

Servlet

Java Servlets are programs that run on a Web or Application server and act as a middle layer between requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.

Using Servlets, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

Properties of Servlets :

- Servlets work on the server-side.
- Servlets capable of handling complex request obtained from web server.

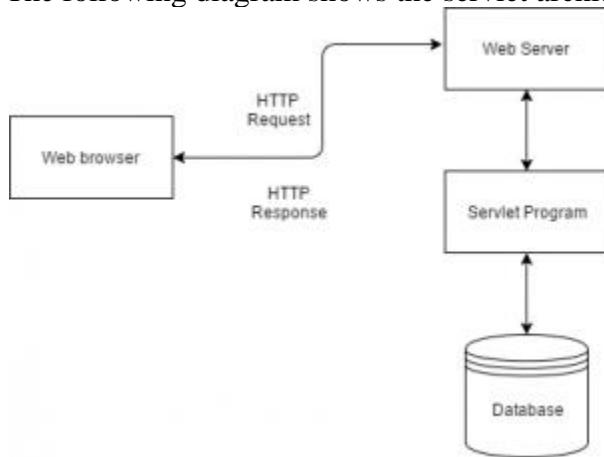
Execution of Servlets :

Execution of Servlets involves the six basic steps:

1. The clients send the request to the web server.
2. The web server receives the request.
3. The web server passes the request to the corresponding servlet.
4. The servlet processes the request and generate the response in the form of output.
5. The servlet send the response back to the web server.
6. The web server sends the response back to the client and the client browser displays it on the screen.
- 7.

Servlet Architecture

The following diagram shows the servlet architecture:



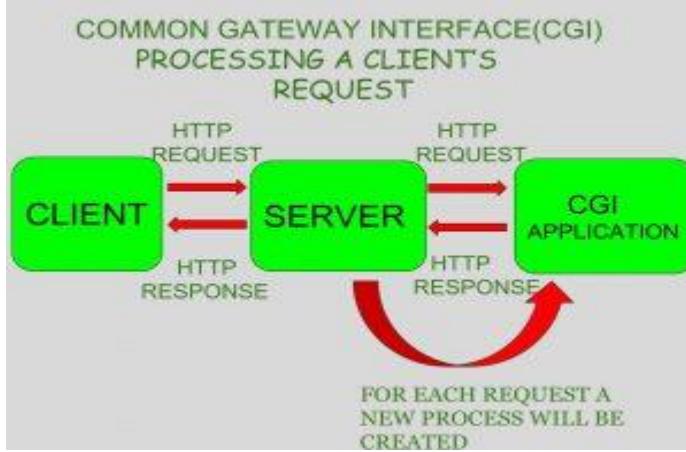
CGI

CGI is actually an external application which is written by using any of the programming languages like **C** or **C++** and this is responsible for processing client requests and generating dynamic content.

In CGI application, when a client makes a request to access dynamic Web pages, the Web server performs the following operations :

- It first locates the requested web page *i.e* the required CGI application using URL.
- It then creates a new process to service the client's request.

- Invokes the CGI application within the process and passes the request information to the server.
- Collects the response from CGI application.
- Destroys the process, prepares the HTTP response and sends it to the client.



So, in **CGI** server has to create and destroy the process for every request. Its easy to understand that this approach is applicable for handling few clients but as the number of client increases the workload on the server increases and so the time taken to process requests increases.

Advantages of Servlet over CGI

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 manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.
4. **Secure:** because it uses java language.

Features of Servlet

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

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.

Various classes and interfaces present in these packages are:

COMPONENT	TYPE	PACKAGE
Servlet	Interface	javax.servlet.*
ServletRequest	Interface	javax.servlet.*
ServletResponse	Interface	javax.servlet.*
GenericServlet	Class	javax.servlet.*
HttpServlet	Class	javax.servlet.http.*
HttpServletRequest	Interface	javax.servlet.http.*
HttpServletResponse	Interface	javax.servlet.http.*
Filter	Interface	javax.servlet.*
ServletConfig	Interface	javax.servlet.*

Servlet Lifecycle

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

- The servlet is initialized by calling the **init()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

The init() Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this –

```
public void init() throws ServletException {  
    // Initialization code...  
}
```

The service() Method

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method –

```
public void service(ServletRequest request, ServletResponse response)  
throws ServletException, IOException {
```

```
}
```

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    // Servlet code  
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)  
throws ServletException, IOException {  
    // Servlet code  
}
```

The destroy() Method

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection.

