

Servlet Programming

- Java Servlets are programs that run on a Web or Application server and act as a middle layer between a requests coming from a Web browser or other HTTP client and databases or applications on the HTTP server.
- Using Servlets, we can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically.

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

HTTP

- HTTP stands for Hyper Text Transfer Protocol.
- Following are some of the important properties of HTTP:
 - It is a stateless protocol, meaning that every HTTP request is independent of each other.
 - It can send data in the request. The server side program reads the data, processes it and sends the response back. This is the most important feature of HTTP, the ability to send data to server side program.
- Based on how the data is sent in the request, HTTP requests are categorized into several types, but the most important and widely used are the **GET requests and POST requests.**

- A typical HTTP request is identified by something called URI (Uniform Resource Identifier) as shown below:

http:// <host name>:<port number>/<request details>

GET Request

- GET request is the simplest of all the requests.
- This type of request is normally used for accessing server-side static resources such as HTML files, images etc.
- This doesn't mean that GET requests cannot be used for retrieving dynamic resources.
- To retrieve dynamic resources, some data must be sent in the request and GET request can send the data by appending it in the URI itself as shown below:

<http://somedomain.com?uid=John&role=admin>

- In the above URI, the HTTP request sends two pieces of data in the form of name value pairs. Each name-value pair must be separated by ‘&’ symbol. The server side program will then read the data, process it, and send a dynamic response.
- Though GET request is simplest of all, it has some limitations:
 1. All the data in the request is appended to the URI itself making the data visible to everyone. When secure information need to be sent, GET request is not the solution.
 2. GET requests can only send text data and not the binary data. Therefore in situations where you

need to upload image files to server, GET request cannot be used.

POST Request

- POST request is normally used for accessing dynamic resources.
- POST requests are used when we need to send large amounts of data to the server.
- Moreover, the data in the POST request is hidden behind the scenes therefore making data transmittal more secure.

- In most of the real world applications, all the HTTP requests are sent as POST requests.

Server Side of the Web Application

- The server- side is the heart of any web application as it comprises of the following:
 - Static resources like HTML files, XML files, Images etc
 - Server programs for processing the HTTP requests
 - A runtime that executes the server side programs
 - A deployment tool for deploying the programs in the server
- In order to meet the above requirements, J2EE offers the following:

- Servlet and JSP technology to build server side programs
- Web Container for hosting the server side programs.
- Deployment descriptor which is an XML file used to configure the web application.

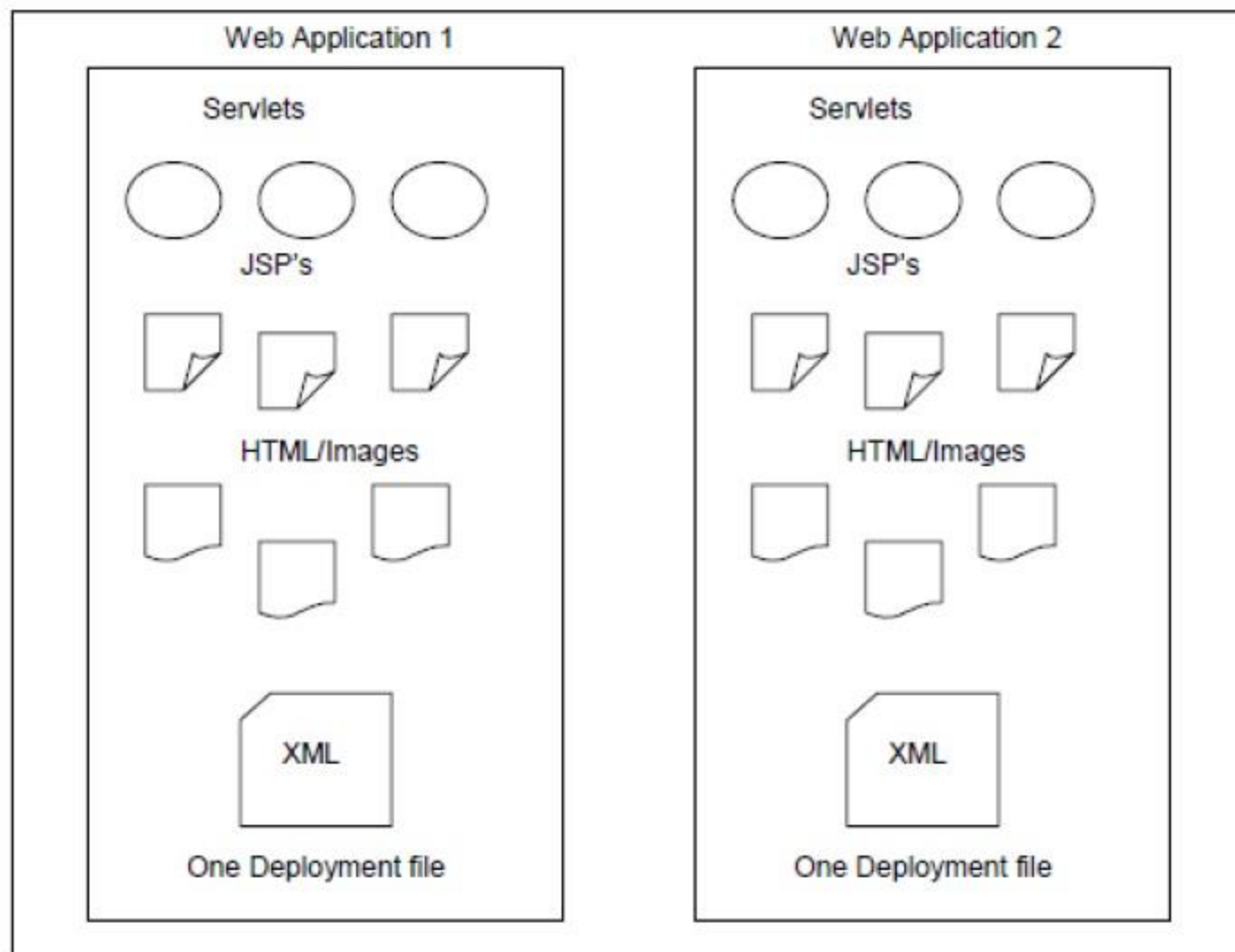
Web Container

- A Web container is the heart of Java based web application which is a sophisticated program that hosts the server side programs like Servlets.
- Once the Servlet programs are deployed in this container, the container is ready to receive HTTP requests and delegate them to the programs

that run inside the container. These programs then process the requests and generate the responses.

- Also a single web container can host several web applications.
- Following figure that shows how a typical web container looks

Web Container



- There are several J2EE Web Containers available in the market.
- One of the most notable one is the Apache's Tomcat container which is available for free of cost.

Servlet Technology

- Servlet technology is a standard J2EE technology used for building dynamic web applications in Java.
- Using Servlet technology is again nothing but using the standard classes and interfaces that it comes with.
- These classes and interfaces form what we call as Servlet API.

Definition of Servlet

- A server side Java program that uses the above API is called as Servlet.
- In simple terms, a Servlet is a server side Java program that processes HTTP requests and generates HTTP response.

Servlet API

- We 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.

- 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.

Package javax.servlet

Some Interfaces	
Interface	Description
RequestDispatcher	This interface implements methods to forward a request or include output from another (active) source such as another servlet.
Servlet	This is the interface for all servlets.
ServletRequest	Whenever the server receives a request it creates a ServletRequest object, puts all the request information in it and passes this along with a ServletResponse object to the appropriate servlet.

Some Classes	
Class	Description
GenericServlet	Abstract base class for all servlets.
HttpConstraintElement	An HttpConstraint annotation value.
HttpMethodConstraintElement	An HttpMethodConstraint annotation value.
MultipartConfigElement	A MultipartConfig annotation value.
ServletContextAttributeEvent	an attribute within the ServletContext changed.
ServletContextEvent	something happened to the specified ServletContext.
ServletInputStream	This class serves as a stream where servlets can read data supplied by the client from.
ServletOutputStream	Used to write output from a Servlet to the client.
ServletRequestAttributeEvent	an attribute within the ServletRequest changed.
ServletRequestEvent	something happened to the specified ServletContext.

