Servlets

---------------------

**Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).

**Servlet** technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context.

70.4K

3/4 Angular NgRx Entity installation tutorial | Real life implementation

* Servlet is a technology which is used to create a web application.
* Servlet is an API that provides many interfaces and classes including documentation.
* Servlet is an interface that must be implemented for creating any Servlet.
* Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
* Servlet is a web component that is deployed on the server to create a dynamic web page.

### Servlet What is a web application?

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

### CGI (Common Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.



### Disadvantages of CGI

There are many problems in CGI technology:

1. If the number of clients increases, it takes more time for sending the response.
2. For each request, it starts a process, and the web server is limited to start processes.
3. It uses platform dependent language e.g. [C](https://www.javatpoint.com/c-programming-language-tutorial), [C++](https://www.javatpoint.com/cpp-tutorial), [perl](https://www.javatpoint.com/perl-tutorial).

### Advantages of Servlet



There are many advantages of Servlet over CGI. The web container creates threads for handling the multiple requests to the Servlet. Threads have many benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The advantages 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:** [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) manages Servlets, so we don't need to worry about the memory leak, [garbage collection](https://www.javatpoint.com/Garbage-Collection), etc.
4. **Secure:** because it uses java language.

# Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

9M

189

OOPs Concepts in Java

### Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

### Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper
5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

### Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

### Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

### Methods of Servlet interface

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

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

# Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

### 3) init method is invoked

|  |
| --- |
| The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below: |

1. **public** **void** init(ServletConfig config) **throws** ServletException

### 4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

1. **public** **void** service(ServletRequest request, ServletResponse response)
2. **throws** ServletException, IOException

### 5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

1. **public** **void** destroy()

# Steps to create a servlet example

There are given 6 steps to create a **servlet example**. These steps are required for all the servers.

The servlet example can be created by three ways:

1. By implementing Servlet interface,
2. By inheriting GenericServlet class, (or)
3. By inheriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

Here, we are going to use **apache tomcat server** in this example. The steps are as follows:

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Create a deployment descriptor
5. Start the server and deploy the project
6. Access the servlet

### 1)Create a directory structures

The **directory structure** defines that where to put the different types of files so that web container may get the information and respond to the client.

The Sun Microsystem defines a unique standard to be followed by all the server vendors. Let's see the directory structure that must be followed to create the servlet.



As you can see that the servlet class file must be in the classes folder. The web.xml file must be under the WEB-INF folder.

### 2)Create a Servlet

|  |
| --- |
| There are three ways to create the servlet.   1. By implementing the Servlet interface 2. By inheriting the GenericServlet class 3. By inheriting the HttpServlet class   The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as doGet(), doPost, doHead() etc. |
| In this example we are going to create a servlet that extends the HttpServlet class. In this example, we are inheriting the HttpServlet class and providing the implementation of the doGet() method. Notice that get request is the default request. |

**DemoServlet.java**

**import** javax.servlet.http.\*;

**import** javax.servlet.\*;

**import** java.io.\*;

**public** **class** DemoServlet **extends** HttpServlet{

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

**throws** ServletException,IOException

{

res.setContentType("text/html");//setting the content type

PrintWriter pw=res.getWriter();//get the stream to write the data

//writing html in the stream

pw.println("<html><body>");

pw.println("Welcome to servlet");

pw.println("</body></html>");

pw.close();//closing the stream

}}

# ServletRequest Interface

An object of ServletRequest is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header informations, attributes etc.

### Example of ServletRequest to display the name of the user

In this example, we are displaying the name of the user in the servlet. For this purpose, we have used the getParameter method that returns the value for the given request parameter name.

**index.html**

<form action="welcome" method="get">

Enter your name<input type="text" name="name"><br>

<input type="submit" value="login">

</form>

**DemoServ.java**

**import** javax.servlet.http.\*;

**import** javax.servlet.\*;

**import** java.io.\*;

**public** **class** DemoServ **extends** HttpServlet{

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

**throws** ServletException,IOException

{

res.setContentType("text/html");

PrintWriter pw=res.getWriter();

String name=req.getParameter("name");//will return value

pw.println("Welcome "+name);

pw.close();

}}

# RequestDispatcher in Servlet

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

There are two methods defined in the RequestDispatcher interface.

### Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

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



As you see in the above figure, response of second servlet is sent to the client. Response of the first servlet is not displayed to the user.



|  |
| --- |
| As you can see in the above figure, response of second servlet is included in the response of the first servlet that is being sent to the client. |

### Example of RequestDispatcher interface

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

* **index.html file:** for getting input from the user.
* **Login.java file:** a servlet class for processing the response. If password is servet, it will forward the request to the welcome servlet.
* **WelcomeServlet.java file:** a servlet class for displaying the welcome message.
* **web.xml file:** a deployment descriptor file that contains the information about the servlet.



**index.html**

<form action="servlet1" method="post">

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

Password:<input type="password" name="userPass"/><br/>

<input type="submit" value="login"/>

</form>

**Login.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** Login **extends** HttpServlet {

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

**throws** ServletException, IOException {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

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

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

**if**(p.equals("servlet"){

        RequestDispatcher rd=request.getRequestDispatcher("servlet2");

        rd.forward(request, response);

    }

**else**{

        out.print("Sorry UserName or Password Error!");

        RequestDispatcher rd=request.getRequestDispatcher("/index.html");

        rd.include(request, response);

        }

    }

}

**WelcomeServlet.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** WelcomeServlet **extends** HttpServlet {

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

**throws** ServletException, IOException {

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

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

    out.print("Welcome "+n);

    }

}

**web.xml**

<web-app>

 <servlet>

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

    <servlet-**class**>Login</servlet-**class**>

  </servlet>

  <servlet>

    <servlet-name>WelcomeServlet</servlet-name>

    <servlet-**class**>WelcomeServlet</servlet-**class**>

  </servlet>

  <servlet-mapping>

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

    <url-pattern>/servlet1</url-pattern>

  </servlet-mapping>

  <servlet-mapping>

    <servlet-name>WelcomeServlet</servlet-name>

    <url-pattern>/servlet2</url-pattern>

  </servlet-mapping>

  <welcome-file-list>

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

  </welcome-file-list>

</web-app>

# SendRedirect in servlet

The **sendRedirect()** method of **HttpServletResponse** interface can be used to redirect response to another resource, it may be servlet, jsp or html file.

It accepts relative as well as absolute URL.

It works at client side because it uses the url bar of the browser to make another request. So, it can work inside and outside the server.

## Difference between forward() and sendRedirect() method

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

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

### Full example of sendRedirect method in servlet

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

*DemoServlet.java*

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** DemoServlet **extends** HttpServlet{

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

**throws** ServletException,IOException

{

res.setContentType("text/html");

PrintWriter pw=res.getWriter();

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

pw.close();

}}

### Creating custom google search using sendRedirect

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

*index.html*

<!DOCTYPE html**>**

**<html>**

**<head>**

**<meta** charset="ISO-8859-1"**>**

**<title>**sendRedirect example**</title>**

**</head>**

**<body>**

**<form** action="MySearcher"**>**

**<input** type="text" name="name"**>**

**<input** type="submit" value="Google Search"**>**

**</form>**

**</body>**

**</html>**

*MySearcher.java*

**import** java.io.IOException;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**public** **class** MySearcher **extends** HttpServlet {

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

**throws** ServletException, IOException {

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

        response.sendRedirect("https://www.google.co.in/#q="+name);

    }

}

# Session Tracking in Servlets

**Session** simply means a particular interval of time.

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

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

HTTP is stateless that means each request is considered as the new request. It is shown in the figure given below:M

275

Exception Handling in Java -int



### Why use Session Tracking?

**To recognize the user** It is used to recognize the particular user.

### Session Tracking Techniques

There are four techniques used in Session tracking:

1. **Cookies**
2. **Hidden Form Field**
3. **URL Rewriting**
4. **HttpSession**

# 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, cookie is added with request by default. Thus, we recognize the user as the old user.