```
public void destroy() {  
    // Finalization code...  
}
```

Ex:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

// Extend HttpServlet class
public class HelloWorld extends HttpServlet {

    private String message;

    public void init() throws ServletException {
        // Do required initialization
        message = "Hello World";
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Set response content type
        response.setContentType("text/html");

        // Actual logic goes here.
        PrintWriter out = response.getWriter();
        out.println("<h1>" + message + "</h1>");
    }

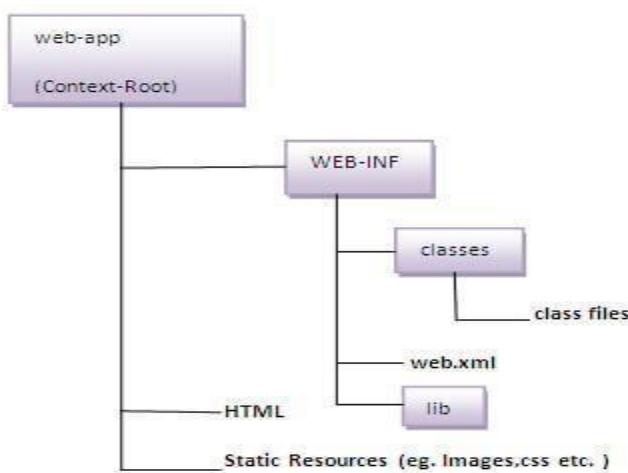
    public void destroy() {
        // do nothing.
    }
}
```

Steps to create a servlet example

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.

```

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
}}
```

3)Compile the servlet

For compiling the Servlet, jar file is required to be loaded. Different Servers provide different jar files:

Jar file	Server
1) servlet-api.jar	Apache Tomcat
2) weblogic.jar	Weblogic
3) javaee.jar	Glassfish
4) javaee.jar	JBoss

Two ways to load the jar file

1. set classpath
2. paste the jar file in JRE/lib/ext folder

Put the java file in any folder. After compiling the java file, paste the class file of servlet in **WEB-INF/classes** directory.

Create the deployment descriptor (web.xml file)

The **deployment descriptor** is an xml file, from which Web Container gets the information about the servet to be invoked.

The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull. There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

web.xml file

```
<web-app>
<servlet>
<servlet-name>sonoojaival</servlet-name>
<servlet-class>DemoServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>sonoojaival</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

Description of the elements of web.xml file

There are too many elements in the web.xml file. Here is the illustration of some elements that is used in the above web.xml file. The elements are as follows:

<web-app> represents the whole application.

<servlet> is sub element of **<web-app>** and represents the servlet.

<servlet-name> is sub element of **<servlet>** represents the name of the servlet.

<servlet-class> is sub element of **<servlet>** represents the class of the servlet.

<servlet-mapping> is sub element of **<web-app>**. It is used to map the servlet.

<url-pattern> is sub element of **<servlet-mapping>**. This pattern is used at client side to invoke the servlet.

5)Start the Server and deploy the project

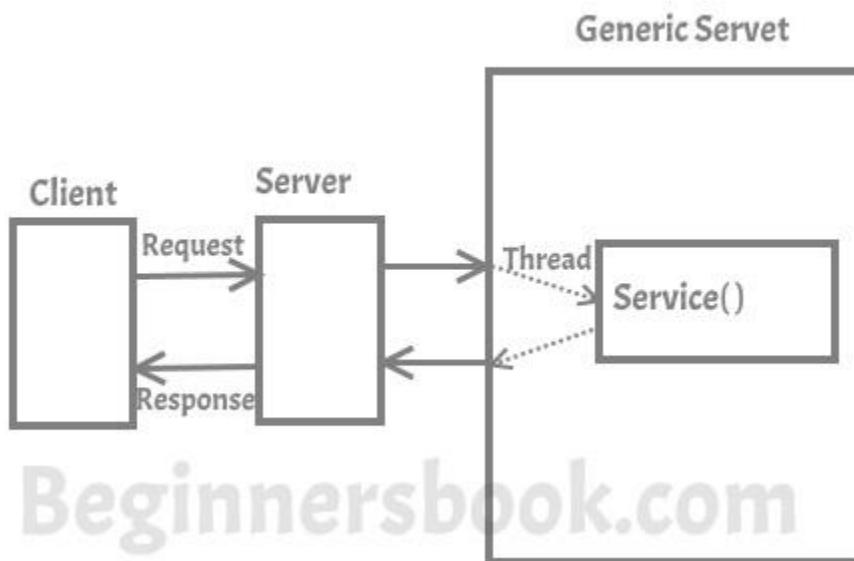
Generic Servlet

While 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:

