**Introduction to Servlets**

Servlet is a java program that runs inside JVM on the web server. It is used for developing dynamic web applications.  
Before we proceed further let understand what is **dynamic web application?** A web application can be described as collection of web pages (e.g. a website) and when we call it dynamic, it simply means that the web pages are not same for all the users, web pages would be generated on server side based on the request made by client(user’s browser).

The main **difference between static and dynamic web page** is that static page as name suggests remains same for all users however a dynamic web page changes based on the request from client (user’s browser). For example, consider a web application that shows you two input fields & and add button and when you enter two numbers and click add, it shows you another web page that has the result of addition of two numbers, this web application is dynamic in nature as the second web page that shows you the result changes based on the user input, it is not static for all users.

However you can very well say that what a servlet does can be done by CGI (Common Gateway Interface), well it true but here is the thing – **CGI has several limitations such as performance, scalability, reusability etc. that a servlet doesn’t have**. I am not going to discuss CGI in detail but I am going to tell you, how a servlet is better than CGI.

### Limitations of CGI

Server has to create a new CGI process for every client request. For example, If 100 users are accessing the web application, then the server has to create 100 CGI processes to handle the request made by them. Since a server has limited resources, creating new process every time for a new request is not a viable option, this imposed the limitation on server, due to that the server cannot handle more than a specified number of users at the same time.

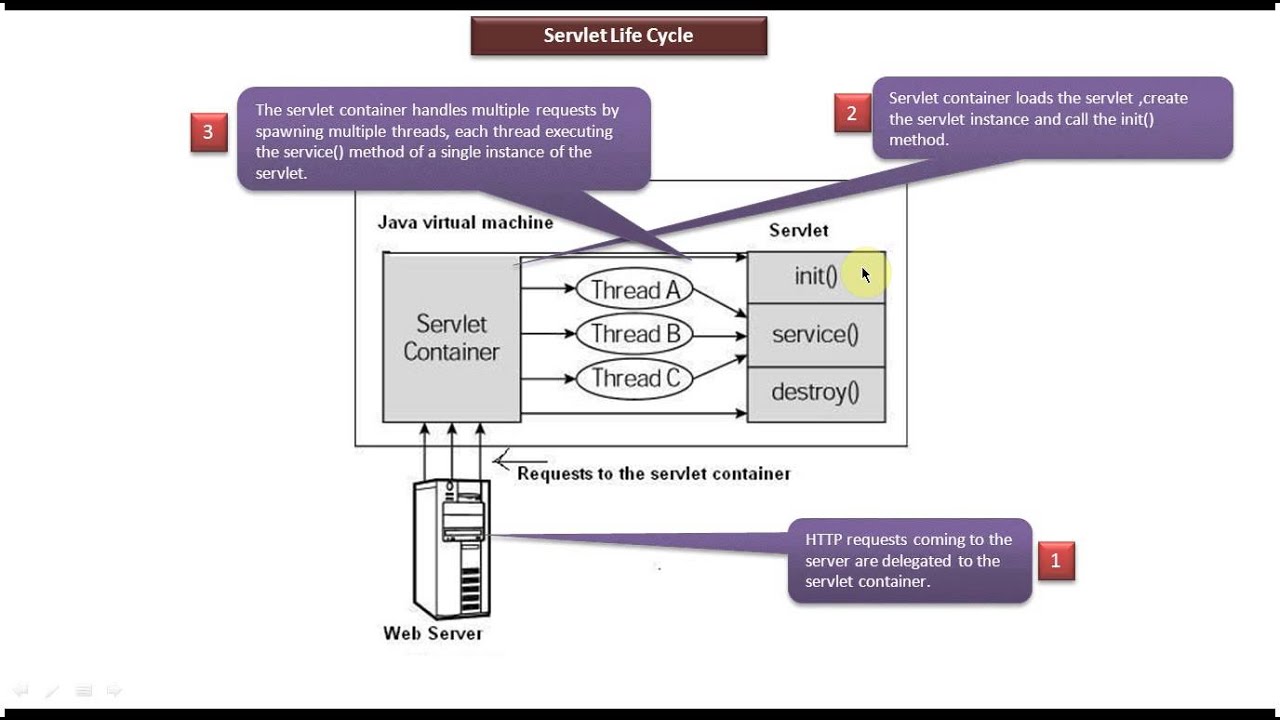
### How Servlet is better than CGI

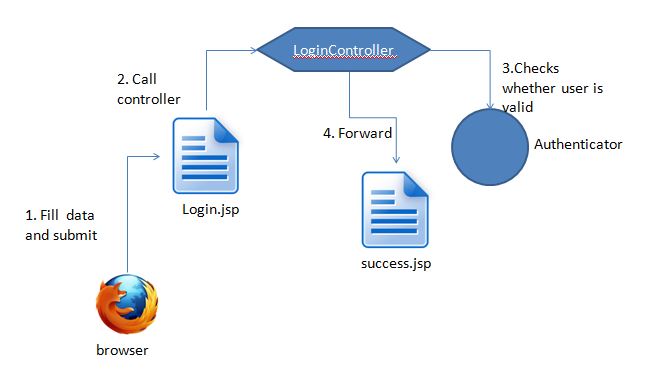
CGI programs are handled by a new process every time a new request has been made. Unlike CGI, the servlet programs are handled by separate threads that can run concurrently more efficiently.

CGI program can be written in any programming language that makes it mostly platform dependent as not all programming languages are platform independent. Servlet only uses Java as programming language that makes it platform independent and portable. Another benefit of using java is that the servlet can take advantage of the object oriented programming features of java.

#### How Servlet Works

As mentioned above that concurrent requests to the server are handled by threads, here is the graphical representation of the same –





## Features of Servlet

Now that we have understood what is a servlet and for what purpose it is being used. Let’s proceed further and discuss its main features.

**1. Portable:**  
As I mentioned above that Servlet uses Java as a programming language, Since java is platform independent, the same holds true for servlets. For example, you can create a servlet on Windows operating system that users GlassFish as web server and later run it on any other operating system like Unix, Linux with Apache tomcat web server, this feature makes servlet portable and this is the main advantage servlet has over CGI.

**2. Efficient and scalable:**  
Once a servlet is deployed and loaded on a web server, it can instantly start fulfilling request of clients. The web server invokes servlet using a lightweight thread so multiple client requests can be fulling by servlet at the same time using the multithreading feature of Java. Compared to CGI where the server has to initiate a new process for every client request, the servlet is truly efficient and scalable.

**3. Robust:**  
By inheriting the top features of Java (such as Garbage collection, Exception handling, Java Security Manager etc.) the servlet is less prone to memory management issues and memory leaks. This makes development of web application in servlets secure and less error prone.

# Servlet API

You need to use Servlet API to create servlets. There are two packages that you must remember while using API, the javax.servlet package that contains the classes to support generic servlet (protocol-independent servlet) and the javax.servlet.http package that contains classes to support http servlet. You may be wondering what is generic and http servlet, I have explained them later in this post.

**Let’s see the hierarchy of packages:**

java.lang.Object

|\_extended byjavax.servlet.GenericServlet

|\_extended byjavax.servlet.http.HttpServlet

Every Servlet must implement the java.servlet.Servlet interface, you can do it by extending one of the following two classes: javax.servlet.GenericServlet or javax.servlet.http.HttpServlet. The first one is for protocol independent Servlet and the second one for http Servlet.