ServletRequestWrapper	This class provides an adapter that makes it easy to wrap a request The default behavior of this class is to pass all method calls in the ServletRequest interface through to the underlying request object
ServletResponseWrapper	A convenience class for developers to adapt to a ServletResponse.

Package javax.servlet.http

Some Interfaces	
Interface	Description
HttpServletRequest	Contains all the client's request information.
HttpServletResponse	Object for HttpServlets to return information to the client.
HttpSession	A HttpSession holds session-dependant data on the server side.

Some classes	
Class	Description
Cookie	A cookie is basically a {String,String} name/value pair that the server tells the client to remember and to send back to him attached to every future request. Using cookies a server can maintain a state in between client requests.
HttpServlet	The mother-of-all-HttpServlets.
HttpServletRequestWrapper	Serves a convenience class for developers to adapt to HttpServletRequests by default, passes all method calls which are part of the HttpServletRequest interface through to the wrapped object
HttpServletResponseWrapper	Provided as a convenience to a developer that wishes to write an adapter to an HttpServletResponse Object.
HttpSessionBindingEvent	Send to an Object that implements HttpSessionBindingListener when bound into a session or unbound from a session.
HttpSessionEvent	A session lifecycle event.

- 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.
- 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.

Deployment Descriptor

What is Deployment?

- Copying the .class file of the Servlet from the current directory to the classes folder of Tomcat (or any Web server) is known as deployment.
- When deployed, Tomcat is ready to load and execute the Servlet, at any time, at the client request.

What is Deployment Descriptor?

- As the name indicates, the deployment descriptor describes the deployment information (or Web Information) of a Servlet.
- The deployment descriptor is an XML file known as web.xml. XML is the easiest way to give the information to a server, just writing in between the tags, instead of writing in a text file or RDBMS file.
- The name and tags of web.xml are Servlet API specifications

- A deployment descriptor is a standard XML file named web.xml that is used by the web container to run the web applications. Every web application will have one and only one deployment descriptor (web.xml file).
- This file defines the following important information pertaining to a Servlet:
 1. Servlet name and Servlet class
 2. The URL mapping used to access the Servlet
- Let's assume we wrote a servlet named FormProcesssingServlet in a package named

myservlets. The definition for this servlet in the web.xml will be as shown below:

```
<servlet>
```

```
<servlet-
```

```
    name>FormProcessingServlet</servletname>
```

```
<servletclass>myservlets.FormProcessingServlet</servletclass>
```

```
</servlet>
```

- The servlet name can be any arbitrary name, but it's a good practice to have the class name as the servlet name. However, the servlet class tag must represent the fully qualified name of the servlet which includes the package name as shown above. This completes Step 1.

- The next thing we need to define is the URL mapping which identifies how the servlet is accessed from the browser. For the above servlet, the url mapping will be as shown below:

```
<servlet-mapping>
```

```
<servlet-name>FormProcessingServlet</servlet-name>
```

```
<url-pattern>/FormProcessingServlet</url-pattern> </servlet-mapping>
```

- With the above mapping, the FormProcessingServlet should be accessed with the following URL:

http://localhost:8080/myweb/FormProcessingServlet

- The web container then delegates the request to `myservlets.FormProcessingServlet` class to process the request.

Structure of servlet web app

➤ *Container(tomcat)*

□ *webapps*

firstWebapp

- *WEB-INF*

- ✓ *classes*

- abc.class*

- xyz.class*

- ✓ *lib*

- ✓ *Web.XML*

- *index.html*

Eg:index.html

```
<html>
```

```
<head></head>
```

```
<body>hello</body>
```

```
</html>
```

web.xml

```
<web-app>
```

```
<servlet>
```

```
  <servlet-name>my</servlet-name>
```

```
  <servlet-class>my</servlet-class>
```

```
</servlet>
```



```
<servlet-mapping>
```

```
    <servlet-name>my</servlet-name>
```

```
    <url-pattern>/my</url-pattern>
```

```
</servlet-mapping>
```

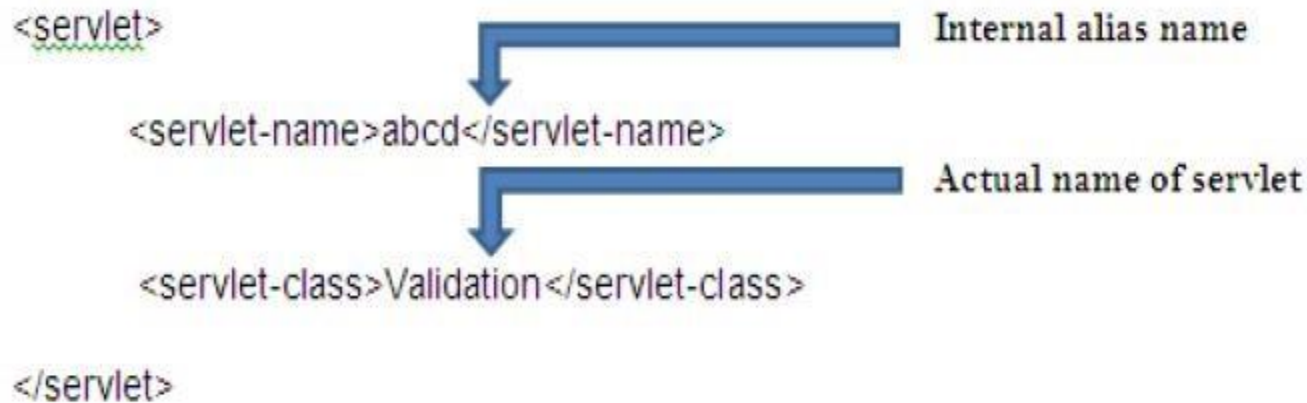
```
</web-app>
```

What information can be stored with deployment descriptor?

The following activities can be done by the programmer in **web.xml** file.

- **1. Mapping alias name with the actual Servlet name**
- First and foremost is the alias name to the Servlet.
- Never a client is given the actual name of the Servlet. Always an alias name is given just for security (avoid

hacking). The alias name is given in the following XML tags.



my.java

```
import jakarta.servlet.*; import  
jakarta.servlet.http.*; import  
java.io.*; public class my extends  
HttpServlet  
{  
    public void service(HttpServletRequest req, HttpServletResponse res)  
        throws ServletException, IOException
```

```
{  
    PrintWriter p=res.getWriter();  
  
    for(int i=1;i<=10;i++)  
    {  
        p.println("tomcat tomcat");  
    }  
}
```

Directory	Description
myweb	This directory represents the entire web application
WEB-INF	This directory contains the configuration file like web.xml
classes	This directory must contain all the Java class files in the web application

- All the static resources namely html files, image files can be stored directly under myweb directory.
- With the above directory structure, all the components within myweb application should be accessed with the URL starting with:

http://localhost:8080/myweb/

- If you want to create another web application like mywebproject, you must again create the same directory structure with mywebproject as the root directory and should access the components using the URL starting with:

http://localhost:8080/mywebproject/

What information can be stored with deployment descriptor?

- The following activities can be done by the programmer in **web.xml** file.
- **1. Mapping alias name with the actual Servlet name**
- First and foremost is the alias name to the Servlet. Never a client is given the actual name of the Servlet.

- Always an alias name is given just for security (avoid hacking). The alias name is given in the following XML tags.
- The Servlet comes with two alias names, internal and external.
- The internal name is used by the Tomcat and the external name is given (to be written in <FORM> tag of HTML file) to the client to invoke the Servlet on the server.
- That is, there exists alias to alias. All this is for security. Observe, the names are given in two different XML tags, in the web.xml file, to make it difficult for hacking (for

more security in EJB(Enterprise Java Beans)), two alias are given in two different XML files).

