the application is removed from the server.
* ServletContext object will be available even before giving the first request
* In web.xml – <*context-param*> tag will be appear under <*web-app*> tag
* Accessing **context init parameter** in a servlet:  
  getServlet**Context**().getInitParameter(“email”);

So finally…….

No. of web applications  =  That many number of ServletContext objects [ 1 per web application ]  
No. of servlet classes = That many number of ServletConfig objects

**Web Server**

Web server contains only web or servlet container. It can be used for servlet, jsp, struts, jsf etc. It can't be used for EJB.

Example of Web Servers are: Apache Tomcat and Resin.

**Application Server**

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc.

Example of Application Servers are:

JBoss Open-source server from JBoss community.

Glassfish provided by Sun Microsystem. Now acquired by Oracle.

Weblogic provided by Oracle. It more secured.

Websphere provided by IBM.

**Advantage of load-on-startup element**

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

**Session Tracking:**

session simply means a particular interval of time.

Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet. Like if we want to know about the user information throughout the session like user name, last time visited, number of times visited etc to fetch all these info we need to implement the session management in http servlet because it is a stateless protocol it will not maintain any state.

There are four techniques used in Session tracking:

Cookies

Hidden Form Field

URL Rewriting

HttpSession

**Cookies:**

Cookies in Servlet

A cookie is a small piece of information that is persisted between the multiple client requests. A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, we will get the cookies from the request Cookies cokkie= request.getCokkies(); and we will check the cookie info for further action. If we want delete cookie then set the cookie age as 0 then it will be deleted from the client browser (Cookie.setMaxAge(0) ).

Adding the cookies in response from the servelet.

protected void doPost(HttpServletRequest request,

            HttpServletResponse response) throws ServletException, IOException {

        // get request parameters for userID and password

        String user = request.getParameter("user");

        String pwd = request.getParameter("pwd");

        if(userID.equals(user) && password.equals(pwd)){

            Cookie loginCookie = new Cookie("user",user);

            //setting cookie to expiry in 30 mins

            loginCookie.setMaxAge(30\*60);

            response.addCookie(loginCookie);

            response.sendRedirect("LoginSuccess.jsp");

        }else{

            RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");

            PrintWriter out= response.getWriter();

            out.println("<font color=red>Either user name or password is wrong.</font>");

            rd.include(request, response);

        }

    }

To delete the cookies from the browser

 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

        response.setContentType("text/html");

        Cookie loginCookie = null;

        Cookie[] cookies = request.getCookies();

        if(cookies != null){

        for(Cookie cookie : cookies){

            if(cookie.getName().equals("user")){

                loginCookie = cookie;

                break;

            }

        }

        }

        if(loginCookie != null){

            loginCookie.setMaxAge(0);

            response.addCookie(loginCookie);

        }

        response.sendRedirect("login.html");

    }

**Hidden Form Field**

In case of Hidden Form Field a hidden (invisible) textfield is used for maintaining the state of an user. In such case, we store the information in the hidden field and get it from another servlet. This approach is better if we have to submit form in all the pages and we don't want to depend on the browser. It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Advantage of Hidden Form Field

It will always work whether cookie is disabled or not.

Disadvantage of Hidden Form Field:

* 1. It is maintained at server side.
  2. Extra form submission is required on each pages.
  3. Only textual information can be used.

**Session Management using Http Session**

servlet provides HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

The servlet container uses this interface to create a session between an HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. You would get HttpSession object by calling the public method **getSession()** of HttpServletRequest, as below:

HttpSession session = request.getSession();

Servlet API provides Session management through HttpSession interface. We can get session from HttpServletRequest object using following methods. HttpSession allows us to set objects as attributes that can be retrieved in future requests.

1. **HttpSession getSession()** – This method always returns a HttpSession object. It returns the session object attached with the request, if the request has no session attached, then it creates a new session and return it.
2. **HttpSession getSession(boolean flag)** – This method returns HttpSession object if request has session else it returns null.

Some of the important methods of HttpSession are:

1. **String getId()** – Returns a string containing the unique identifier assigned to this session.
2. **Object getAttribute(String name)** – Returns the object bound with the specified name in this session, or null if no object is bound under the name. Some other methods to work with Session attributes are getAttributeNames(), removeAttribute(String name) and setAttribute(String name, Object value).
3. **long getCreationTime()** – Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT. We can get last accessed time withgetLastAccessedTime() method.
4. setMaxInactiveInterval(int interval) – Specifies the time, in seconds, between client requests before the servlet container will invalidate this session. We can get session timeout value fromgetMaxInactiveInterval() method.
5. **ServletContext getServletContext()** – Returns ServletContext object for the application.
6. **boolean isNew()** – Returns true if the client does not yet know about the session or if the client chooses not to join the session.
7. **void invalidate()** – Invalidates this session then unbinds any objects bound to it.