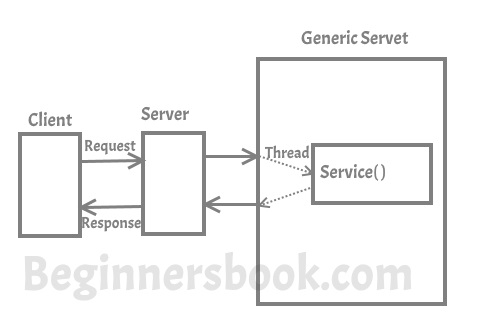
## Generic Servlet

As mentioned above, if you are creating a Generic Servlet then you must extend javax.servlet.GenericServlet class. GenericServlet class has an abstract service() method. Which means the subclass of GenericServlet should always override the service() method.  
**Signature of service() method:**

publicabstractvoid service(ServletRequest request,ServletResponse response)

throwsServletException, java.io.IOException

The service() method accepts two arguments ServletRequest object and ServletResponse object. The request object tells the servlet about the request made by client while the response object is used to return a response back to the client.

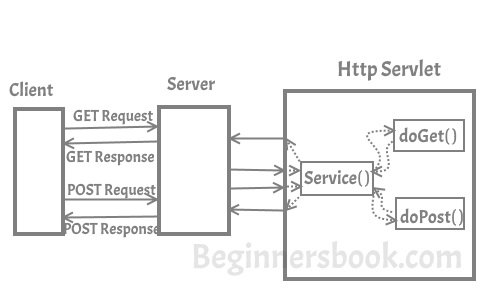


## HTTP Servlet

If you creating Http Servlet you must extend javax.servlet.http.HttpServlet class, which is an abstract class. Unlike Generic Servlet, the HTTP Servlet doesn’t override the service() method. Instead it overrides one or more of the following methods. It must override at least one method from the list below:

* **doGet()** – This method is called by servlet service method to handle the HTTP GET request from client. The Get method is used for getting information from the server
* **doPost()** – Used for posting information to the Server
* **doPut()** – This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server
* **doDelete()** – allows a client to delete a document, webpage or information from the server
* **init() and destroy()** – Used for managing resources that are held for the life of the servlet
* **getServletInfo()** – Returns information about the servlet, such as author, version, and copyright.

**In Http Servlet there is no need to override the service() method** as this method dispatches the Http Requests to the correct method handler, for example if it receives HTTP GET Request it dispatches the request to the doGet() method.



#### Interfaces in javax.servlet package

* Servlet
* ServletRequest
* ServletResponse
* ServletConfig
* ServletContext
* SingleThreadModel
* RequestDispatcher
* ServletRequestListener
* ServletRequestAttributeListener
* ServletContextListener
* ServletContextAttributeListener
* Filter
* FilterConfig
* FilterChain

#### Classes in javax.servlet package

* GenericServlet
* ServletInputStream
* ServletOutputStream
* ServletException
* ServletRequestWrapper
* ServletRequestEvent
* ServletResponseWrapper
* ServletContextEvent
* ServletRequestAttributeEvent
* ServletContextAttributeEvent
* UnavailableException

#### Interfaces in javax.servlet.http package

* HttpSession
* HttpServletRequest
* HttpServletResponse
* HttpSessionAttributeListener
* HttpSessionListener
* HttpSessionBindingListener
* HttpSessionActivationListener
* HttpSessionContext

#### Classes in javax.servlet.http package

* HttpServlet
* Cookie
* HttpSessionEvent
* HttpSessionBindingEvent
* HttpServletRequestWrapper
* HttpServletResponseWrapper
* HttpUtils

# Servlet Interface explained with Example

In any Servlet you must implement the Servlet interface directly or indirectly (indirectly implementation means extending those classes that implements Servlet interface, These classes are GenericServlet and HttpServlet).

If you are creating protocol dependent servlet such as http servlet then you should extend HttpServlet class else for protocol independent Servlet you extend GenericServlet class.

In short you have 3 ways to create a servlet:  
1) By extending HttpServlet class  
2) By extending GenericServlet class  
3) By implementing Servlet interface

Note: However you should always prefer the first way of creating servlet i.e. by extending HttpServlet class.

### Servlet Interface methods

Here is the list of methods available in Servlet interface.  
1) **void destroy()**: This method is called by Servlet container at the end of servlet life cycle. Unlike service() method that gets called multiple times during life cycle, this method is called only once by Servlet container during the complete life cycle. Once destroy() method is called the servlet container does not call the service() method for that servlet.

2) **void init(ServletConfig config)**: When Servlet container starts up (that happens when the web server starts up) it loads all the servlets and instantiates them. After this init() method gets called for each instantiated servlet, this method initializes the servlet.

3) **void service(ServletRequest req, ServletResponse res)**: This is the only method that is called multiple times during servlet life cycle. This methods serves the client request, it is called every time the server receives a request.

4) **ServletConfig getServletConfig()**: Returns a ServletConfig object, which contains initialization and startup parameters for this servlet.

5) **java.lang.String getServletInfo()**: Returns information about the servlet, such as author, version, and copyright.

## Example:

In this example we have created a servlet class by extending Servlet interface.

index.html

<ahref="welcome">Click here to call the servlet</a>

DemoServlet.java

import java.io.\*;

import javax.servlet.\*;

publicclassDemoServletimplementsServlet{

ServletConfig config=null;

publicvoid init(ServletConfig config){

this.config=config;

System.out.println("Initialization complete");

}

publicvoid service(ServletRequest req,ServletResponse res)

throwsIOException,ServletException{

res.setContentType("text/html");

PrintWriter pwriter=res.getWriter();

pwriter.print("<html>");

pwriter.print("<body>");

pwriter.print("<h1>Servlet Example Program</h1>");

pwriter.print("</body>");

pwriter.print("</html>");

}

publicvoid destroy(){

System.out.println("servlet life cycle finished");

}

publicServletConfig getServletConfig(){

return config;

}

publicString getServletInfo(){

return"A Demo program written by me";

}

}

web.xml

<web-app>

<servlet>

<servlet-name>booksNbooks</servlet-name>

<servlet-class>DemoServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>booksNbooks</servlet-name>

<url-pattern>/welcome</url-pattern>

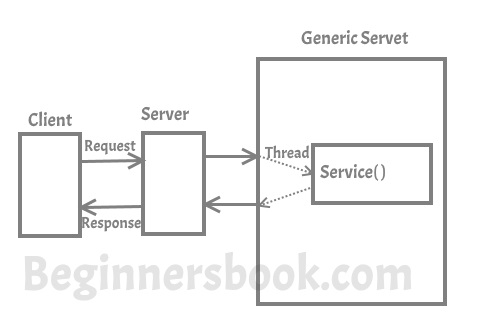
</servlet-mapping>

</web-app>

# GenericServlet Class with example

A generic servlet is a protocol independent Servlet that should always override the service() method to handle the client request. The service() method accepts two arguments ServletRequest object and ServletResponse object. The request object tells the servlet about the request made by client while the response object is used to return a response back to the client.

### How Generic Servlet works?



### Hierarchy of Generic Servlet

java.lang.Object

|\_extended byjavax.servlet.GenericServlet

GenericServlet is an abstract class and it has only one abstract method, which is service(). That’s why when we create Generic Servlet by extending GenericServlet class, we must override service() method.

**Pros of using Generic Servlet:**  
1. Generic Servlet is easier to write  
2. Has simple lifecycle methods  
3. To write Generic Servlet you just need to extend javax.servlet.GenericServlet and override the service() method (check the example below).

**Cons of using Generic Servlet:**  
Working with Generic Servlet is not that easy because we don’t have convenience methods such as doGet(), doPost(), doHead() etc in Generic Servlet that we can use in Http Servlet.  
In Http Servlet we need to override particular convenience method for particular request, for example if you need to get information then override doGet(), if you want to send information to server override doPost(). However in Generic Servlet we only override service() method for each type of request which is cumbersome.