<servlet>

<servlet-name>abcd</servlet-name>

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

</servlet>

Internal alias name

Actual name of servlet

<servlet-mapping>

<servlet-name>abcd</servlet-name>

<url-pattern>/roses</url-pattern>

</servlet-mapping>

Internal alias name

External alias name

- To invoke the Validation Servlet, the client calls the server with the name roses.
- When roses call reaches the server, the Tomcat server opens the web.xml file to check the deployment particulars.
- Searches such a <servlet-mapping> tag that matches roses. roses is exchanged with abcd.
- Then, searches such a <servlet> tag that matches abcd and exchanges with Validation.
- Now the server, loads Validation Servlet, executes and sends the output of execution as response to client.

2. To write Initialization Parameters

- Initialization parameters are read by the Servlet from web.xml file.
- Programmer can write code to be used for initialization.
- An example code is given below

<init-param>

<param-name>trainer</param-name>

<param-value>S. Nageswara Rao</param-value>

</init-param>

3. To write tag libraries

- **(this is mostly used in frameworks like Struts etc)**
 - **Following are a few important XML elements in web.xml.**
- <context-param>, <filter-mapping>, <taglib> and <mime-mapping> etc.**

Steps for Writing a Servlet

- **Once Tomcat (Web Container) is installed and configured, you can put it to work. Six steps take you**

from writing your servlet to running it. These steps are as follows:

1. Create a directory structure under Tomcat (Web container) for your application.
 2. Write the servlet source code. You need to import the **javax.servlet package and the javax.servlet.http package in your source file.**
 3. Compile your source code.
 4. Create a deployment descriptor.
 5. Start the server and deploy the project
 6. Call your servlet from a web browser.
- A servlet can be created by three ways:
 - By implementing Servlet interface,
 - By inheriting GenericServlet class, (or)

- 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.
- So, writing a servlet is very simple. You just have to follow a standard process as shown below:
 - Create a class that extends **`HttpServlet`**
 - Define methods like **`init()`**, **`doGet()`** ,**`doPost()`** tec.
- These three methods are the standard methods for any servlet.

- These are called as callback methods that the web container invokes automatically when a HTTP request comes to this servlet
- For all the **GET** requests, the web container invokes the **doGet() method**, and for **POST** requests it invokes the **doPost() method**.

Life Cycle of a Servlet

- The life cycle of a servlet represents how the web container uses the servlet to process the requests. Following is what a web container does with a servlet:
 1. Loads the Servlet
 2. Instantiates the Servlet
 3. Initializes the servlet by executing the **init() method**.

4. Invokes **service()** / **doGet()** / **doPost** /... method **repeatedly for each client request**
 5. Destroys the servlet by invoking **destroy()** method
- Three methods are central to the life cycle of a servlet. These are **init()**, **service()**, and **destroy()**. They are implemented by every servlet and are invoked at specific times by the server.

Let us consider a typical user scenario to understand when these methods are called.

- First, when a user enters a Uniform Resource Locator (URL) to a Web browser. The browser then generates an HTTP request for this URL. This request is then sent to the appropriate server.
- Second, this HTTP request is received by the Web server. The server maps this request to a particular servlet. The

servlet is dynamically retrieved and loaded into the address space of the server.

- Third, the server invokes the **init() method of the servlet**. This method is invoked only when the servlet is first loaded into memory. **It is possible to pass initialization parameters to the servlet so it may configure itself.**
- Fourth, the server invokes the **service() method** of the servlet. This method is called to process the HTTP request. It is possible for the servlet to read data that has been provided in the HTTP request. It may also formulate an HTTP response for the client. The servlet remains in the server's address space and is available to process any other HTTP requests received from clients. The service() method is called for each HTTP request.

- Finally, the server may decide to unload the servlet from its memory. The server calls the **destroy()** method to relinquish any resources such as file handles that are allocated for the servlet. Important data may be saved to a persistent store. The memory allocated for the servlet and its objects can then be garbage collected.
- 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()

- **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:

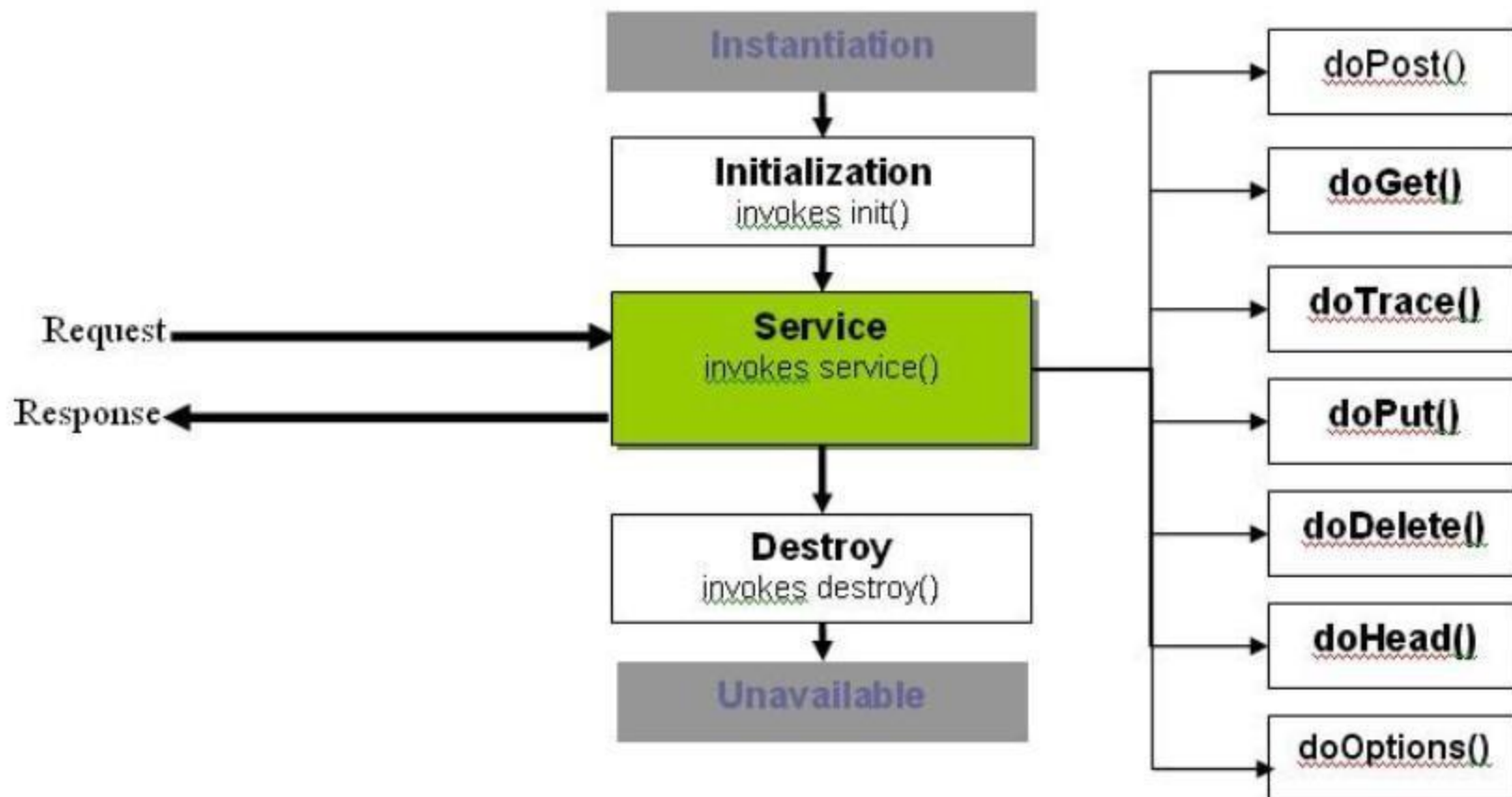
```
<servlet>  
  <servlet-name>MyServlet</servlet-name>  
  <servlet-class>com.beginnersbook.MyServletDemo</servlet-class>  
  <load-on-startup>1</load-on-startup>  
</servlet>
```
- Now the **`init()`** method for corresponding servlet class **`com.beginnersbook.MyServletDemo`** would be invoked during 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.*

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.