```
public abstract void service(ServletRequest request, ServletResponse response)
    throws ServletException, 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.



- Generic Servlet is protocol independent(**i.e**)it can handle all types of protocols like **http, ftp, smtp** etc.
- GenericServlet class is direct subclass of **Servlet Interface**.
- GenericServlet is an abstract class which implements **Servlet, ServletConfig and java.io.Serializable**interfaces.
- GenericServlet belongs to **javax.servlet** package.
- Generic Servlet supports **only service() method**. Extending class must override **public abstract void service(ServletRequest req,ServletResponse res)** method.
- GenericServlet implements **ServletConfig** interface and provides way to accept initialization parameter passed to Servlet from web.xml e.g. by using **getInitParamter()**.

There are many methods in GenericServlet class. They are as follows:

1. **public void init(ServletConfig config)** is used to initialize the servlet.
2. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
3. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
4. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
5. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
6. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
7. **public ServletContext getServletContext()** returns the object of ServletContext.
8. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
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 and a stack trace.

Eg:

```

import java.io.*;
import javax.servlet.*;
public class First extends GenericServlet{
    public void service(ServletRequest req,ServletResponse res)
    throws IOException,ServletException{
        res.setContentType("text/html");
        PrintWriter out=res.getWriter();
        out.print("<html><body>");
    }
}

```

```

out.print("<b>hello generic servlet</b>");
out.print("</body></html>");
}
}

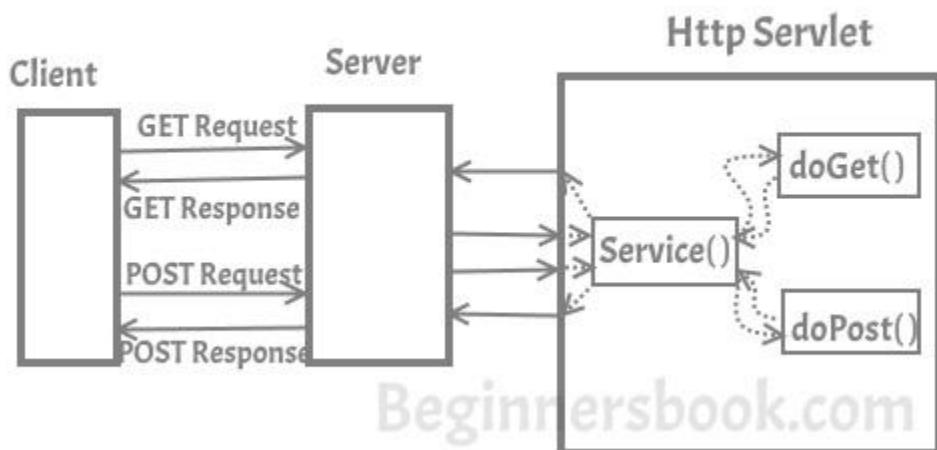
```

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.