I would always recommend you to use HttpServlet instead of the GenericServlet. HttpServlet is easier to work with, and has more methods to work with than GenericServlet.

### Example of GenericServlet

Using Netbeans for this example. Create New “Dynamic Web Project” from the file menu.

**index.html**  
We are creating an html file that would call the servlet once we click on the link on web page. Create this file in WebContent folder. This path of the file should look like this: WebContent/index.html

<!DOCTYPE html>

<html>

<head>

<metacharset="UTF-8">

<title>Generic Servlet Demo</title>

</head>

<body>

<ahref="welcome">Click to call Servlet</a>

</body>

</html>

**ExampleGeneric.java**  
Now, we are creating a Generic Servlet by extending GenericServlet class. When creating a GenericServlet you should always override service() method. Right click on the src folder and create a new class file, name the file as ExampleGeneric. The file path should look like this: Java Resouces/src/default package/ExampleGeneric.java

import java.io.\*;

import javax.servlet.\*;

publicclassExampleGenericextendsGenericServlet{

publicvoid service(ServletRequest req,ServletResponse res)

throwsIOException,ServletException{

res.setContentType("text/html");

PrintWriter pwriter=res.getWriter();

pwriter.print("<html>");

pwriter.print("<body>");

pwriter.print("<h2>Generic Servlet Example</h2>");

pwriter.print("<p>Hello Book Readers!</p>");

pwriter.print("</body>");

pwriter.print("</html>");

}

}

**web.xml**  
This file can be found at this path WebContent/WEB-INF/web.xml. In this file we will map the Servlet with the specific URL. Since we are calling welcome page upon clicking the link on index.html page so we are mapping the welcome page to the Servlet class we created above.

<web-app>

<display-name>BookServlet</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>MyGenericServlet</servlet-name>

<servlet-class>ExampleGeneric</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyGenericServlet</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

## Methods of GenericServlet class:

Here is the list of all the methods of GenericServlet class.  
1. **public void init()**: it is a convenient method. This method can be overridden so that there’s no need to call super.init(config).

2. **public void init(ServletConfig config)**: Called by the servlet container to indicate that the servlet is being placed into service, this method is used for initializing the servlet.

3. **public String getInitParameter(String name)**: Returns a String containing the value of the given initialization parameter, or null if the parameter does not exist.

4. **public Enumeration getInitParameterNames()**: Returns the names of all the parameters defined in the web.xml file or null if web.xml does’t have any parameter.

5. **public abstract void service(ServletRequest request, ServletResponse response)**: Called by the Servlet container to allow servlet to respond to the requests made by client.

6. **public void destroy()**: It is called by servlet container once at the end of servlet life cycle to indicate that servlet is being destroyed.

7. **public ServletConfig getServletConfig()**: Return the ServletConfig object that initialized this servlet

8. **public String getServletInfo()**: Returns information about servlet.

9. **public ServletContext getServletContext()**: Return ServletContext object, passed to this servlet by the init method

10. **public String getServletName()**: Returns the name of the servlet instance.

11. **public void log(String msg)**: Writes the given message in the servlet log file.

12. **public void log(String msg,Throwable t)**: Writes the explanatory message in the servlet log file including a String that describes the error or exception.

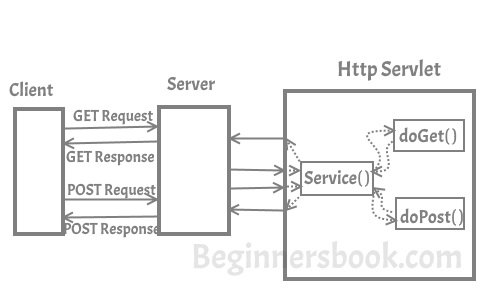
# HttpServlet class with example

Unlike Generic Servlet, the HTTP Servlet doesn’t override the service() method. Instead it overrides the doGet() method or doPost() method or both. The doGet() method is used for getting the information from server while the doPost() method is used for sending information to the server.

In Http Servlet there is no need to override the service() method because this method dispatches the Http Requests to the correct method handler, for example if it receives HTTP GET Request it dispatches the request to the doGet() method.

## How Http Servlet works?

As you can see in the diagram below that client (user’s browser) make requests. These requests can be of any type, for example – Get Request, Post Request, Head Request etc. Server dispatches these requests to the servlet’s service() method, this method dispatches these requests to the correct handler for example if it receives Get requests it dispatches it to the doGet() method.



## Hierarchy of Http Servlet

java.lang.Object

|\_extended byjavax.servlet.GenericServlet

|\_extended byjavax.servlet.http.HttpServlet

We have already discussed in the Generic Servlet article that you should always use HttpServlet instead of the GenericServlet. HttpServlet is easier to work with, and has more methods to work with than GenericServlet.

## Http Servlet example

I am using Netbeans IDE for this example. Create New “Dynamic Web Project” from the file menu.

**index.html**  
We are creating an html file that would call the servlet once we click on the link on web page. Create this file in WebContent folder. The path of the file should look like this: WebContent/index.html

index<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>HttpServletDemo</title>

</head>

<body>

<a href="welcome">Click to call Servlet</a>

</body>

</html>

**ExampleHttpServlet.java**  
Now, we are creating a Http Servlet by extending HttpServlet class. Right click on the src folder and create a new class file, name the file as ExampleHttpServlet. The file path should look like this: Java Resources/src/default package/ExampleHttpServlet.java

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

// Creating Http Servlet by Extending HttpServlet class

publicclassExampleHttpServletextendsHttpServlet

{

privateString mymsg;

publicvoid init()throwsServletException

{

mymsg="Http Servlet Demo";

}

publicvoid doGet(HttpServletRequest request,

HttpServletResponse response) throwsServletException,

IOException

{

// Setting up the content type of web page

response.setContentType("text/html");

// Writing the message on the web page

PrintWriterout=response.getWriter();

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

out.println("<p>"+"Hello Friends!"+"</p>");

}

publicvoid destroy()

{

// Leaving empty. Use this if you want to perform

//something at the end of Servlet life cycle.

}

}

**web.xml**  
This file can be found at this path WebContent/WEB-INF/web.xml. In this file we will map the Servlet with the specific URL. Since we are calling welcome page upon clicking the link on index.html page so we are mapping the welcome page to the Servlet class we created above.

<web-app>

<display-name>BookServlet</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>MyHttpServlet</servlet-name>

<servlet-class>ExampleHttpServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyHttpServlet</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

## Methods of HttpServlet class

1. **protected void doGet(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the HTTP GET request from client. When overriding this method, read the request data, write the response headers, get the response’s writer or output stream object, and finally, write the response data.

2. **protected long getLastModified(HttpServletRequest req)**: Returns a long integer specifying the time the HttpServletRequest object was last modified, in milliseconds since midnight, January 1, 1970 GMT, or -1 if the time is not known

3. **protected void doHead(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the HTTP HEAD request from client. The client sends a HEAD request when it wants to see only the headers of a response, such as Content-Type or Content-Length

4. **protected void doPost(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the POST request from client. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information to the server. Unlike, doGet where we get information from the sever this method is used when we are transferring information from client to the server.

5. **protected void doPut(HttpServletRequest req, HttpServletResponse resp)**: This method is called by servlet service method to handle the PUT request from client. This method is similar to doPost method but unlike doPost method where we send information to the server, this method sends file to the server, this is similar to the FTP operation from client to server.

6. **protected void doDelete(HttpServletRequest req, HttpServletResponse resp)**: Called by servlet service() method to handle the DELETE request from client that allows a client to delete a document, webpage or information from the server.

7. **protected void doOptions(HttpServletRequest req, HttpServletResponse resp)**: Called by the service method to allow a servlet to handle a OPTIONS request. The OPTIONS request determines which HTTP methods the server supports and returns an appropriate header.