Servlet Initialization :Example

- Initializing a servlet is one of the most common practices. Servlet initialization is done in the **init() method of the servlet.**
- Defining initialization parameters for servlets in web.xml is a good practice as it eliminates hard coding in the servlet.
- The web container while loading the servlet, invokes the **init() method of the servlet as part of the life cycle and passes the parameters defined in the web.xml to it.** The other advantage with this is, in the future if the initialization values need to be changed, you only have to change the web.xml without having to recompile the entire web application.

- Initialization parameters to a servlet are defined in the **web.xml** using the **<init-param>** element as shown below:

```
<init-param>
```

```
<param-name>driver</param-name>
```

```
</param-value>com.mysql.jdbc.Driver</param-value>
```

```
</init-param>
```

```
<init-param>
```

```
<param-name>URL</param-name>
```

```
<param-
```

```
value>jdbc:mysql://localhost:3306/MyDB</paramvalue>
```

```
</init-param>
```

With the above definition, a servlet can access the parameter in the `init()` method as shown below: **public void init(ServletConfig config) {**

```

    String drivename = config.getInitParameter("driver");
}
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.*;
public class initParamDemo extends HttpServlet
{
    String drivename;
    String databaseURL;
    @Override
    public void init(ServletConfig config) throws ServletException {
        drivename = config.getInitParameter("driver");
        databaseURL = config.getInitParameter("URL");

    }
    @Override public void service(HttpServletRequest req, HttpServletResponse res)throws
    ServletException,
    IOException {
        res.setContentType("text/html");
        PrintWriter pw = res.getWriter();
        // Send the response
        pw.println("<h1>Driver Name is : </h1><br/>" + drivename);
        pw.println("<h1>DB URL is : </h1>" + databaseURL);
    }
}

```

Reading HTML Form Data

- HTML form processing is one of the most common things that any web application does.
- You might have noticed on various websites where you fill in html form with all the information and submit it for processing.
- The application then responds with a confirmation message something like “information is successfully processed”.
Registration pages, Email composing etc are some of the examples.

Following are the steps for processing a HTML form using a Servlet

- **User fills the data in the html page and submits it.**
- **The form data will then be sent to Servlet**
- **Servlet reads the form data, processes it and send a confirmation.**

- The form data will be sent in the HTTP request object as name value pairs as shown below:

**http://localhost:8080/myweb/SomeServlet?name=John
&age=20**

- The servlet then reads the above request data using the **HttpServletRequest object as shown below:**

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

```
String age = request.getParameter("age");
```

- **fn will now have John and age will have 20. The servlet can do whatever it want with the data, and send a confirmation message back.**

Reading Form Data using Servlet

- Servlets handles form data parsing automatically using the following methods depending on the situation –

- **getParameter()** – You call `request.getParameter()` method to get the value of a form parameter.
- **getParameterValues()** – Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
- **getParameterNames()** – Call this method if you want a complete list of all parameters in the current request.

Example: Web application to compute simple interest

//File : uiforsimpleinterest.html

```
<html>
  <head>
    <title>Simple Interest</title>
  </head>
  <body>
    <h1> Enter Following Values and Click on Compute Button</h1>
    <form action="findsi" method="POST">
  <table>
  <tr>
    <td>Principle</td>
    <td><input type="text" name="principle"/>
  </tr>
  <tr>
    <td>Time</td>
    <td><input type="text" name="time"/>
```

```
</tr>
<tr>
    <td>Rate</td>
    <td><input type="text" name="rate" />
</tr>
<tr>
    <td></td>
    <td><input type="submit" value="Compute" />
</tr>
</table>
</form>
</body>
</html>
```

//File:

SimpleInterestFindingServlet.java

```
import javax.servlet.*; import
javax.servlet.http.*; import java.io.*;
public class SimpleInterestFindingServlet extends HttpServlet {
@Override protected void doPost(HttpServletRequest request, HttpServletResponse
    response)
throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
//reading the values
    String principle =request.getParameter("principle");
    String time=request.getParameter("time");
    String rate=request.getParameter("rate");
//parsing the string values to double double t =
    Double.parseDouble(time); double p =
    Double.parseDouble(principle); double r =
    Double.parseDouble(rate);
```



```
// declaring variable to hold the result double  
    si;
```

```
//computing simple interest
```

```
    si = (p*t*r)/100;
```

```
//to send response
```

```
    out.println("<h3>Simple Interest =
```

```
    "+si+
```

```
    "</h3>");
```

```
}
```

```
}
```

```
// Inside web.xml
```

...

<servlet>

 <servlet-name>SimpleInterestFindingServlet</servlet-name>

<servlet-class>someServlets.SimpleInterestFindingServlet</servlet-class>

</servlet>

<servlet-mapping>

 <servlet-name>SimpleInterestFindingServlet</servlet-name>

 <url-pattern>/findsi</url-pattern>

</servlet-mapping>

...

Simple Interest

localhost:8084/OurW

Enter Following Values and Click on Compute Button

Principle

Time

Rate

After Clicking "Compute"

localhost:8084/OurWebApp/fin

localhost:8084/OurW

Simple Interest = 366.89799999999997

Example: Reading the values from html form and storing in a table of a database

//File: simpleform.html

```
<html>
<head> <title>Simple Form</title> </head>
<body>
<h3>Please fill in the following details and submit it</h3>
<form action="formprocessing" method="POST"> <table>
<tr>
    <td>Name</td>
    <td><input type="text" name="name"/>
</tr>
<tr>
    <td>Age</td>
    <td><input type="text" name="age"/>
</tr>
<tr>
    <td>Address </td>
    <td><input type="text" name="address"/>
```

```
</tr>
```

```
<tr>
```

```
    <td> <br/><br/></td>
```

```
    <td><br/><br/><input type="submit" value="Submit"/>
```

```
</tr>
```

```
</table>
```

```
</form>
```

```
</body>
```

```
</html>
```

//File: FormProcessingServlet.java public class

FormProcessingServlet extends HttpServlet {

String driverName; String

dbURL;

final String DB_USERNAME="root"; final

String DB_PASSWORD="";

@Override

public void init(ServletConfig config) throws

ServletException {

// This method is called before the following methods are called

//and gets called only ONCE. This is like a

// constructor. We do all the initialization here.

driverName = config.getInitParameter("driver");

dbURL = config.getInitParameter("URL"); }

@Override

public void doPost(HttpServletRequest request, HttpServletResponse
response)

throws ServletException, IOException {

response.setContentType("text/html")

; PrintWriter out =

response.getWriter();

String responseMessage;

// Reading the form data

```
String name = request.getParameter("name");  
String a = request.getParameter("age");  
String addr = request.getParameter("address");
```

```
try{  
    //converting age value into int  
    int age = Integer.parseInt(a);  
    //calling method which connects to db and inserts values  
    int status=connectAndInsert(name,age,addr); if (status!=  
    1)  
    responseMessage = "Congratulations! the form is successfully  
    submitted"; else responseMessage = "Sorry! something went wrong.  
    Please try again";  
}
```

```
catch(Exception e)  
{  
    responseMessage= "Error !!! " + e;  
}
```

```
// Sending the response out.println("<h1>" +  
    responseMessage + "</h1>");  
}
```

//method to handle db related works...

```
private int connectAndInsert(String name, int age, String  
    addr)throws Exception{  
    Class.forName(driverName);  
    Connection con; con =  
    DriverManager.getConnection(dbURL, DB_USERNAME,  
    DB_PASSWORD);  
    String queryToFire = "INSERT INTO personalrecord  
    VALUES(\"'+name+'\",'+age+',    \"'+addr+'\"");  
    Statement st;  
    st = con.createStatement();  
    int n;  
    n=st.executeUpdate(queryToFire)  
    ; st.close(); con.close();
```



```
return n;  
}
```

```
}
```

in web.xml

.....

.....

```
<init-param>
```

```
  <param-name>driver</param-name>
```

```
  <param-value>com.mysql.jdbc.Driver</param-value>
```

```
</init-param>
```

```
<init-param>
```

```
  <param-name>URL</param-name>
```

```
  <paramvalue>jdbc:mysql://localhost:3306/java2lab</p
```

```
aramvalue> </init-param>
```

.....