The real power of the session object is found in the application data. Though the documentation doesn't say so, somewhere up the inheritance hierarchy there must be a hashtable, because the session object supports the methods to push and pop objects on and off the session. These methods, putValue(String name, Object value) and getValue(java.lang.String name) allow you to get and set data in the session that you can use for any variety of things.

Let's say that you are building a site to allow users to buy books online (what a concept!). One thing you will need is a shopping basket, which will track the items a user purchases.

You can create a class such as this:

import java.util.\*;

import java.io.Serializable;

public class ShoppingCart implements Serializable

{

private int accountNum;

private Vector Items;

public ShoppingCart()

{

Items = new Vector();

}

public int getAccountNum()

{return this.accountNum;

}

public void setAccountNum(int accountNum)

{

this.accountNum = accountNum;

}

public Iterator getItems()

{

return this.Items.iterator();

}

public boolean addItem(int item)

{

return Items.add(new Integer(item));

}

public String toString()

{

return "ShoppingCart for " + accountNum;

}

}

This is a simple example of a shopping cart that will allow you to set an account number, and then add items to the cart. Our purpose here is not to learn how to build a shopping cart, but how to manage sessions, so we will leave this design alone for now.

Your servlet might look something like this, then:  
(Note: the penultimate line of code has been split so it fits across the page you're currently viewing.)

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.\*;

public class BookStoreServlet extends HttpServlet

{

/\*\* This method is the first method executed when the servlet loads.

\* @param HttpServletRequest

\* @param HttpServletResponse

\* @return void

\*/

public synchronized void init (ServletConfig config)

{

super.init();

}

/\*\* doGet method

\*/

public void doGet (HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException

{

// Send out first page:

res.setContentType("text/html");

PrintWriter out = res.getWriter();

out.println("<HTML<");

out.println("<BODY>");

out.println("<CENTER>");

out.println("<H1>Welcome to WholeLottaBooks.com!</H1>");

out.println("Would you like to begin shopping?");

out.println("<FORM ACTION='/servlet/BookStoreServlet' METHOD='POST'>");

out.println("<INPUT TYPE='submit' VALUE='You bet!'>");

out.println("</FORM>");

out.println("</CENTER>");

out.println("</BODY>");

out.println("</HTML>");

}

/\*\*

\* doPost method

\*/

public void doPost (HttpServletRequest req, HttpServletResponse res)

throws ServletException, IOException

{

StringBuffer HTML = new StringBuffer();

res.setContentType("text/html");

PrintWriter out = res.getWriter();

ShoppingCart sc;

HttpSession session = req.getSession(true);

if (session.getValue("Cart") == null)

// User has no shopping cart, create one.

{

sc= new ShoppingCart();

session.putValue("Cart",sc);

int acct = new Random().nextInt();

if (acct < 0)

acct = acct \* -1;

// Cast to type of ShoppingCart, since HttpSession only deals in Object.

((ShoppingCart)session.getValue("Cart")).setAccountNum(acct);

}

out.println("Your account number is " + ((ShoppingCart)se

ssion.getValue("Cart")).getAccountNum());

}

} //end of servlet

This will put out a welcome page that asks the user whether they want to begin shopping. If they click the "You bet" button, the doPost method executes. The doPost will check to see if they have a shopping cart. If not, one is created, and their account number is assigned to it. Again, the details of assigning account numbers, making sure that concurrency is taken care of - all of that is necessary to implementing such a system, but is outside the scope of this discussion.

After we have a shopping cart, the user will begin adding items to the cart. How this is handled is up to you as a developer. Whatever method you choose, however, you will probably be passing these to your servlet through the HttpRequest object.

We expand the above code to process these parameters as follows:

if (session.getValue("Cart") == null)

// User has no shopping cart, create one.

{

sc= new ShoppingCart();

session.putValue("Cart",sc);

int acct = new Random().nextInt();

if (acct < 0)

acct = acct \* -1;

// Cast to type of ShoppingCart, since HttpSession only deals in Object.