8. **protected void doTrace(HttpServletRequest req, HttpServletResponse resp)**: This method is called by service() method for handling TRACE request. Used for debugging purposes.

9. **protected void service(HttpServletRequest req, HttpServletResponse resp)**: There is no need to override this method, this method receives the HTTP request from client and forwards them to the corresponding doXXX methods such as doGet(), doPost(), doHEAD() etc.

10. **public void service(ServletRequest req, ServletResponse res)**: Forwards client request to the protected service method. There’s no need to override this method as well.

# Servlet Life Cycle

Servlet life cycle can be described as a series of steps through which a servlet goes during its life span, starting from loading till it gets destroyed.

Before I start explaining the life cycle of Servlet, lets discuss few terminologies that you will encounter while reading this guide. It is important to learn what each term means, this will help you understand things faster.

**Web Server**: It is also known as HTTP Server, it can handle HTTP Requests send by client and responds the request with an HTTP Response.

**Web Container**: Also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

Note: The servlet post you find in this post uses apache tomcat web server. Although I mentioned it as web server, it is in fact a web server and web container both. (As mentioned above web container is a part of web server).

## Life Cycle of Servlet

Servlet life cycle contains five steps: 1) Loading of Servlet 2) Creating instance of Servlet 3) Invoke init() once 4) Invoke service() repeatedly for each client request 5) Invoke destroy()

For those who are wondering what is instance and invoke means: Instance and objects are same thing. Invoking a method means calling a method, it is just a fancy word that we use in programming world in place of calling :)

Let’s back to the main topic. Here are the five steps of servlet life cycle.

**Step 1: Loading of Servlet**  
When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets.

**Step 2: Creating instance of Servlet**  
Once all the Servlet classes loaded, the servlet container creates instances of each servlet class. Servlet container creates only once instance per servlet class and all the requests to the servlet are executed on the same servlet instance.

**Step 3: Invoke init() method**  
Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. This method initializes the servlet. There are certain init parameters that you can specify in the deployment descriptor (web.xml) file. For example, if a servlet has value >=0 then its init() method is immediately invoked during web container startup.

You can specify the element in web.xml file like this:

<servlet>

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

<servlet-class>MyServletDemo</servlet-class>

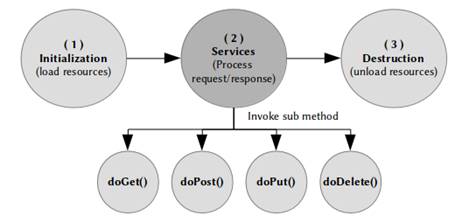
<load-on-startup>1</load-on-startup>

</servlet>

Now the init() method for corresponding servlet class **com.booksNbooks.MyServletDemo** would be invoked during web container startup.

**Note: The init() method is called only once during the life cycle of servlet.**

**Step 4: Invoke service() method**  
Each time the web server receives a request for servlet, it spawns a new thread that calls service() method. If the servlet is GenericServlet then the request is served by the service() method itself, if the servlet is HttpServlet then service() method receives the request and dispatches it to the correct handler method based on the type of request.

For example if its a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

**Note**: Unlike init() and destroy() that are called only once, the service() method can be called any number of times during servlet life cycle. As long as servlet is not destroyed, for each client request the service() method is invoked.

**Out of all the 5 steps in life cycle, this is the only step that executes multiple times.**

**Step 5: Invoke destroy() method**  
When servlet container shuts down(this usually happens when we stop the web server), it unloads all the servlets and calls destroy() method for each initialized servlets.

# Working of Servlet

**Web Server**: it can handle HTTP Requests send by clients and responds the request with an HTTP Response.

**Web Application(webapp)**: I would refer this as webapp in this guide. Basically the project is your web application, it is the collection of servlets.

**Web Container**: Also known as Servlet Container and Servlet Engine. It is a part of Web Server that interacts with Servlets. This is the main component of Web Server that manages the life cycle of Servlets.

## How Servlet Works?

1) When the web server (e.g. Apache Tomcat) starts up, the servlet container deploy and loads all the servlets. During this step Servlet container creates ServletContext object. **ServletContext is an interface that defines the set of methods that a servlet can use to communicate with the servlet container**.

**Note**: **There is only one ServletContext per webapp** which is common to all the servlets. ServletContext has several useful methods such as addListener(), addFilter() etc. For now I am not explaining them as I will cover them in a separate text about ServletContext.

2) Once the servlet is loaded, the servlet container creates the instance of servlet class. For each instantiated servlet, its init() method is invoked.

3) Client (user browser) sends an Http request to web server on a certain port. Each time the web server receives a request, the servlet container creates HttpServletRequest and HttpServletResponse objects. The HttpServletRequest object provides the access to the request information and the HttpServletResponse object allows us to format and change the http response before sending it to the client.

The servlet container spawns a new thread that calls service() method for each client request. **The service() method dispatches the request to the correct handler method based on the type of request**.  
For example if server receives a Get Request the service() method would dispatch the request to the doGet() method by calling the doGet() method with request parameters. Similarly the requests like Post, Head, Put etc. are dispatched to the corresponding handlers doPost(), doHead(), doPut() etc. by service() method of servlet.

4) When servlet container shuts down, it unloads all the servlets and calls destroy() method for each initialized servlets.

# welcome-file-list tag in web.xml file of Project

Have you ever seen <welcome-file-list> tag in your web.xml file and wondering what it is? In this text, I will explain what is this tag and why we use it.

The tag <welcome-file-list> is used for specifying the files that needs to be invoked by server by default, if you do not specify a file name while loading the project on browser.

For e.g. You have created a project named “MyServletProject” and you have few html pages and servlet classes defined in the project. However in browser you have given the url like this:

http://localhost:8080/MyServletProject

Usually we give the complete path like this:http://localhost:8080/MyServletProject/index.html. However if you have given the path like above then the webserver will look for the <welcome-file-list> tag in your project’s web.xml file. Lets say you have the following content in your web.xml file:

<web-app>

....

<welcome-file-list>

<welcome-file>myhome.htm</welcome-file>

<welcome-file>myindex.htm</welcome-file>

<welcome-file>mydefaultpage.htm</welcome-file>

</welcome-file-list>

....

</web-app>

Based on the welcome file list, server would look for the myhome.htm page if this doesn’t exist then the second welcome file myindex.html and so on till it finds a valid welcome file.

**Note**: If the <welcome-file-list> tag is not defined in web.xml or the welcome files defined in the <welcome-file> tags does not exist then the server would look for the following files in the given sequence:  
1) index.html  
2) index.htm  
3) index.jsp

# How to use load-on-startup tag in web.xml file with Example

By default Servlet is not loaded until servlet container receives a request for the particular servlet. This may cause a delay in accessing the servlet for the first time. To avoid the delay in access time, you can use **<load-on-startup> tag in your web.xml** file that allows you to force the servlet container to load (instantiated and have its init() called) the servlet as soon as the server starts.

## How to use <load-on-startup>?

Here is a sample web.xml file:

<web-app>

…

<servlet>

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

<servlet-class>DemoServlet</servlet-class>

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

</servlet>

…

</web-app>

If I didn’t specify the <load-on-startup>, the web container would not have loaded the servlet until it receives a request for DemoServlet servlet class. Since I have specified a value >=0, this servlet (DemoServlet class) would be loaded on the startup.

value>= 0 means that the servlet is loaded when the web-app is deployed or when the server starts, if the value < 0 then servlet would be loaded whenever the container feels like.