.....

Please fill in the following details and submit it

Name

Mahima

Age

19

Address

Nepal

Submit

**Congratulations! the form is
successfully submitted**

Session Management

- The examples we saw until now just deal with one servlet.
- However, a typical web application comprises of several servlets that require collaborating with each other to give a complete response.
- For instance, if you go online to purchase a book, you need to go through multiple pages like search page, shopping page, billing page etc before you can complete the transaction.

- In situations like this, it is important that one servlet shares information or data with other servlet.

- In web application terminology we call the shared data or information as state.
- Having state is just not sufficient.
- Someone must be able to pass this state from one servlet to other servlet so that they can share the data.
- So, who does this state propagation?
- Can HTTP do this? No, because HTTP is a stateless protocol which means it cannot propagate the state.

- So, is there anyone to help us out here to make the state available to all the servlets?
- Yes, there is one guy who is always there for our rescue and it's none other than web container (Tomcat).
- *HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the Web server and the server automatically does not keep any record of previous client request.*
- A web container provides a common storage area called as session store the state and provides access to this session to all the servlets within the web application.

- For instance, servlet A can create some state (information) and store it in the session. Servlet B can then get hold of the session and read the state.
- 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
- There are following ways to maintain session between web client and web server (i.e. State can be maintained indirectly) using –
 1. Cookies
 2. Hidden Form Fields
 3. URL Rewriting
 4. HttpSession Object
- But here we are just focusing on **HttpSession Object**

1. 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 time browser does not support a cookie.

2. 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.

3. URL Rewriting

- You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.

- 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.
- 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.

4. The HttpSession Object

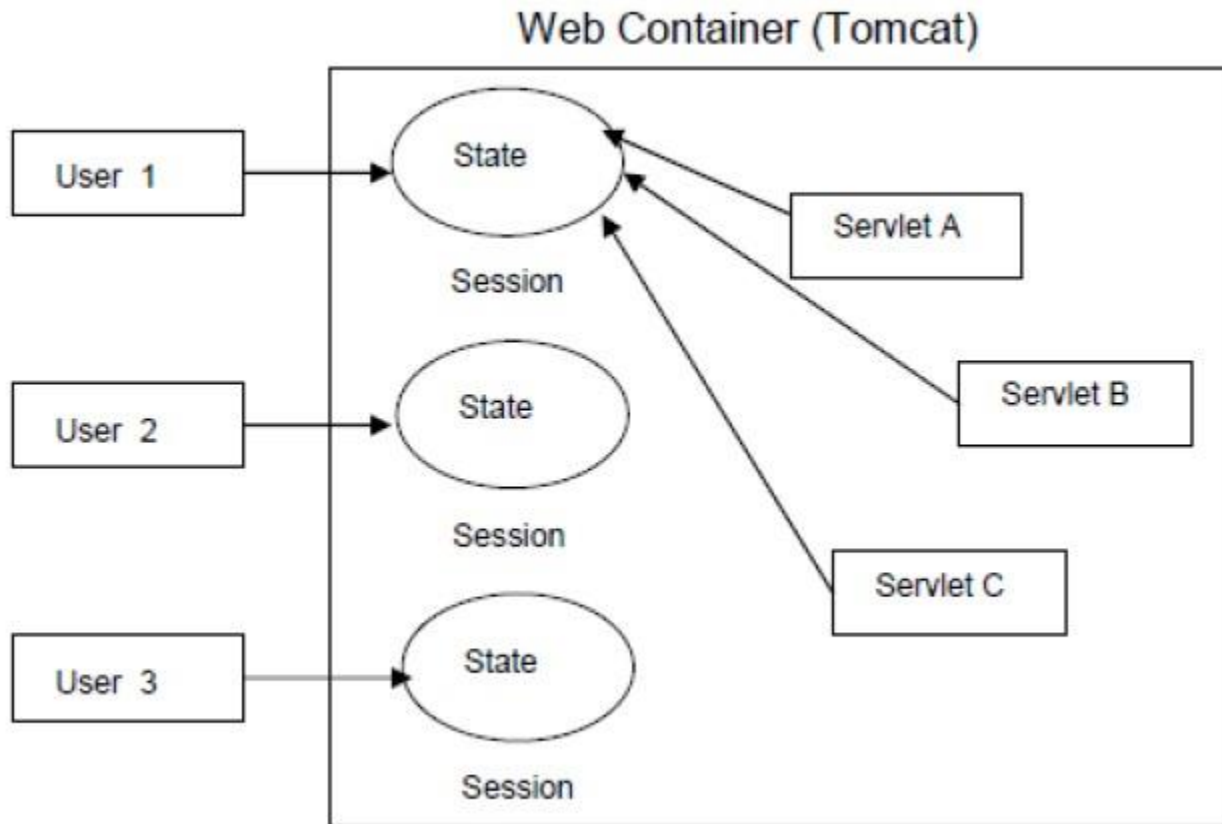
- Apart from the above mentioned three ways, servlet provides **HttpSession Interface** which provides a way to identify a user across more than one page request

or visit to a Web site and to store information about that user.

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

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

- Since the state (data or information) in the session is user specific, the web container maintains a separate session for each and every user as shown in the following diagram.



(These four operations are also called as Session Management operations)

1. Create the session

2. Store the data in the session
3. Read the data from the session
4. Destroy the session or invalidate the session.

Creating a Session

- The servlet API provides us with a class called **HttpSession** to work with sessions.

- To create a session, we do the following:

HttpSession session = request.getSession(true);

- The above method returns a new session object if one is not present, otherwise it

returns the old session object that is already created before.

Storing the data in Session

- Data in session is stored as key-value pair just like in HashMap or Hashtable.
- The value can be any Java object and the key is usually a String.
- To store the data we use the **setAttribute()** method as shown below:

```
session.setAttribute("price",new Double("12.45"));
```


Reading the data from the Session

- To read the data, we need to use the **getAttribute()** method by passing in the key as shown below which then returns the value object:
- ***Double d = (Double)session.getAttribute("price");***
- Notice that we need to do proper casting here.
- Since we stored Double object, we need to cast it again as Double while reading it.

Destroying the Session

- A session is usually destroyed by the last page or servlet in the web application. A session is destroyed by invoking the **invalidate() method** as shown below: *session.invalidate()*