### Simple example of Servlet Cookies

In this example, we are storing the name of the user in the cookie object and accessing it in another servlet. As we know well that session corresponds to the particular user. So if you access it from too many browsers with different values, you will get the different value.



### index.html

<form action="servlet1" method="post">

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

<input type="submit" value="go"/>

</form>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

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

**try**{

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

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

    out.print("Welcome "+n);

    Cookie ck=**new** Cookie("uname",n);//creating cookie object

    response.addCookie(ck);//adding cookie in the response

    //creating submit button

    out.print("<form action='servlet2'>");

    out.print("<input type='submit' value='go'>");

    out.print("</form>");

    out.close();

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

  }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

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

**try**{

    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    Cookie ck[]=request.getCookies();

    out.print("Hello "+ck[0].getValue());

    out.close();

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

    }

}

### web.xml

<web-app>

<servlet>

<servlet-name>s1</servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s1</servlet-name>

<url-pattern>/servlet1</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>s2</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s2</servlet-name>

<url-pattern>/servlet2</url-pattern>

</servlet-mapping>

</web-app>

# 2) 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.

Let's see the code to store value in hidden field.

1. <input type="hidden" name="uname" value="Swamy">

### Example of using Hidden Form Field

In this example, we are storing the name of the user in a hidden textfield and getting that value from another servlet.



### index.html

<form action="servlet1">

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

<input type="submit" value="go"/>

</form>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

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

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

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

        out.print("Welcome "+n);

        //creating form that have invisible textfield

        out.print("<form action='servlet2'>");

        out.print("<input type='hidden' name='uname' value='"+n+"'>");

        out.print("<input type='submit' value='go'>");

        out.print("</form>");

        out.close();

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

    }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        //Getting the value from the hidden field

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

        out.print("Hello "+n);

        out.close();

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

    }

}

### web.xml

<web-app>

<servlet>

<servlet-name>s1</servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s1</servlet-name>

<url-pattern>/servlet1</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>s2</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s2</servlet-name>

<url-pattern>/servlet2</url-pattern>

</servlet-mapping>

</web-app>

# 3)URL Rewriting

In URL rewriting, we append a token or identifier to the URL of the next Servlet or the next resource. We can send parameter name/value pairs using the following format:

url?name1=value1&name2=value2&??

A name and a value is separated using an equal = sign, a parameter name/value pair is separated from another parameter using the ampersand(&). When the user clicks the hyperlink, the parameter name/value pairs will be passed to the server. From a Servlet, we can use getParameter() method to obtain a parameter value.



### Example of using URL Rewriting

In this example, we are maintaning the state of the user using link. For this purpose, we are appending the name of the user in the query string and getting the value from the query string in another page.

OOPs Concepts in Java

**Next**

**Stay**

### index.html

<form action="servlet1">

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

<input type="submit" value="go"/>

</form>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

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

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

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

        out.print("Welcome "+n);

        //appending the username in the query string

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

        out.close();

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

    }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        //getting value from the query string

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

        out.print("Hello "+n);

        out.close();

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

}

### web.xml

<web-app>

<servlet>

<servlet-name>s1</servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s1</servlet-name>

<url-pattern>/servlet1</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>s2</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s2</servlet-name>

<url-pattern>/servlet2</url-pattern>

</servlet-mapping>

</web-app>

# 4) HttpSession interface

In such case, container creates a session id for each user.The container uses this id to identify the particular user.An object of HttpSession can be used to perform two tasks:

1. bind objects
2. view and manipulate information about a session, such as the session identifier, creation time, and last accessed time.



### How to get the HttpSession object ?

The HttpServletRequest interface provides two methods to get the object of HttpSession:

1. **public HttpSession getSession():**Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):**Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

### Example of using HttpSession

In this example, we are setting the attribute in the session scope in one servlet and getting that value from the session scope in another servlet. To set the attribute in the session scope, we have used the setAttribute() method of HttpSession interface and to get the attribute, we have used the getAttribute method.