### How to specify the order of servlet loading using <load-on-startup> tag?

<web-app>

…

<servlet>

<servlet-name>MyServlet1</servlet-name>

<servlet-class>DemoServlet1</servlet-class>

<load-on-startup>5</load-on-startup>

</servlet>

<servlet>

<servlet-name>MyServlet2</servlet-name>

<servlet-class>DemoServlet2</servlet-class>

<load-on-startup>0</load-on-startup>

</servlet>

<servlet>

<servlet-name>MyServlet3</servlet-name>

<servlet-class>DemoServlet3</servlet-class>

<load-on-startup>-2</load-on-startup>

</servlet>

…

</web-app>

In this example we have three Servlets specified in the web.xml file, since servlet classes DemoServlet1 and DemoServlet2 has load-on-startup value >=0, they both will be loaded as soon as the server starts. However servlet class DemoServlet2 would be loaded before the DemoServlet1 class because it has lower load-on-startup value.

Servlet class DemoServlet3 would not be loaded on startup as it has negative load-on-startup value.

# ServletRequest Interface with example

When a client sends a request to the web server, the servlet container creates ServletRequest & ServletResponse objects and passes them as an argument to the servlet’s service() method. The request object provides the access to the request information such as header and body information of request data.

First we will see an example and then we will see the list of methods available in the ServletRequest interface:

## Example 1: ServletRequest getParameter() method to display the user input

In this example I am demonstrating the use of getParameter() method that returns the value of the given parameter.

In this html form, we are taking user input (name and age) and storing them in the parameters uname and uage respectively.  
index.html

<formaction="details"method="get">

User Name: <inputtype="text"name="uname"><br>

User Age: <inputtype="text"name="uage"><br>

<inputtype="submit"value="submit">

</form>

MyServletDemo.java  
In this servlet class we are getting the value of the parameters by using getParameter() method, this method belongs to the ServletRequest interface. In this example we have HttpServletRequest as a parameter of doGet() method, HttpServletRequest extends ServletRequest interface thats why the getParameter() method is available to the req object.

After getting the values, we are writing them on the webpage.

import javax.servlet.http.\*;

import javax.servlet.\*;

import java.io.\*;

publicclassMyServletDemoextendsHttpServlet{

publicvoid doGet(HttpServletRequest req,HttpServletResponse res)

throwsServletException,IOException

{

res.setContentType("text/html");

PrintWriter pwriter=res.getWriter();

String name =req.getParameter("uname");

String age =req.getParameter("uage");

pwriter.println("Name: "+name);

pwriter.println("Age: "+age);

pwriter.close();

}

}

Web.xml  
This is your deployment descriptor file that maps the servlet to the url. Since our form has details page as action, we mapped the servlet class to the details page.

<web-app>

<display-name>BookDemo</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>MyServletDemo</servlet-name>

<servlet-class>MyServletDemo</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyServletDemo</servlet-name>

<url-pattern>/details</url-pattern>

</servlet-mapping>

</web-app>

## Example 2: Getting parameter names and values

In this example, we will be using getParameterNames() and getParameter() methods to get parameter names and values.  
getParameterNames(): Returns an Enumeration of String objects containing the names of the parameters contained in this request. If the request has no parameters, the method returns an empty Enumeration.

getParameter(): As mentioned above, this returns the value of given parameter.  
**index.html**

<formaction="details"method="get">

User Name: <inputtype="text"name="uname"><br>

User Age: <inputtype="text"name="uage"><br>

<inputtype="submit"value="submit">

</form>

MyServletDemo.class

import java.io.IOException;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Enumeration;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

publicclassMyServletDemoextendsHttpServlet{

publicvoid doGet(HttpServletRequest req,HttpServletResponse res)

throwsServletException,IOException

{

PrintWriter pwriter=res.getWriter(); res.setContentType("text/html");

Enumeration en=req.getParameterNames();

while(en.hasMoreElements())

{

Object obj=en.nextElement();

String param=(String)obj;

String pvalue=req.getParameter(param);

pwriter.print("Parameter Name: "+param+

" Parameter Value: "+pvalue);

}

pwriter.close();

}

}

web.xml

<web-app>

<servlet>

<servlet-name>MyServletDemo</servlet-name>

<servlet-class>MyServletDemo</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>MyServletDemo</servlet-name>

<url-pattern>/details</url-pattern>

</servlet-mapping>

</web-app>

## Example 3: Display Header information

index.html

<h1>Servlet Request Demo</h1>

<body>

<ahref="headinfo">Click Here</a>

</body>

HeaderDetails.java

import java.io.IOException;

import java.io.PrintWriter;

import java.util.Enumeration;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

publicclassHeaderDetailsextendsHttpServlet{

publicvoid doGet(HttpServletRequest request,

HttpServletResponse response)

throwsIOException,ServletException

{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

pwriter.println("HTTP header Information:<br>");

Enumeration en =request.getHeaderNames();

while(en.hasMoreElements()){

String hName =(String)en.nextElement();

String hValue =request.getHeader(hName);

pwriter.println("<b>"+hName+": </b>"

+hValue +"<br>");

}

}

}

web.xml

<web-app>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>HeaderDetails</servlet-name>

<servlet-class>HeaderDetails</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>HeaderDetails</servlet-name>

<url-pattern>/headinfo</url-pattern>

</servlet-mapping>

</web-app>

## Methods of ServletRequest interface

**String getParameter(String name)**: It returns the value of the given parameter as String or null if the given parameter does not exist.

**Enumeration getParameterNames()**: It returns an Enumeration of Strings objects containing the names of parameters in the request.

**String[] getParameterValues(String name)**: It returns an array of Strings containing the all the values, the parameters has, returns null if parameter doesn’t have any value.

**String getCharacterEncoding()**: Returns the name of the character encoding used in the body of this request. This method returns null if the request does not specify a character encoding.

**void setCharacterEncoding(String env)**: Overrides the character encoding in the body of the request.

**int getContentLength()**: Returns the length of the request content in bytes.

**String getContentType()**: Returns the MIME type of the body of the request, or null if the type is not known.

# RequestDispatcher methods with examples in Servlet

The **RequestDispatcher** interface defines an object that receives the request from client and dispatches it to the resource(such as servlet, JSP, HTML file). This interface has following two methods:

**public void forward(ServletRequest request, ServletResponse response)**: It forwards the request from one servlet to another resource (such as servlet, JSP, HTML file).

**public void include(ServletRequest request, ServletResponse response)**: It includes the content of the resource(such as servlet, JSP, HTML file) in the response.

## Difference between forward() vs include() method

To understand the difference between these two methods, lets take an example: Suppose you have two pages X and Y. In page X you have an include tag, this means that the control will be in the page X till it encounters include tag, after that the control will be transferred to page Y. At the end of the processing of page Y, the control will return back to the page X starting just after the include tag and remain in X till the end.  
**In this case the final response to the client will be send by page X.**

Now, we are taking the same example with forward. We have same pages X and Y. In page X, we have forward tag. In this case the control will be in page X till it encounters forward, after this the control will be transferred to page Y. The main difference here is that the control will not return back to X, it will be in page Y till the end of it.  
**In this case the final response to the client will be send by page Y.**

## Example:

In this example, I will be using both the methods include and forward. Using include method, I will be changing the content of current page and when I’m ready to transfer the control to the next page, I will use forward method.