Creating a new session

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

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

```
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

Note: Some useful methods are listed below

1	public Object getAttribute(String name) This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	public Enumeration getAttributeNames() This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	public long getCreationTime() This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	public String getId() This method returns a string containing the unique identifier assigned to this session.

5	public long getLastAccessedTime() This method returns the last accessed time of the session, in the format of milliseconds since midnight January 1, 1970 GMT
6	public int getMaxInactiveInterval() This method returns the maximum time interval (seconds), that the servlet container will keep the session open between client accesses.
7	public void invalidate() This method invalidates this session and unbinds any objects bound to it.
8	public boolean isNew() This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	public void removeAttribute(String name) This method removes the object bound with the specified name from this session.
10	public void setAttribute(String name, Object value) This method binds an object to this session, using the name specified.
11	public void setMaxInactiveInterval(int interval) This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Request Dispatching

- Request dispatching is the ability of one servlet to dispatch or delegate the request to another servlet for processing.
- In simple words, let's say we have a servlet A which doesn't know how to completely process the request.
- Therefore, after partially processing the request, it should forward the request to another servlet B.
- This servlet then does the rest of the processing and sends the final response back to the browser.
- The class used for dispatching requests is the **RequestDispatcher** interface in **Servlet API**. This interface has two methods namely **forward()** and **include()**.
- The **getRequestDispatcher()** method of **ServletRequest** interface returns the object of **RequestDispatcher**.

Syntax:

- `public RequestDispatcher getRequestDispatcher(String resource);` Example:
- *`RequestDispatcher rd=request.getRequestDispatcher("servlet2");`*
`//servlet2` is the url-pattern of a servlet
`rd.forward(request, response);`//method may be include or forward

The forward() method

- This method is used for forwarding request from one servlet to another.
- Consider two servlets A and B. Using this method, A gets the request, which then forwards the request to B, B processes the request and sends the response to the browser.

- This method takes **HttpServletRequest** and **HttpServletResponse** as parameters.

The **include()** method

- With this method, one servlet can include the response of other servlet.
- The first servlet will then send the combined response back to the browser.
- This method also takes **HttpServletRequest** and **HttpServletResponse** as parameters.

Following two figures demonstrate dispatching using **forward()** and **include()** methods

Forward Dispatching



Include Dispatching

forward() vs include() method

To understand the difference between these two methods, let's take an example:

1.

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.

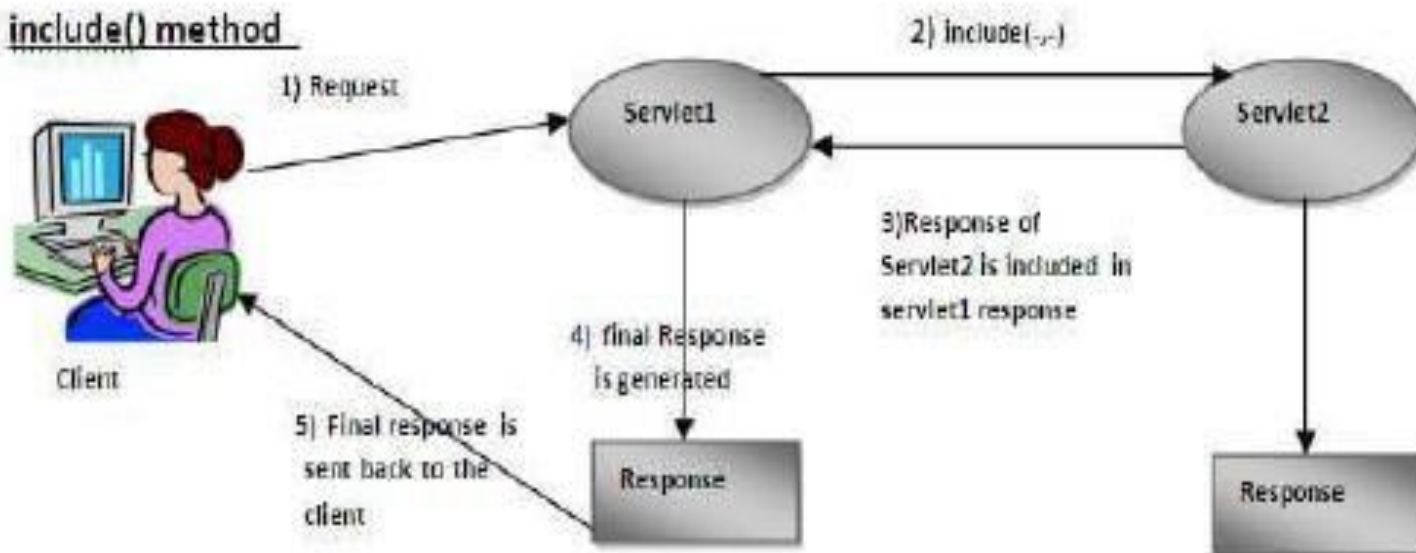
- At the end of the processing of page Y, the control will return back to the page X starting just after the include tag and remain in X till the end.
- *In this case the final response to the client will be send by page X.*

2.

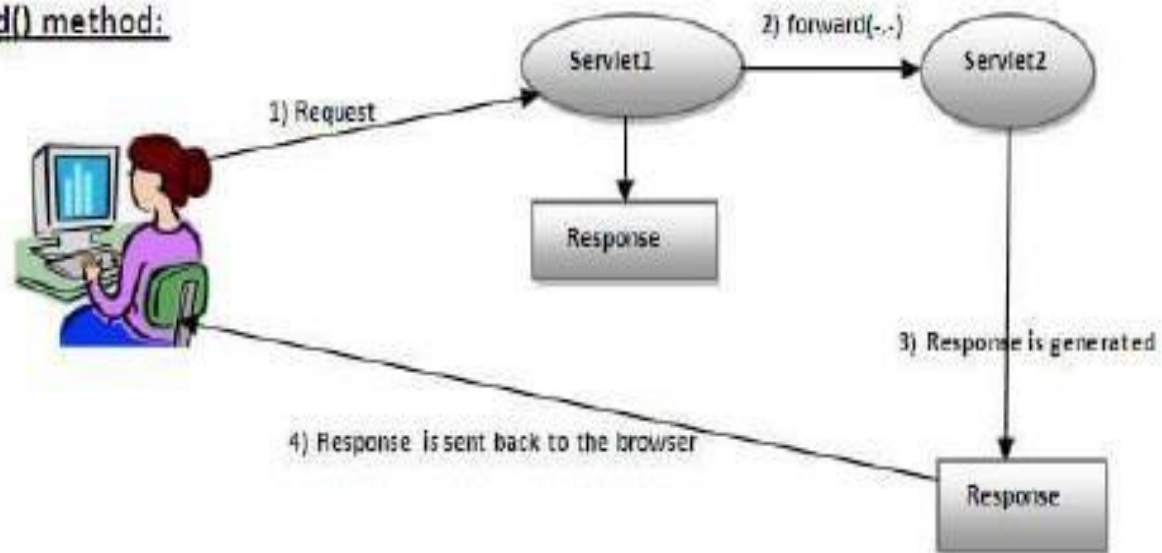
Now, we are taking the same example with forward.

- 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.*

include() method



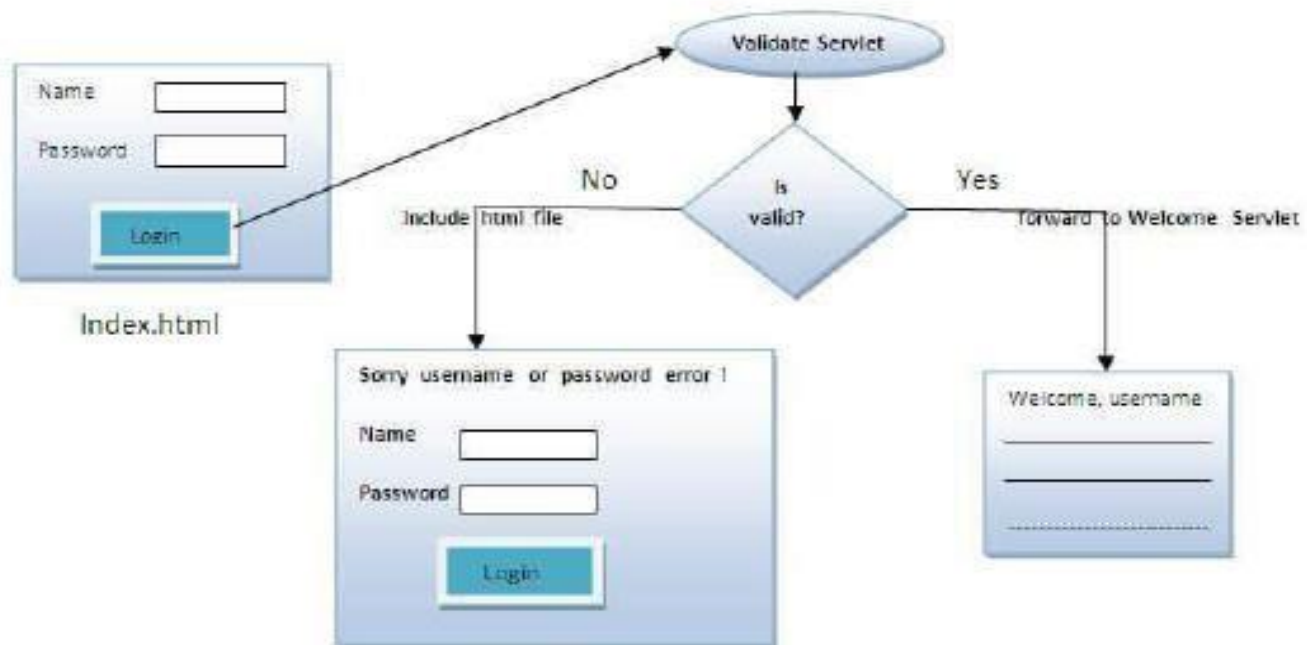
forward() method:



Example : Request Dispatching (RequestDispatcher interface, forward() and include() methods)

- In this example, we are validating the username and password entered by the user.
- If username is Learner and password is servlet123, 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. 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 username is Learner and password is servlet123, 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