- HttpServlet is protocol dependent. It supports only **http** protocol.
- HttpServlet class is the direct subclass of **Generic Servlet**.
- HttpServlet is an abstract class which extends **GenericServlet** and implements **java.io.Serializable** interface.
- HttpServlet belongs to **javax.servlet.http** package.
- HttpServlet overrides **service()** method of Generic Servlet and provides callback on **doXXX(HttpServletRequest request, HttpServletResponse)** method whenever it receives HTTP request, it supports **doGet(), doPost(), doPut(), doDelete(), doHead(), doTrace(), doOptions()** methods.
- HttpServlet has two service methods **public void service(ServletRequest req, ServletResponse res)** and **protected void service(HttpServletRequest req, HttpServletResponse res)** All the request first goes to the **public service()** method, which wrap into Http Objects and calls **protected service()** method

Eg

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HttpServletExample extends HttpServlet{

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
    {
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<h2>Http Servlet Example!!!</h2>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Request Dispatcher

Servlet **Request Dispatcher** is an interface whose implementation defines that an object can dispatch requests to any resource (such as HTML, Image, JSP, Servlet etc.) on the server.

Another advantage of this interface is that it is used in two cases:

- To **include** the response of one Servlet into another (i.e. the client gets the response of both Servlets)
- To **forward** the client request to another Servlet to honor the request (i.e. the client calls a Servlet but the response to client is given by another Servlet)

This interface is placed in the `javax.servlet` package and has the following two methods:

Method	Description
<pre>public void forward(ServletRequest request, ServletResponse response) throws IOException, ServletException</pre>	This method <u>forwards</u> a request from a Servlet to another resource (i.e. Servlet to Servlet, Servlet to JSP, Servlet to HTML etc.) on the server and there is no return type
<pre>public void include(ServletRequest request, ServletResponse response) throws ServletException, IOException</pre>	This method <u>includes</u> the content of a resource in the response and there is no return type

1.1 Difference between forward() and include()

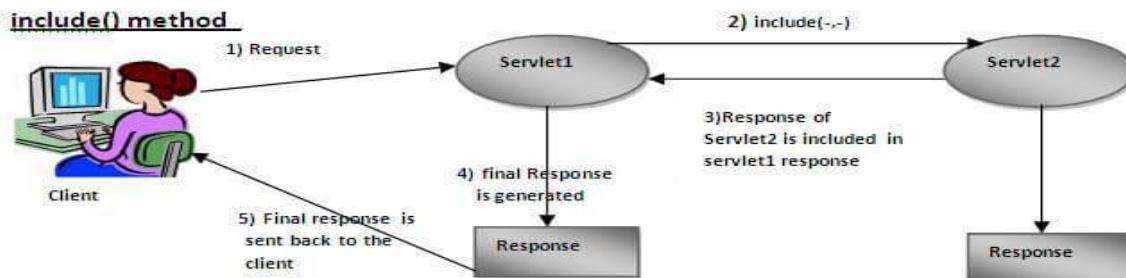
Both the methods are a part of Request Dispatcher interface. These methods will accept an object of the Servlet request and response interface. The main *difference* is that when a programmer uses forward, the control is transferred to the next Servlet or JSP the application is calling while in the case of include, the control retains with the current Servlet and it just includes the processing done by the calling of Servlet or the JSP.

1.1.1 Request Dispatcher forward() Method

In the below conceptual figure, the response generated by the Servlet2 is visible to the user, but the response generated by the Servlet1 is not visible to the user.

1.1.2 Request Dispatcher include() Method

In the `include` method concept, the response of the Servlet2 is included in the response of the Servlet1 and the generated final response is sent back to the client