**index.html**

<formaction="loginPage"method="post">

User Name:<inputtype="text"name="uname"/><br/>

Password:<inputtype="password"name="upass"/><br/>

<inputtype="submit"value="SUBMIT"/>

</form>

**Validation.java**

import java.io.\*;

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassValidationextendsHttpServlet

{

publicvoid doPost(HttpServletRequest request,

HttpServletResponse response)

throwsServletException,IOException

{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

String name=request.getParameter("uname");

Stringpass=request.getParameter("upass");

if(name.equals("Chaitanya")&&

pass.equals("book"))

{

RequestDispatcher dis=request.getRequestDispatcher("welcome");

dis.forward(request, response);

}

else

{

pwriter.print("User name or password is incorrect!");

RequestDispatcher dis=request.getRequestDispatcher("index.html");

dis.include(request, response);

}

}

}

**WelcomeUser.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassWelcomeUserextendsHttpServlet{

publicvoid doPost(HttpServletRequest request,

HttpServletResponse response)

throwsServletException,IOException

{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

String name=request.getParameter("uname");

pwriter.print("Hello "+name+"!");

pwriter.print(" Welcome to book.com");

}

}

**web.xml**

<web-app>

<display-name>BookDemo</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>Login</servlet-name>

<servlet-class>Validation</servlet-class>

</servlet>

<servlet>

<servlet-name>Welcome</servlet-name>

<servlet-class>WelcomeUser</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Login</servlet-name>

<url-pattern>/loginPage</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>Welcome</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>

# ServletConfig Interface with example

Servlet Container creates ServletConfig object for each Servlet during initialization, to pass information to the Servlet. This object can be used to get configuration information such as parameter name and values from deployment descriptor file(web.xml).

## Methods of ServletConfig interface

**public String getInitParameter(String name)**: Returns the value of given parameter as String, or null if the given parameter doesn’t exist in web.xml.  
**public Enumeration getInitParameterNames()**: Returns an enumeration of all the parameter names.  
**public String getServletName()**: Returns the name of the servlet instance.  
**public ServletContext getServletContext()**: Returns an object of ServletContext.

## Example:

In this example, we will use two methods getInitParameter() and getInitParameterNames() to get all the parameters from web.xml along with their values.  
The getInitParameterNames() method returns an enumeration of all parameters names and by passing those names during the call of getInitParameter() method, we can get the corresponding parameter value from web.xml.

**DemoServlet.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

import java.util.Enumeration;

publicclassDemoServletextendsHttpServlet{

protectedvoid doGet(HttpServletRequest request,

HttpServletResponse response)

throwsServletException,IOException

{

response.setContentType("text/html;charset=UTF-8");

PrintWriter pwriter =response.getWriter();

ServletConfig sc=getServletConfig();

Enumeration<String> e=sc.getInitParameterNames();

Stringstr;

while(e.hasMoreElements()){

str=e.nextElement();

pwriter.println("<br>Param Name: "+str);

pwriter.println(" value: "+sc.getInitParameter(str));

}

}

}

**web.xml**

<web-app>

<display-name>BeginnersBookDemo</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

<servlet>

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

<servlet-class>DemoServlet</servlet-class>

<init-param>

<param-name>MyName</param-name>

<param-value>Chaitanya</param-value>

</init-param>

<init-param>

<param-name>MyWebsite</param-name>

<param-value>Mybook.com</param-value>

</init-param>

</servlet>

<servlet-mapping>

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

<url-pattern>/scdemo</url-pattern>

</servlet-mapping>

</web-app>

# ServletContext Interface

We discussed ServletConfig, the Servlet Container creates ServletConfig object for each Servlet during initialization. The main difference between ServletConfig and ServletContext is that unlike ServletConfig, the ServletContext is being created once per web application, i.e. ServletContext object is common to all the servlets in web application.

This is how we can create ServletContext object. In this code we are creating object in init() method, however you can create the object anywhere you like.

ServletContext sc;

publicvoid init(ServletConfig scfg)

{

sc=scfg.getServletContext();

}

Once we have the ServletContext object, we can set the attributes of the ServletContext object by using the setAttribute() method. Since the ServletContext object is available to all the servlets of the Web application, other servlets can retrieve the attribute from the ServletContext object by using the getAttribute() method.

**Context Initialization Parameter**  
Context Initialization parameters are the parameter name and value pairs that you can specify in the deployment descriptor file (the web.xml file). Here you can specify the parameters that will be accessible to all the servlets in the web application.

When we deploy the Web application, the Servlet container reads the initialization parameter from the web.xml file and initializes the ServletContext object with it. We can use the getInitParameter()and getInitParameterNames() methods of the ServletContext interface to get the parameter value and enumeration of parameter names respectively.

For example, here I have specified the parameter email\_id with the value, since this is common to all the servlets, you can get the parameter name and value in any servlet.

<context-param>

<param-name>email\_id</param-name>

<param-value>abc@gmail.com</param-value>

</context-param>

#### ServletContext complete example: To get the initialization parameters

In this example we have two context initialization parameters (user name and user email) in web.xml file and we are getting the value in Servlet using getInitParameter() method that returns the value of given parameter.

DemoServlet.java

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassDemoServletextendsHttpServlet{

publicvoid doGet(HttpServletRequest request,HttpServletResponse response)

throwsServletException,IOException

{

response.setContentType("text/html");

PrintWriter pwriter=response.getWriter();

//ServletContext object creation

ServletContext scontext=getServletContext();

//fetching values of initialization parameters and printing it

String userName=scontext.getInitParameter("uname");

pwriter.println("User name is="+userName);

String userEmail=scontext.getInitParameter("email");

pwriter.println("Email Id is="+userEmail);

pwriter.close();

}

}

web.xml

<web-app>

<servlet>

<servlet-name>DemoServlet</servlet-name>

<servlet-class>DemoServlet</servlet-class>

</servlet>

<context-param>

<param-name>uname</param-name>

<param-value>ChaitanyaSingh</param-value>

</context-param>

<context-param>

<param-name>email</param-name>

<param-value>abc@gmail.com</param-value>

</context-param>

<servlet-mapping>

<servlet-name>DemoServlet</servlet-name>

<url-pattern>/context</url-pattern>

</servlet-mapping>

</web-app>

## Methods of ServletContext interface

Here is the list of frequently used methods of ServletContext interface.

**public String getInitParameter(String param)**: It returns the value of given parameter or null if the parameter doesn’t exist.

**public Enumeration getInitParameterNames()**: Returns an enumeration of context parameters names.

**public void setAttribute(String name,Object object)**: Sets the attribute value for the given attribute name.

**public Object getAttribute(String name)**:Returns the attribute value for the given name or null if the attribute doesn’t exist.

**public String getServerInfo()**: eturns the name and version of the servlet container on which the servlet is running.

**public String getContextPath()**: Returns the context path of the web application.  
To get the full list of methods, refer the official documentation [here](https://docs.oracle.com/javaee/7/api/javax/servlet/ServletContext.html).

# ServletResponse Interface

The servlet container is connected to the web server that receives Http Requests from client on a certain port. When client sends a request to web server, the servlet container creates HttpServletRequest and HttpServletResponse objects and passes them as an argument to the servlet service() method.

The response object allows you to format and send the response back to the client. First we will see the commonly used methods in the ServletReponse interface and then we will see an example.

## Method of ServletResponse interface

1) String getCharacterEncoding(): It returns the name of the MIME charset used in body of the response sent to the client.  
2) String getContentType(): It returns the response content type. e.g. text, html etc.  
3) ServletOutputStream getOutputStream(): Returns a ServletOutputStream suitable for writing binary data in the response.  
4) java.io.PrintWriter getWriter(): Returns the PrintWriter object.  
5) void setCharacterEncoding(java.lang.String charset): Set the MIME charset (character encoding) of the response.  
6) void setContentLength(int len): It sets the length of the response body.  
7) void setContentType(java.lang.String type): Sets the type of the response data.  
8) void setBufferSize(int size): Sets the buffer size.  
9) int getBufferSize(): Returns the buffer size.  
10) void flushBuffer(): Forces any content in the buffer to be written to the client.  
11) boolean isCommitted(): Returns a boolean indicating if the response has been committed.  
12) void reset(): Clears the data of the buffer along with the headers and status code.