```
<html>
```

```
<head>
```

```
<title>RequestDispatching</title>
```

```
<body>
```

```
    <h4> Login </h4>
```

```
    <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>
```

```
</body>
```

```
</html>
```

- **//Login.java** public class Login
extends HttpServlet {
@Override
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String name=request.getParameter("userName");
String pass=request.getParameter("userPass");
if(name.equals("Learner")&& pass.equals("servlet123")){
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 public class  
WelcomeServlet extends HttpServlet {
```

@Override

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");    PrintWriter    out    =
    response.getWriter();

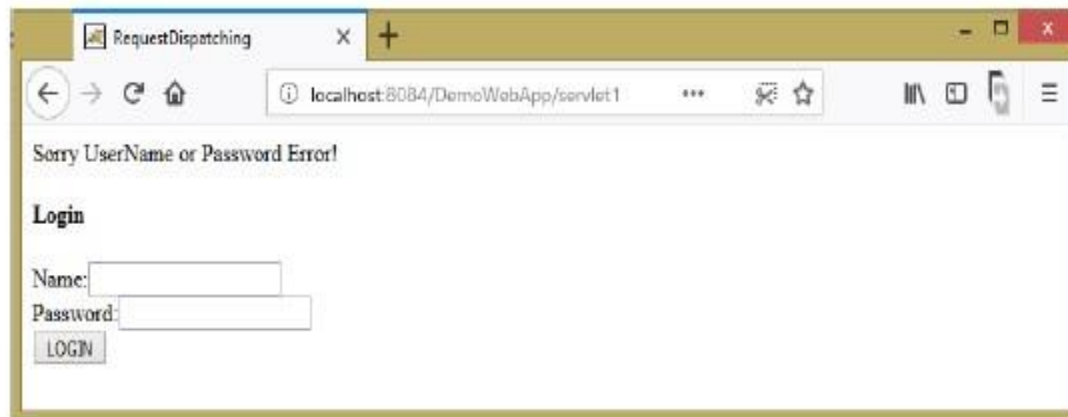
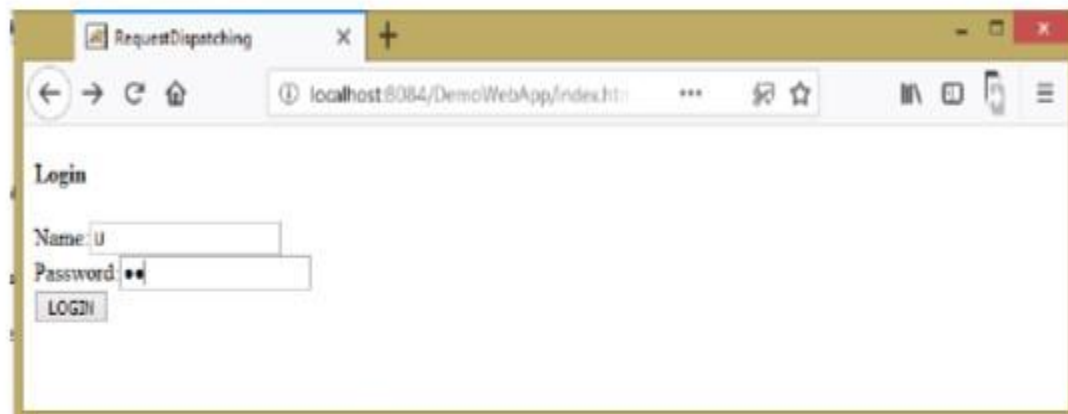
        String name=request.getParameter("userName");
        out.print("Welcome "+name);

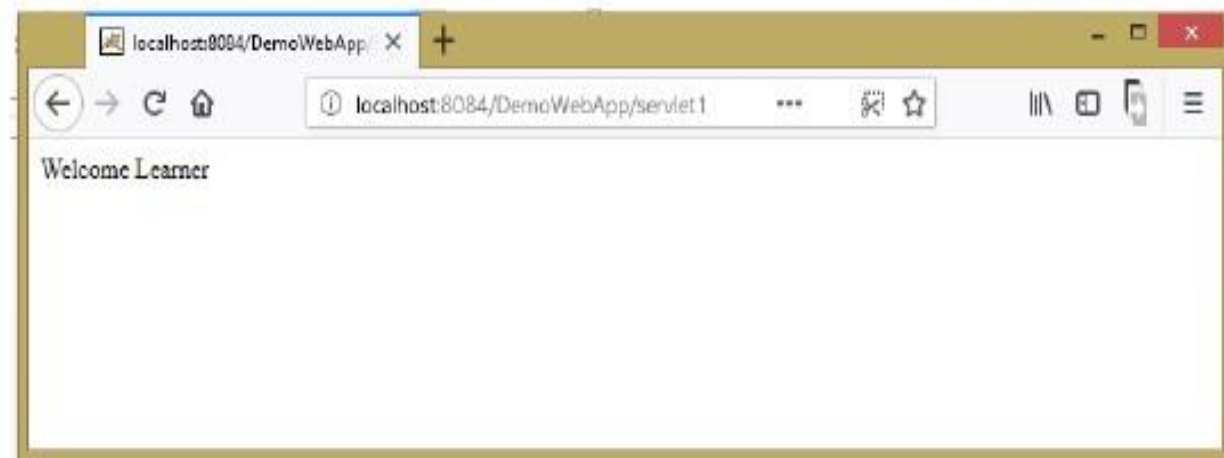
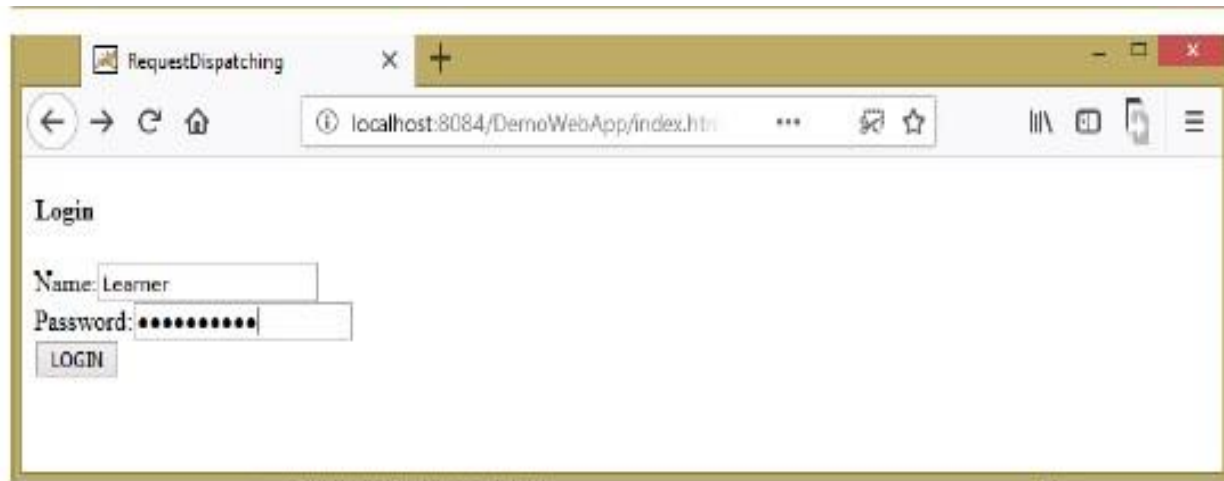
} }
```

//web.xml

```
<web-app>
    <servlet>
        <servlet-name>Login</servlet-name>
        <servlet-class>someServlets.Login</servlet-class>
    </servlet>
```