### index.html

<form action="servlet1">

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

<input type="submit" value="go"/>

</form>

### FirstServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** FirstServlet **extends** HttpServlet {

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

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

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

        out.print("Welcome "+n);

        HttpSession session=request.getSession();

        session.setAttribute("uname",n);

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

        out.close();

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

    }

}

### SecondServlet.java

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**public** **class** SecondServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**try**{

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(**false**);

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

        out.print("Hello "+n);

        out.close();

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

    }

}

### web.xml

<web-app>

<servlet>

<servlet-name>s1</servlet-name>

<servlet-**class**>FirstServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s1</servlet-name>

<url-pattern>/servlet1</url-pattern>

</servlet-mapping>

<servlet>

<servlet-name>s2</servlet-name>

<servlet-**class**>SecondServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s2</servlet-name>

<url-pattern>/servlet2</url-pattern>

</servlet-mapping>

</web-app>

# Servlet Filter

A **filter** is an object that is invoked at the 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.



#### **Note: Unlike Servlet, One filter doesn't have dependency on another filter.**

### Usage of Filter

* recording all incoming requests
* logs the IP addresses of the computers from which the requests originate
* conversion
* data compression
* encryption and decryption
* input validation etc.

### Advantage of Filter

1. Filter is pluggable.
2. One filter don't have dependency onto another resource.
3. Less Maintenance

### Filter API

Like servlet filter have its own API. The javax.servlet package contains the three interfaces of Filter API.

1. Filter
2. FilterChain
3. FilterConfig

### 1) Filter interface

For creating any filter, you must implement the Filter interface. Filter interface provides the life cycle methods for a filter.

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

### 2) FilterChain interface

The object of FilterChain is responsible to invoke the next filter or resource in the chain.This object is passed in the doFilter method of Filter interface.The FilterChain interface contains only one method:

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

### How to define Filter

We can define filter same as servlet. Let's see the elements of filter and filter-mapping.

<web-app>

<filter>

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

<filter-**class**>...</filter-**class**>

</filter>

<filter-mapping>

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

<url-pattern>...</url-pattern>

</filter-mapping>

</web-app>

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

### Simple Example of Filter

In this example, we are simply displaying information that filter is invoked automatically after the post processing of the request.

### index.html

<a href="servlet1">click here</a>

**MyFilter.java**

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.\*;

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

**public** **void** init(FilterConfig arg0) **throws** ServletException {}

**public** **void** doFilter(ServletRequest req, ServletResponse resp,

    FilterChain chain) **throws** IOException, ServletException {

    PrintWriter out=resp.getWriter();

    out.print("filter is invoked before");

    chain.doFilter(req, resp);//sends request to next resource

    out.print("filter is invoked after");

    }

**public** **void** destroy() {}

}

### HelloServlet.java

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.\*;

**public** **class** HelloServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        out.print("<br>welcome to servlet<br>");

    }

}

**web.xml**

|  |
| --- |
| For defining the filter, filter element of web-app must be defined just like servlet. |

<web-app>

<servlet>

<servlet-name>s1</servlet-name>

<servlet-**class**>HelloServlet</servlet-**class**>

</servlet>

<servlet-mapping>

<servlet-name>s1</servlet-name>

<url-pattern>/servlet1</url-pattern>

</servlet-mapping>

<filter>

<filter-name>f1</filter-name>

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

</filter>

<filter-mapping>

<filter-name>f1</filter-name>

<url-pattern>/servlet1</url-pattern>

</filter-mapping>

</web-app>

ServletRequestListener

This interface is used for receiving notification events about requests coming into and going out of scope of a web application. Notifications will be generated when a request is first comming into scope and when it get out of scope. The ServletRequestListener can be registerd by @WebListener annotation, adding the listener to the servlet descriptor or programmatically adding a listener with .addListener() to the servlet context. In this example we use the @WebListener annotation.

ServletRequestListener receives notifications for ServletRequest init and destroy

In order to listen to the lifecyle event of initialization or destroying of a ServletRequest we need to implement the javax.servlet.ServletRequestListener interface. This interface lets us listen to the following events, the names speak for themselves.