((ShoppingCart)session.getValue("Cart")).setAccountNum(acct);

}

else // Shopping cart exists, see if user wants to add itesms

{

Enumeration enum = req.getParameterNames();

while (enum.hasMoreElements())

{

String parmName = (String)enum.nextElement();

if (parmName.equals("stockNum"))

{

String parmValue = req.getParameter(parmName);

((ShoppingCart)session.getValue("Cart")).addItem(parmValue);

}

}

}

We have added an else branch to the if statement to handle the case where the user already has a shopping cart, and wants to add items to the cart. GetParameterNames will return an Enumeration of the paramter names coming in the the HttpRequest object. If the parameter name is "stockNum" we know the user wants to add an item to their cart. We are not concerned with book titles or publishers, as everything will be keyed off the [stock](http://webdevelopersjournal.com/articles/httpsession_object.html) number. We then extract the parameter value, and add it to the Shopping Cart by calling the addItem method. Note that to do this we first need to get a reference to the ShoppingCart object by calling getValue on the session object, and casting it to the appropriate type (ShoppingCart).

Using this mechanism, you can add any kind of object to your session object. This provides you as a developer with a lot of flexibility to implement your servlet in any way you choose.

A word about the safety of the doPost method of your servlet. Servlets are capable of handling more than one request at a time. However, there could be ramifications for the user based on the threading model that you choose to implement. The default model is to allow multiple threads to access the servlet. There is only one instance of the servlet running in the Web server, which serves all users. However, each user has his own copy of the doPost method, and everything goes swimmingly. The caveat to this is if you are accessing shared resources from the doPost method, you will need to account for the safety of that. For example, if you have declared a variable that is a class level or static variable, and the access to it needs to be atomic, you will need to synchronize the code that accesses the resource. The best way around this is to simply avoid using class-level or static variables that need to have atomic access. The other threading model the servlet API supports is the SingleThreadModel. Under this model, the Web server maintains a pool of servlet instances, rather than just one servlet. Each user gets a different instance of the servlet. The only thing needed for implementing this model is to have the servlet implement the interface javax.servlet.SingleThreadModel.

All this interface does is act as a marker - there is nothing for you as a developer to implement. The servlet engine will maintain a pool of servlet instances, and serve up a free instance to a user of your site. This method will still not guarantee that shared resources such as static class variables or classes outside the scope of the servlet will be safely accessed. Again, you will need to synchronize access to these resources to ensure thread safety.

My preference is for the default threading model. You simply need to be careful about how you implement your servlet. This is not too difficult however. Just stay away from class variables and synchronize access to shared resources and you will be fine.

Let's return to our WholeLottaBooks.com servlet. Another feature of the HttpSession object which can work in your favor is that the session relates to the server, not the servlet. For example, if WholeLottaBooks becomes wildly successful, and they decide to branch out into selling music, a separate servlet to handle music purchases could access the same shopping cart as the books servlet! This is very convenient to your users. Regardless of what they are purchasing, it all goes into the same cart.

This brief discussion of the HttpSession object gives you an idea of the flexibility and power available to you as a servlet developer. Like most coding exercises however, if you don't use it, you lose it. So experiment with a small servlet of your own and see for yourself what the session can do for you.

**ThreadSafe:**

**Why we need to implement Single Thread model in case of Servlet.**

Ans: In J2EE we can implement our servlet on two different ways either by using:

1. Single Thread Model

2. Multithread Model

Depending upon our scenario, if we have implemented single thread means only one instance is going handle one request at a time no two thread will concurrently execute service method of servlet.

**Example** in banking account where sensitive data is handle mostly this scenario was used this interface is deprecated in Servlet API version 2.4.

As the name signifies multi thread means a servlet is capable to handle multiple requests at same time.

Read more: <http://javarevisited.blogspot.com/2011/09/servlet-interview-questions-answers.html#ixzz3eNAxs5cF>

**RequestDispatcher**

**RequestDispatcher** interface provides two important methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| voidforward(ServletRequest request, ServletResponse response) | forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server |
| voidinclude(ServletRequest request, ServletResponse response) | includes the content of a resource (servlet, JSP page, HTML file) in the response |

getRequestDispatcher() method of **ServletRequest** returns the object of **RequestDispatcher**.