Creating object of RequestDispatcher

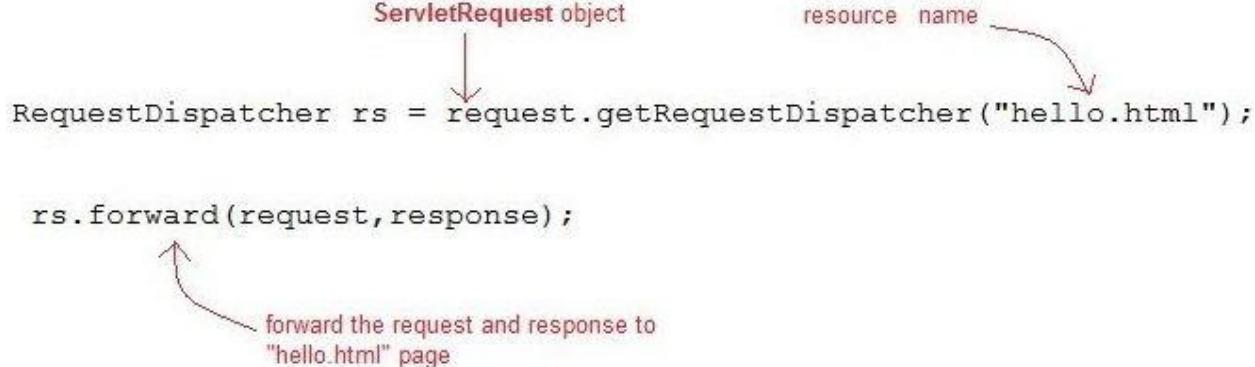
The `getRequestDispatcher()` method of the **Servlet Request** interface returns the object of the **Request Dispatcher**.

Syntax

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

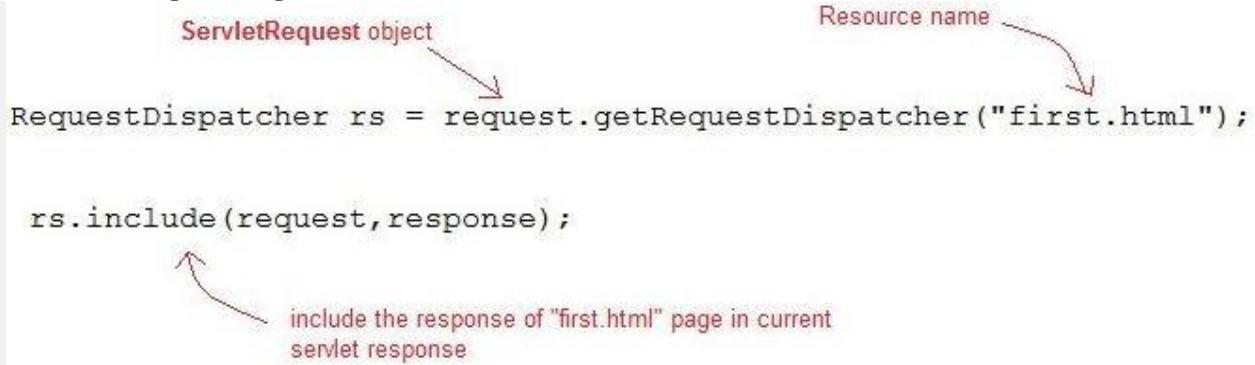
After creating the `RequestDispatcher` object, developers will call the `forward()` or `include()` method as per the application's requirement.

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



Or

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



In this example, we are validating the password entered by the user. If password is servet, 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>

```

Session in Servlet

In web terminology, a session is simply the limited interval of time in which two systems communicate with each other. The two systems can share a client-server or a peer-to-peer relationship. However, in Http protocol, the state of the communication is not maintained. Hence, the web applications that work on http protocol use several different technologies that comprise **Session Tracking**, which means maintaining the state (data) of the user, in order to recognize him/her.

There are following three ways to maintain session between web client and web server –

Cookies

A webserver can assign a unique session ID as a cookie to each web client and for subsequent requests from the client they can be recognized using the received cookie.

This may not be an effective way because many times browser does not support a cookie, so I would not recommend to use this procedure to maintain the sessions.

Hidden Form Fields

A web server can send a hidden HTML form field along with a unique session ID as follows –