## Example:

In the below example, we have used setContentType() and getWriter() methods of ServletResponse interface.

index.html

<formaction="mydetails"method="get">

User name: <inputtype="text"name="uname">

<inputtype="submit"value="login">

</form>

MyServletDemo.java

import javax.servlet.http.\*;

import javax.servlet.\*;

import java.io.\*;

publicclassMyServletDemoextendsHttpServlet{

publicvoid doGet(HttpServletRequest req,HttpServletResponse res)

throwsServletException,IOException

{

res.setContentType("text/html");

PrintWriter pwriter=res.getWriter();

String name=req.getParameter("uname");

pwriter.println("User Details Page:");

pwriter.println("Hello "+name);

pwriter.close();

}

}

web.xml

<web-app>

<servlet>

<servlet-name>DemoServlet</servlet-name>

<servlet-class>MyServletDemo</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>DemoServlet</servlet-name>

<url-pattern>/mydetails</url-pattern>

</servlet-mapping>

</web-app>

# HttpSession with example in Servlet

The HttpSession object is used for session management. A session contains information specific to a particular user across the whole application. When a user enters into a website (or an online application) for the first time HttpSession is obtained via request.getSession(), the user is given a unique ID to identify his session. This unique ID can be stored into a cookie or in a request parameter.

The HttpSession stays alive until it has not been used for more than the timeout value specified in tag in deployment descriptor file( web.xml). The default timeout value is 30 minutes, this is used if you don’t specify the value in tag. This means that when the user doesn’t visit web application time specified, the session is destroyed by servlet container. The subsequent request will not be served from this session anymore, the servlet container will create a new session.

This is how you create a HttpSession object.

protectedvoid doPost(HttpServletRequest req,

HttpServletResponse res)

throwsServletException,IOException{

HttpSession session =req.getSession();

}

You can store the user information into the session object by using setAttribute() method and later when needed this information can be fetched from the session. This is how you store info in session. Here we are storing username, emailid and userage in session with the attribute name uName, uemailId and uAge respectively.

session.setAttribute("uName","Abc");

session.setAttribute("uemailId","myemailid@gmail.com");

session.setAttribute("uAge","30");

This First parameter is the attribute name and second is the attribute value. For e.g. uName is the attribute name and Abcis the attribute value in the code above.

TO get the value from session we use the getAttribute() method of HttpSession interface. Here we are fetching the attribute values using attribute names.

String userName =(String)session.getAttribute("uName");

String userEmailId =(String)session.getAttribute("uemailId");

String userAge =(String)session.getAttribute("uAge");

## Methods of HttpSession

**public void setAttribute(String name, Object value)**: Binds the object with a name and stores the name/value pair as an attribute of the HttpSession object. If an attribute already exists, then this method replaces the existing attributes.

**public Object getAttribute(String name)**: Returns the String object specified in the parameter, from the session object. If no object is found for the specified attribute, then the getAttribute() method returns null.

**public Enumeration getAttributeNames()**: Returns an Enumeration that contains the name of all the objects that are bound as attributes to the session object.

**public void removeAttribute(String name)**: Removes the given attribute from session.

**setMaxInactiveInterval(int interval)**: Sets the session inactivity time in seconds. This is the time in seconds that specifies how long a sessions remains active since last request received from client.

## Session Example

index.html

<formaction="login">

User Name:<inputtype="text"name="userName"/><br/>

Password:<inputtype="password"name="userPassword"/><br/>

<inputtype="submit"value="submit"/>

</form>

MyServlet1.java

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassMyServlet1extendsHttpServlet{

publicvoid doGet(HttpServletRequest request,HttpServletResponse response){

try{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

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

String password =request.getParameter("userPassword");

pwriter.print("Hello "+name);

pwriter.print("Your Password is: "+password);

HttpSession session=request.getSession();

session.setAttribute("uname",name);

session.setAttribute("upass",password);

pwriter.print("<a href='welcome'>view details</a>");

pwriter.close();

}catch(Exception exp){

System.out.println(exp);

}

}

}

MyServlet2.java

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassMyServlet2extendsHttpServlet{

publicvoid doGet(HttpServletRequest request,HttpServletResponse response){

try{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

HttpSession session=request.getSession(false);

String myName=(String)session.getAttribute("uname");

String myPass=(String)session.getAttribute("upass");

pwriter.print("Name: "+myName+" Pass: "+myPass);

pwriter.close();

}catch(Exception exp){

System.out.println(exp);

}

}

}

web.xml

<web-app>

<servlet>

<servlet-name>Servlet1</servlet-name>

<servlet-class>MyServlet1</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Servlet1</servlet-name>

<url-pattern>/login</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>Servlet2</servlet-name>

<servlet-class>MyServlet2</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Servlet2</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

# Cookies in Servlet with example

Here we will discuss Cookies which is also used for session management. Let’s recall few things here from last tutorial so that we can relate sessions and cookies. When a user visits web application first time, the servlet container crates new HttpSession object by calling request.getSession(). A unique Id is assigned to the session. The **Servlet container also sets a Cookie in the header of the HTTP response with cookie name and the unique session ID as its value.**

The cookie is stored in the user browser, the client (user’s browser) sends this cookie back to the server for all the subsequent requests until the cookie is valid. **The Servlet container checks the request header for cookies and get the session information from the cookie and use the associated session from the server memory.**

The session remains active for the time specified in tag in web.xml. If tag in not set in web.xml then the session remains active for 30 minutes. **Cookie remains active as long as the user’s browser is running**, as soon as the browser is closed, the cookie and associated session info is destroyed. So when the user opens the browser again and sends request to web server, the new session is being created.

## Types of Cookies

We can classify the cookie based on their expiry time:

1. Session
2. Persistent

**1) SessionCookies:**  
Session cookies do not have expiration time. It lives in the browser memory. As soon as the web browser is closed this cookie gets destroyed.

**2) Persistent Cookies:**  
Unlike Session cookies they have expiration time, they are stored in the user hard drive and gets destroyed based on the expiry time.

## How to send Cookies to the Client

Here are steps for sending cookie to the client:

1. Create a Cookie object.
2. Set the maximum Age.
3. Place the Cookie in HTTP response header.

#### 1) Create a Cookie object:

Cookie c =newCookie("userName","Abc");

#### 2) Set the maximum Age:

By using **setMaxAge ()** method we can set the maximum age for the particular cookie in seconds.

c.setMaxAge(1800);

#### 3) Place the Cookie in HTTP response header:

We can send the cookie to the client browser through response.addCookie() method.

response.addCookie(c);

## How to read cookies

Cookiec[]=request.getCookies();

//c.length gives the cookie count

for(int i=0;i<c.length;i++){

out.print("Name: "+c[i].getName()+" & Value: "+c[i].getValue());

}

## Example of Cookies in java servlet

**index.html**

<formaction="login">

User Name:<inputtype="text"name="userName"/><br/>

Password:<inputtype="password"name="userPassword"/><br/>

<inputtype="submit"value="submit"/>

</form>

**MyServlet1.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassMyServlet1extendsHttpServlet

{

publicvoid doGet(HttpServletRequest request,

HttpServletResponse response){

try{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

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

String password =request.getParameter("userPassword");

pwriter.print("Hello "+name);

pwriter.print("Your Password is: "+password);

//Creating two cookies

Cookie c1=newCookie("userName",name);

Cookie c2=newCookie("userPassword",password);

//Adding the cookies to response header

response.addCookie(c1);

response.addCookie(c2);

pwriter.print("<br><a href='welcome'>View Details</a>");

pwriter.close();

}catch(Exception exp){

System.out.println(exp);

}

}

}

**MyServlet2.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

publicclassMyServlet2extendsHttpServlet{

publicvoid doGet(HttpServletRequest request,

HttpServletResponse response){

try{

response.setContentType("text/html");

PrintWriter pwriter =response.getWriter();

//Reading cookies

Cookiec[]=request.getCookies();

//Displaying User name value from cookie

pwriter.print("Name: "+c[1].getValue());

//Displaying user password value from cookie

pwriter.print("Password: "+c[2].getValue());

pwriter.close();

}catch(Exception exp){

System.out.println(exp);

}

}

}

**web.xml**

<web-app>

<display-name>BookDemo</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

<servlet>

<servlet-name>Servlet1</servlet-name>

<servlet-class>MyServlet1</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Servlet1</servlet-name>

<url-pattern>/login</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>Servlet2</servlet-name>

<servlet-class>MyServlet2</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Servlet2</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

## Methods of Cookie class

**public void setComment(String purpose)**: This method is used for setting up comments in the cookie. This is basically used for describing the purpose of the cookie.