* requestInitialized()
* requestDestroyed()

To register a listener we can add the @WebListener, define the listener in the servlet descriptor (web.xml) or programatigally add it to the servlet context. In this example we choose to add the listener through the @WebListener annotation.

**package** com.memorynotfound;

**import** javax.servlet.\*;

**import** javax.servlet.ServletRequestListener;

**import** javax.servlet.annotation.WebListener;

@WebListener

**public** **class** ServletRequestLogger **implements** ServletRequestListener {

@Override

**public** **void** requestInitialized(ServletRequestEvent event) {

System.out.println(**"request initialized, request by: "** + event.getServletRequest().getRemoteAddr());

}

@Override

**public** **void** requestDestroyed(ServletRequestEvent event) {

System.out.println(**"request destroyed"**);

}

}

**Note:**If you prefer the web.xml **servlet descriptor** over the @WebListener annotation you can add the context listener as follows:

<**web-app** xmlns=**"http://xmlns.jcp.org/xml/ns/javaee"**

xmlns:xsi=**"http://www.w3.org/2001/XMLSchema-instance"**

xsi:schemaLocation=**"http://xmlns.jcp.org/xml/ns/javaee**

**http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"** version=**"3.1"**>

<**listener**>

<**listener-class**>com.memorynotfound.ServletRequestLogger</**listener-class**>

</**listener**>

</**web-app**>

# HttpSessionEvent and HttpSessionListener

|  |
| --- |
| The HttpSessionEvent is notified when session object is changed.  The corresponding Listener interface for this event is HttpSessionListener. |

|  |
| --- |
| We can perform some operations at this event such as counting total and current logged-in users,  maintaing a log of user details such as login time, logout time etc. |

### Methods of HttpSessionListener interface

|  |
| --- |
| There are two methods declared in the HttpSessionListener interface which must be implemented  by the servlet programmer to perform some action.   1. **public void sessionCreated(HttpSessionEvent e)**: is invoked when session object is created. 2. **public void sessionDestroyed(ServletContextEvent e)**: is invoked when session is invalidated. |

### Example of HttpSessionEvent and HttpSessionListener to count total and current logged-in users

|  |
| --- |
| In this example, are counting the total and current logged-in users. For this purpose,  we have created three files:   1. **index.html**: to get input from the user. 2. **MyListener.java**: A listener class that counts total and current logged-in users and stores this information in ServletContext object as an attribute. 3. **First.java**: A Servlet class that creates session and prints the total and current logged-in users. 4. **Logout.java**: A Servlet class that invalidates session. |

#### index.html

<form action="servlet1">

Name:<input type="text" name="username"><br>

Password:<input type="password" name="userpass"><br>

<input type="submit" value="login"/>

</form>

#### MyListener.java

**import** javax.servlet.ServletContext;

**import** javax.servlet.http.HttpSessionEvent;

**import** javax.servlet.http.HttpSessionListener;

**public** **class** CountUserListener **implements** HttpSessionListener{

    ServletContext ctx=**null**;

**static** **int** total=0,current=0;

**public** **void** sessionCreated(HttpSessionEvent e) {

    total++;

    current++;

    ctx=e.getSession().getServletContext();

    ctx.setAttribute("totalusers", total);

    ctx.setAttribute("currentusers", current);

    }

**public** **void** sessionDestroyed(HttpSessionEvent e) {

        current--;

        ctx.setAttribute("currentusers",current);

    }

}

#### First.java

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.ServletContext;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.servlet.http.HttpSession;

**public** **class** First **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request,

 HttpServletResponse response)

**throws** ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        String n=request.getParameter("username");

        out.print("Welcome "+n);

        HttpSession session=request.getSession();

        session.setAttribute("uname",n);

        //retrieving data from ServletContext object

        ServletContext ctx=getServletContext();