```
<input type = "hidden" name = "sessionid" value = "12345">
```

This entry means that, when the form is submitted, the specified name and value are automatically included in the GET or POST data. Each time when web browser sends request back, then session_id value can be used to keep the track of different web browsers.

This could be an effective way of keeping track of the session but clicking on a regular (<A HREF...>) hypertext link does not result in a form submission, so hidden form fields also cannot support general session tracking.

URL Rewriting

URL rewriting is a better way to maintain sessions and it works even when browsers don't support cookies. The drawback of URL re-writing is that you would have to generate every URL dynamically to assign a session ID, even in case of a simple static HTML page.

For example, with <http://tutorialspoint.com/file.htm;sessionid = 12345>, the session identifier is attached as sessionid = 12345 which can be accessed at the web server to identify the client.

Cookies

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users –

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

Types of Cookies

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.

- Servlet Cookies Methods
- Following is the list of useful methods which you can use while manipulating cookies in servlet.

Sr.No.	Method & Description
1	public void setDomain(String pattern) This method sets the domain to which cookie applies, for example tutorialspoint.com.
2	public String getDomain() This method gets the domain to which cookie applies, for example tutorialspoint.com.
3	public void setMaxAge(int expiry) This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	public int getMaxAge() This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	public String getName() This method returns the name of the cookie. The name cannot be changed after creation.
6	public void setValue(String newValue) This method sets the value associated with the cookie
7	public String getValue() This method gets the value associated with the cookie.
8	public void setPath(String uri) This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.

9	public String getPath() This method gets the path to which this cookie applies.
10	public void setSecure(boolean flag) This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	public void setComment(String purpose) This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie to the user.
12	public String getComment() This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

Setting Cookies with Servlet

Setting cookies with servlet involves three steps –

(1) Creating a Cookie object – You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key", "value");
```

Keep in mind, neither the name nor the value should contain white space or any of the following characters –

```
[ ] ( ) = , " / ? @ : ;
```

(2) Setting the maximum age – You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

```
cookie.setMaxAge(60 * 60 * 24);
```

(3) Sending the Cookie into the HTTP response headers – You use response.addCookie to add cookies in the HTTP response header as follows –

```
response.addCookie(cookie);
```

Example of Cookies in java servlet

index.html

```
<form action="login">
User Name:<input type="text" name="userName"/><br/>
Password:<input type="password" name="userPassword"/><br/>
<input type="submit" value="submit"/>
</form>
```

```
MyServlet1.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet1 extends HttpServlet
{
    public void 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=new Cookie("userName",name);
            Cookie c2=new Cookie("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.*;
public class MyServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response){
        try{
            response.setContentType("text/html");
            PrintWriter pwriter = response.getWriter();

            //Reading cookies
            Cookie c[]=request.getCookies();
            //Displaying User name value from cookie
            pwriter.print("Name: "+c[1].getValue());
        }
    }
}
```

```

//Displaying user password value from cookie
pwrtwr.print("Password: "+c[2].getValue());

pwrtwr.close();
}catch(Exception exp){
    System.out.println(exp);
}
}
}

```

web.xml

```

<web-app>
<display-name>BeginnersBookDemo</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>

```

Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

Hidden Form Field

Hidden form field can also be used to store session information for a particular client. In case of hidden form field a hidden field is used to store client state. In this case user information is stored in hidden field value and retrieved from another servlet.

Advantages :

- Does not have to depend on browser whether the cookie is disabled or not.
- Inserting a simple HTML Input field of type hidden is required. Hence, its easier to implement.

Disadvantage :

- Extra form submission is required on every page. This is a big overhead.
- Only textual information can be used.

Real application of hidden form field

It is widely used in comment form of a website. In such case, we store page id or page name in the hidden field so that each page can be uniquely identified.