RequestDispatcher *rs* = request.**getRequestDispatcher**("hello.html");

rs.**forward**(request,response);



**OR**

RequestDispatcher *rs* = request.**getRequestDispatcher**("hello.html");

rs.**include**(request,response);



**SendRedirect():**

forward

Control can be forward to resources available within the server from where the call is made. This transfer of control is done by the container internally and browser / client is not involved. This is the major difference between forward and sendRedirect. When the forward is done, the original request and response objects are transfered along with additional parameters if needed.

redirect

Control can be redirect to resources to different servers or domains. This transfer of control task is delegated to the browser by the container. That is, the redirect sends a header back to the browser / client. This header contains the resource url to be redirected by the browser. Then the browser initiates a new request to the given url. Since it is a new request, the old request and response object is lost

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

For example, sendRedirect can transfer control from http://javapapers.com to http://anydomain.com but forward cannot do this.

‘session’ is not lost in both forward and redirect.

To feel the difference between forward and sendRedirect visually see the address bar of your browser,  
in forward, you will not see the forwarded address (since the browser is not involved)  
in redirect, you can see the redirected address.

**When can we use forward and when can we use sendRedirect?**

Technical scenario: redirect should be used

1. If you need to transfer control to different domain
2. To achieve separation of task.

For example, database update and data display can be separated by redirect. Do the PaymentProcess and then redirect to displayPaymentInfo. If the client refreshes the browser only the displayPaymentInfo will be done again and PyamenProcess will not be repeated. But if you use forward in this scenario, both PaymentProcess and displayPaymentInfo will be re-executed sequentially, which may result in incosistent data.

For other than the above two scenarios, forward is efficient to use since it is faster than sendRedirect.

**Example for forward and sendRedirect based on real world**

Consider the real world scenario, the milk man comes and asks for monthly payment to you in your house. Here house is the container and you are a resource existing in the container. Milk man is the client or browser.

He asks for the monthly payment to you, this is the request made by the browser to resource A. If you go inside your house and ask your mother (another resource B inside the same container) for the cash and come back and deliver to milkman this is called forward.

If you ask the milkman to speak himself to your mother inside your house or you ask the milkman to speak to your father who is in his office (different domain) then this is called redirect.

**When Container calls destroy method.**

**Exception Handling in servlets.**

**Filters:**

A filter is an object that is invoked at the preprocessing and postprocessing of a request.

It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc. The servlet filter is pluggable, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet. So maintenance cost will be less.

we learned how we can manage session in web application and if we want to make sure that a resource is accessible only when user session is valid, we can achieve this using servlet session attributes. The approach is simple but if we have a lot of servlets and jsps, then it will become hard to maintain because of redundant code. If we want to change the attribute name in future, we will have to change all the places where we have session authentication.

That’s why we have servlet filter. Servlet Filters are **pluggable** java components that we can use to intercept and process requests *before* they are sent to servlets and response *after* servlet code is finished and before container sends the response back to the client.

Some common tasks that we can do with filters are:

* Logging request parameters to log files.
* Authentication and autherization of request for resources.
* Formatting of request body or header before sending it to servlet.
* Compressing the response data sent to the client.
* Alter response by adding some cookies, header information etc.

As I mentioned earlier, **servlet filters are pluggable** and configured in deployment descriptor (web.xml) file. Servlets and filters both are unaware of each other and we can add or remove a filter just by editing web.xml.

We can have multiple filters for a single resource and we can create a chain of filters for a single resource in web.xml. We can create a Servlet Filter by implementing javax.servlet.Filter interface.

Filter interface is similar to Servlet interface and we need to implement it to create our own servlet filter. Filter interface contains lifecycle methods of a Filter and it’s managed by servlet container.

Filter interface lifecycle methods are:

1. **void init(FilterConfig paramFilterConfig)** – When container initializes the Filter, this is the method that gets invoked. This method is called only once in the lifecycle of filter and we should initialize any resources in this method. **FilterConfig** is used by container to provide init parameters and servlet context object to the Filter. We can throw ServletException in this method.
2. **doFilter(ServletRequest paramServletRequest, ServletResponse paramServletResponse, FilterChain paramFilterChain)** – This is the method invoked every time by container when it has to apply filter to a resource. Container provides request and response object references to filter as argument. **FilterChain** is used to invoke the next filter in the chain. This is a great example of[**Chain of Responsibility Pattern**](http://www.journaldev.com/1617/chain-of-responsibility-design-pattern-in-java-example-tutorial).
3. **void destroy()** – When container offloads the Filter instance, it invokes the destroy() method. This is the method where we can close any resources opened by filter. This method is called only once in the lifetime of filter.
4. We can declare a filter in web.xml like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <filter>    <filter-name>RequestLoggingFilter</filter-name> <!-- mandatory -->    <filter-class>com.journaldev.servlet.filters.RequestLoggingFilter</filter-class> <!-- mandatory -->    <init-param> <!-- optional -->    <param-name>test</param-name>    <param-value>testValue</param-value>    </init-param>  </filter> |