**int** t=(Integer)ctx.getAttribute("totalusers");

**int** c=(Integer)ctx.getAttribute("currentusers");

        out.print("<br>total users= "+t);

        out.print("<br>current users= "+c);

        out.print("<br><a href='logout'>logout</a>");

        out.close();

    }

}

#### Logout.java

**import** java.io.IOException;

**import** java.io.PrintWriter;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.servlet.http.HttpSession;

**public** **class** LogoutServlet **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request,

 HttpServletResponse response)

**throws** ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        HttpSession session=request.getSession(**false**);

        session.invalidate();//invalidating session

        out.print("You are successfully logged out");

        out.close();

    }

}

# ServletContextEvent and ServletContextListener

The ServletContextEvent is notified when web application is deployed on the server.

If you want to perform some action at the time of deploying the web application such as creating database connection, creating all the tables of the project etc, you need to implement ServletContextListener interface and provide the implementation of its methods.

### Constructor of ServletContextEvent class

There is only one constructor defined in the ServletContextEvent class. The web container creates the instance of ServletContextEvent after the ServletContext instance.

ServletContextEvent(ServletContext e)

### Method of ServletContextEvent class

There is only one method defined in the ServletContextEvent class:

**public ServletContext getServletContext()**: returns the instance of ServletContext.

### Methods of ServletContextListener interface

There are two methods declared in the ServletContextListener interface which must be implemented by the servlet programmer to perform some action such as creating database connection etc.

1. **public void contextInitialized(ServletContextEvent e)**: is invoked when application is deployed on the server.
2. **public void contextDestroyed(ServletContextEvent e)**: is invoked when application is undeployed from the server.

### Example of ServletContextEvent and ServletContextListener

In this example, we are retrieving the data from the emp32 table. To serve this, we have created the connection object in the listener class and used the connection object in the servlet.

**index.html**

<a href="servlet1">fetch records</a>

**MyListener.java**

**import** javax.servlet.\*;

**import** java.sql.\*;

**public** **class** MyListener **implements** ServletContextListener{

**public** **void** contextInitialized(ServletContextEvent event) {

**try**{

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//storing connection object as an attribute in ServletContext

ServletContext ctx=event.getServletContext();

ctx.setAttribute("mycon", con);

}**catch**(Exception e){e.printStackTrace();}

}

**public** **void** contextDestroyed(ServletContextEvent arg0) {}

}

**MyListener.java**

**import** java.io.\*;

**import** javax.servlet.\*;

**import** javax.servlet.http.\*;

**import** java.sql.\*;

**public** **class** FetchData **extends** HttpServlet {

**public** **void** doGet(HttpServletRequest request, HttpServletResponse response)

**throws** ServletException, IOException {

response.setContentType("text/html");

PrintWriter out = response.getWriter();

**try**{

//Retrieving connection object from ServletContext object

ServletContext ctx=getServletContext();

Connection con=(Connection)ctx.getAttribute("mycon");

//retieving data  from emp32 table

PreparedStatement ps=con.prepareStatement("select \* from emp32",

ResultSet.TYPE\_SCROLL\_SENSITIVE,ResultSet.CONCUR\_UPDATABLE);

ResultSet rs=ps.executeQuery();

**while**(rs.next()){

out.print("<br>"+rs.getString(1)+" "+rs.getString(2));

}

con.close();

}**catch**(Exception e){e.printStackTrace();}

out.close();

}

}

### Example of ServletContextListener to create table of a project

In this example, we are creating table of the project. So we don't need to create all the tables manually in the database.

**MyListener.java**

**import** javax.servlet.\*;

**import** java.sql.\*;

**public** **class** MyListener **implements** ServletContextListener{

**public** **void** contextInitialized(ServletContextEvent arg0) {

**try**{

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("

jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

String query="create table emp32(id number(10),name varchar2(40))";

PreparedStatement ps=con.prepareStatement(query);

ps.executeUpdate();

System.out.println(query);

}**catch**(Exception e){e.printStackTrace();}

}

**public** **void** contextDestroyed(ServletContextEvent arg0) {

System.out.println("project undeployed");

}

}