Eg

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);}
}
}

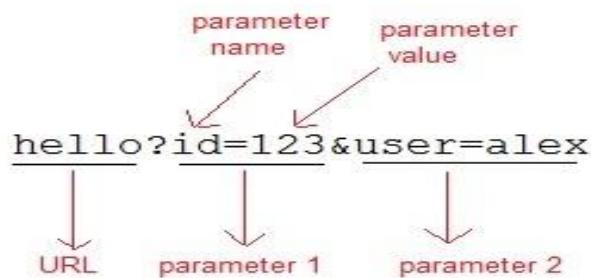
web.xml
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
</web-app>
```

|

URL Rewriting

If the client has disabled cookies in the browser then session management using cookie wont work. In that case **URL Rewriting** can be used as a backup. **URL rewriting** will always work. In URL rewriting, a token(parameter) is added at the end of the URL. The token consist of name/value pair seperated by an **equal(=)** sign.

For Example:



When the User clicks on the URL having parameters, the request goes to the **Web Container** with extra bit of information at the end of URL. The **Web Container** will fetch the extra part of the requested URL and use it for session management.

The `getParameter()` method is used to get the parameter value at the server side.

Advantage of URL Rewriting

1. It will always work whether cookie is disabled or not (browser independent).
2. Extra form submission is not required on each pages.

Disadvantage of URL Rewriting

1. It will work only with links.
2. It can send Only textual information.

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>
```

```

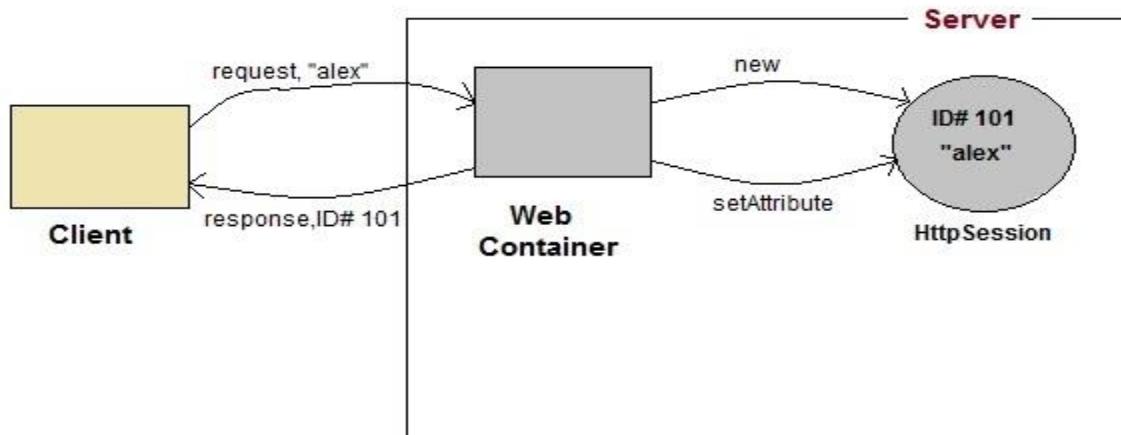
<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>
<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
</web-app>

```

HttpSession

HttpSession object is used to store entire session with a specific client. We can store, retrieve and remove attribute from **HttpSession** object. Any servlet can have access to **HttpSession** object throughout the `getSession()` method of the **HttpServletRequest** object.

How HttpSession works



1. On client's first request, the **Web Container** generates a unique session ID and gives it back to the client with response. This is a temporary session created by web container.
2. The client sends back the session ID with each request. Making it easier for the web container to identify where the request is coming from.

- The **Web Container** uses this ID, finds the matching session with the ID and associates the session with the request.

HttpSession Interface

Creating a new session

```
HttpSession session = request.getSession();
```

getSession() method returns a session.
If the session already exist, it return the existing session else create a new session

```
HttpSession session = request.getSession(true);
```

getSession(true) always return a new session

Getting a pre-existing session

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

return a pre-existing session

Destroying a session

```
session.invalidate();
```

destroy a session

Some Important Methods of HttpSession

Methods	Description
long <code>getCreationTime()</code>	returns the time when the session was created, measured in milliseconds since midnight January 1, 1970 GMT.
String <code>getId()</code>	returns a string containing the unique identifier assigned to the session.
long <code>getLastAccessedTime()</code>	returns the last time the client sent a request associated with the session
int <code>getMaxInactiveInterval()</code>	returns the maximum time interval, in seconds.

void invalidate()	destroy the session
boolean isNew()	returns true if the session is new else false
void setMaxInactiveInterval(int interval)	Specifies the time, in seconds, after servlet container will invalidate the session.

Advantages of HttpSession:

1. There is no restriction on size of HttpSession object. Since, a single object can hold different type of values which are related to client identity.
2. Network traffic flow is very less. Since, only session id is exchanging between client and server.
3. An object of HttpSession neither exists at client side nor exists at server.

Dis-Advantage of HttpSession

1. HttpSession objects allocate memory on the server so this increases burden on the server.
2. If cookies are restricted coming to browser window this technique fails to perform session tracking.

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>
```