1. We can map a Filter to servlet classes or url-patterns like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <filter-mapping>    <filter-name>RequestLoggingFilter</filter-name> <!-- mandatory -->    <url-pattern>/\*</url-pattern> <!-- either url-pattern or servlet-name is mandatory -->    <servlet-name>LoginServlet</servlet-name>    <dispatcher>REQUEST</dispatcher>  </filter-mapping> |

1. Note: While creating the filter chain for a servlet, container first processes the url-patterns and then servlet-names, so if you have to make sure that filters are getting executed in a particular order, give extra attention while defining the filter mapping.
2. Servlet Filters are generally used for client requests but sometimes we want to apply filters with[**RequestDispatcher**](http://www.journaldev.com/1877/java-servlet-tutorial-with-examples-for-beginners#requestdispatcher-interface) also, we can use dispatcher element in this case, the possible values are REQUEST, FORWARD, INCLUDE, ERROR and ASYNC. If no dispatcher is defined then it’s applied only to client requests.

Filters will have init (FilterConfig fconfig), doFilet( req, res, FilterChain fc ) and destroy() method same like Servlets methods. In doFilter method we will have our business logic.

|  |
| --- |
| @WebFilter("/AuthenticationFilter")  public class AuthenticationFilter implements Filter {        private ServletContext context;        public void init(FilterConfig fConfig) throws ServletException {          this.context = fConfig.getServletContext();          this.context.log("AuthenticationFilter initialized");      }        public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {            HttpServletRequest req = (HttpServletRequest) request;          HttpServletResponse res = (HttpServletResponse) response;            String uri = req.getRequestURI();          this.context.log("Requested Resource::"+uri);            HttpSession session = req.getSession(false);            if(session == null && !(uri.endsWith("html") || uri.endsWith("LoginServlet"))){              this.context.log("Unauthorized access request");              res.sendRedirect("login.html");          }else{              // pass the request along the filter chain              chain.doFilter(request, response);          }          }            public void destroy() {          //close any resources here      }    } |

1. Notice that we are not authenticating any HTML page or LoginServlet. Now we will configure these filters mapping in the web.xml file.

|  |  |
| --- | --- |
| web.xml | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" version="3.0">    <display-name>ServletFilterExample</display-name>    <welcome-file-list>      <welcome-file>login.html</welcome-file>    </welcome-file-list>      <filter>      <filter-name>RequestLoggingFilter</filter-name>      <filter-class>com.journaldev.servlet.filters.RequestLoggingFilter</filter-class>    </filter>    <filter>      <filter-name>AuthenticationFilter</filter-name>      <filter-class>com.journaldev.servlet.filters.AuthenticationFilter</filter-class>    </filter>      <filter-mapping>      <filter-name>RequestLoggingFilter</filter-name>      <url-pattern>/\*</url-pattern>      <dispatcher>REQUEST</dispatcher>    </filter-mapping>    <filter-mapping>      <filter-name>AuthenticationFilter</filter-name>      <url-pattern>/\*</url-pattern>    </filter-mapping>  </web-app> |

### Difference between SendRedirect and Forward

Now let’s see some difference between these two method of servlet API in tabular format.

|  |  |
| --- | --- |
| **Forward()** | **SendRediret()** |
| When we use forward method request is transfer to other resource within the same server for further processing. | In case of sendRedirect request is transfer to another resource to different domain or different server for futher processing. |
| In case of forward Web container handle all process internally and client or browser is not involved. | When you use SendRedirect container transfers the request to client or browser so url given inside the **sendRedirect**method is visible as a new request to the client. |
| When forward is called on **requestdispather**object we pass request and response object so our old request object is present on new resource which is going to process our request | In case of SendRedirect call old request and response object is lost because it’s treated as new request by the browser. |
| Visually we are not able to see the forwarded address, its is transparent | In address bar we are able to see the new redirected address it’s not transparent. |
| Using forward () method is faster then send redirect. | SendRedirect is slower because one extra round trip is required beasue completely new request is created and old request object is lost.Two browser request requird. |
| When we redirect using forward and we want to use same data in new resource we can use request.setAttribute () as we have request object available. | But in sendRedirect if we want to use we have to store the data in session or pass along with the URL. |