**public String getComment()**: Returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

**public void setMaxAge(int expiry)**: Sets the maximum age of the cookie in seconds.

**public int getMaxAge()**: Gets the maximum age in seconds of this Cookie.  
By default, -1 is returned, which indicates that the cookie will persist until browser shutdown.

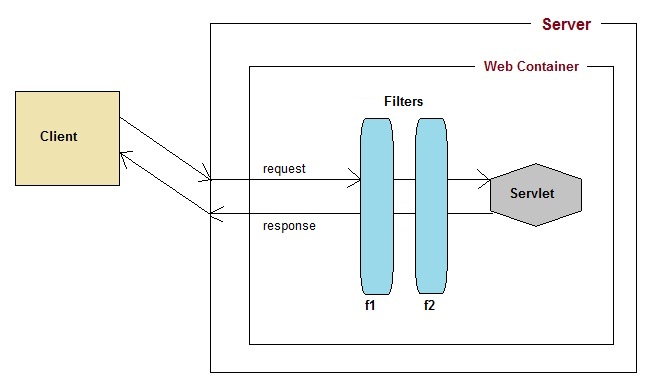
**public String getName()**: Returns the name of the cookie. The name cannot be changed after creation.

**public void setValue(String newValue)**: Assigns a new value to this Cookie.

**public String getValue()**: Gets the current value of this Cookie.

# Introduction to Filter API

Filters are compontents that you can use and configure to perform some filtering tasks. Filter is used for pre-processing of requests and post-processing of responses. You can have any number of filters for pre-processing of a request and post-processing of a response. Filters are configured in the deployment descriptor of a web application.



## How Filters Works?

* When a request reaches the **Web Container**, it checks if any filter has URL patterns that matches the requested URL.
* The **Web Container** locates the first filter with a matching URL pattern and filter's code is executed.
* If another filter has a matching URL pattern, its code is then executed. This continues until there are no filters with matching URL patterns left.
* If no error occurs, the request passes to the target servlet. Hence we know, that the request will be passed to the target servlet only when all the related Filters are successfully executed.
* The servlet returns the response back to its caller. The last filter that was applied to the request is the first filter applied to the response.
* At last the response will be passed to the **Web Container** which passes it to the client.

## More about Filter API

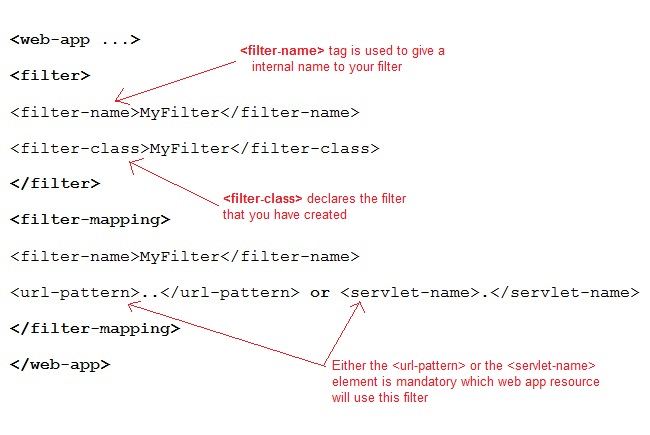
**Filter API** is part of **Servlet API**. Filter interface is found in the **javax.servlet** package.

For creating a filter, we must implement Filter interface. Filter interface gives the following life cycle methods for a filter:

1. void init(FilterConfig filterConfig): invoked by the web container to indicate to a filter that it is being placed into service.
2. void doFilter(ServletRequest request, ServletResponse response, FilterChain chain): invoked by the container each time a request/response pair is passed through the chain due to a client request for a resource at the end of the chain.
3. void destroy(): invoked by the web container to indicate to a filter that it is being taken out of service.

## What is FilterChain Interface?

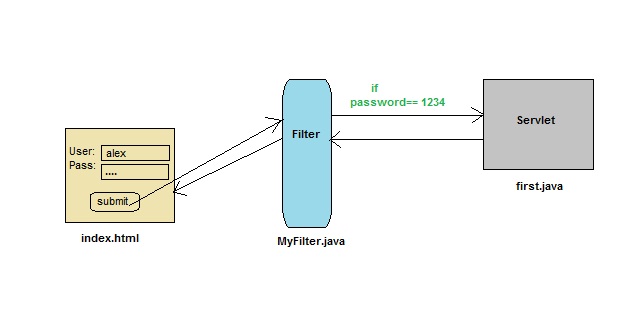
**FilterChain** object is used to invoke the next filter in the chain, or if the calling filter is the last filter in the chain then the rosource at the end of the chain invoked. The resources at the end of Filter chain can either be a target Servlet(in case of request flow) or the Client(in case of response flow) as described in the diagram above.

Declaring a Filter inside Deployment Descriptor

## Example demonstrating Filter usage

In this example we are using Filter to authenticate(check correct username and password). Here **index.html** will ask username and password from the user, **MyFilter** will validate the password entered by the user, if the user has entered "1234" as password, then he will be forwarded to **first**servlet else the index.html will be shown again to the user.

This is exactly what we used to do earlier using two servlet classes earlier, one for validation and the other to Welcome the user. Now we will insert a Filter for validating the user.



**index.html**

<form method="post" action="**first**">

Name:<input type="text" name="user" /><br/>

Password:<input type="text" name="pass" /><br/>

<input type="submit" value="submit" />

</form>

**web.xml**

<web-app..>

<filter>

<filter-name>MyFilter</filter-name>

**<filter-class>MyFilter</filter-class>**

</filter>

<filter-mapping>

<filter-name>MyFilter</filter-name>

**<servlet-name>first</servlet-name>**

</filter-mapping>

<servlet>

<servlet-name>first</servlet-name>

<servlet-class>first</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>first</servlet-name>

<url-pattern>/first</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>

**MyFilter.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class **MyFilter** implements **Filter** {

public void **init**(FilterConfig fc) throws ServletException {}

public void **doFilter**(ServletRequest request, ServletResponse response,

FilterChain chain) throws IOException, ServletException {

PrintWriter out = response.getWriter();

String **pass** = request.getParameter("pass");

if(pass.*equals*("1234"))

{

**chain.doFilter(request, response);**

}

else

{

out.println("You have enter a wrong password");

RequestDispatcher rs = request.getRequestDispatcher("index.html");

rs.include(request, response);

}

}

public void destroy() { }

}

**first.java**

import java.io.\*;

import javax.servlet.\*;

import javax.servlet.http.\*;

public class *first***extends** HttpServlet {

protected void **doPost**(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException

{

response.setContentType("text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

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

out.println("Wellcome "+**user**);

}

}