```
<servlet>
  <servlet-name>WelcomeServlet</servlet-name>
  <servlet-class>someServlets.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>
</web-app>
```





Example : Servlet HttpSession Login and Logout

- We can bind the objects on **HttpSession** instance and get the objects by using **setAttribute** and **getAttribute** methods.
- In this example, we are creating 3 links: login, logout and profile. User can't go to profile page until s/he is logged in. If user is logged out, s/he need to login again to visit profile.
- In this application, we have created following files.
 1. main.html
 2. link.html
 3. login.html
 4. LoginServlet.java
 5. LogoutServlet.java
 6. ProfileServlet.java
 7. web.xml

//main.html


```
<!DOCTYPE html>
<html>
<head><title>MainPage</title></head>
  <body>
    <div align = "center">
      <h1>Login App using HttpSession</h1>
      <hr/>
      <h2>
        <a href="login.html">Login</a>|
        <a href="LogoutServlet">Logout</a>|
        <a href="ProfileServlet">Profile</a>
      </h2>
      <br/><hr/>
    </div>
  </body>
</html>

//login.html
<!DOCTYPE html>
```

```
<html><head><title>LoginPage</title></head>
<body>
<div align ="center"><h2> Login </h2>
  <form action="LoginServlet" method="post">
    User Name:<input type="text" name="name">
    <br/><br/>
    Password:<input type="password" name="password">
    <br/><br/>
    <input type="submit" value="login">
  </form>
</div>
</body>
</html>
```

//link.html

```
<!DOCTYPE html>
```

```
<html>
<head><title>Links</title> </head>
<body>
    <div align ="center">
        <h2> <a href="login.html">Login</a> |
        <a href="LogoutServlet">Logout</a> |
        <a href="ProfileServlet">Profile</a>
        </h2><hr/> <hr/>
    </div>
</body>
</html>
```

//LoginServlet.java

```
public class LoginServlet extends HttpServlet {
    @Override
```

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException { response.setContentType("text/html");
PrintWriter out=response.getWriter();
    request.getRequestDispatcher("link.html").include(request, response);
    String name=request.getParameter("name");
    String
    password=request.getParameter("password");
    if(password.equals("admin123")){
        out.print("Welcome, "+name);
            HttpSession session=request.getSession();
            session.setAttribute("name",name);
    }
    else{ out.print("<h3><font color ='red'>Sorry, username or
        password
        error!</font><h3>");
        request.getRequestDispatcher("login.html").include(request, response);
    }
    out.close();
}
}
```

//LogoutServlet.java

```
public class LogoutServlet extends HttpServlet {  
    @Override  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException { response.setContentType("text/html");  
        PrintWriter out=response.getWriter();  
        request.getRequestDispatcher("link.html").include(request, response);  
        HttpSession session=request.getSession(false);  
        if(session==null){ out.print("You are not logged  
        in yet!");  
        }  
        else  
        {  
            session.invalidate();  
            out.print("You are successfully logged out!");  
        }  
        out.close();  
    }  
}
```

//ProfileServlet.java

```
public class ProfileServlet extends HttpServlet {
```

@Override

```
public void doGet(HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out=response.getWriter();
    request.getRequestDispatcher("link.html").include(request, response);
    HttpSession session=request.getSession(false); if(session!=null){
        String name=(String)session.getAttribute("name"); out.print("Hello,
        "+name+" Welcome to Profile <br/> You are awesome :)");
    }
    else{ out.print("<h3><font color='red'> Please login first</font></h3>");
        request.getRequestDispatcher("login.html").include(request, response);
    }
    out.close();
}
}
```

//web.xml

<web-app>

<servlet>

 <servlet-name>LoginServlet</servlet-name>

 <servlet-class>someervlets.LoginServlet</servlet-class>

</servlet>

<servlet>

 <servlet-name>LogoutServlet</servlet-name>

 <servlet-class>someervlets.LogoutServlet</servlet-class>

</servlet>

<servlet>

 <servlet-name>ProfileServlet</servlet-name>

 <servlet-class>someervlets.ProfileServlet</servlet-class>

</servlet>

<servlet-mapping>

 <servlet-name>LoginServlet</servlet-name>

```
<url-pattern>/LoginServlet</url-pattern>
```

```
</servlet-mapping>
```

```
<servlet-mapping>
```

```
<servlet-name>LogoutServlet</servlet-name>
```

```
<url-pattern>/LogoutServlet</url-pattern>
```

```
</servlet-mapping>
```

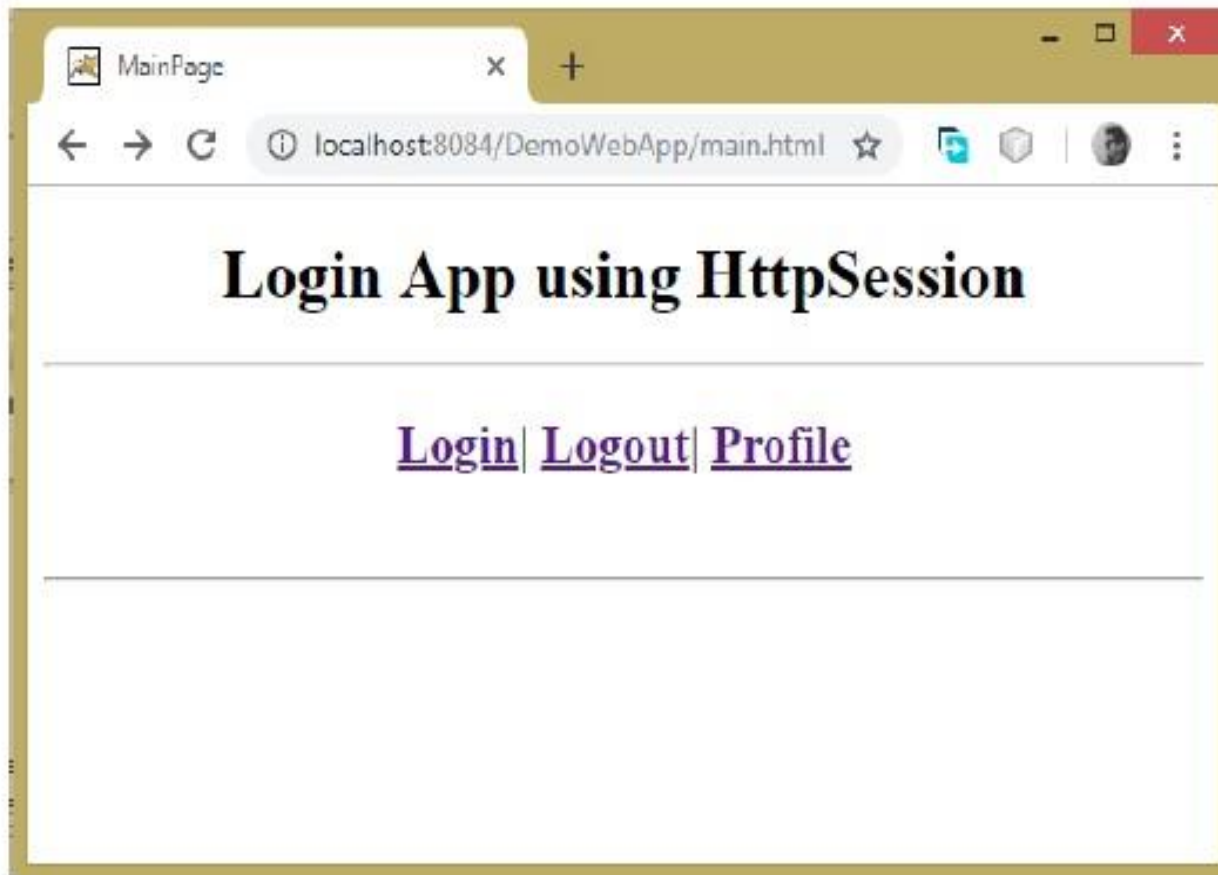
```
<servlet-mapping>
```

```
<servlet-name>ProfileServlet</servlet-name>
```

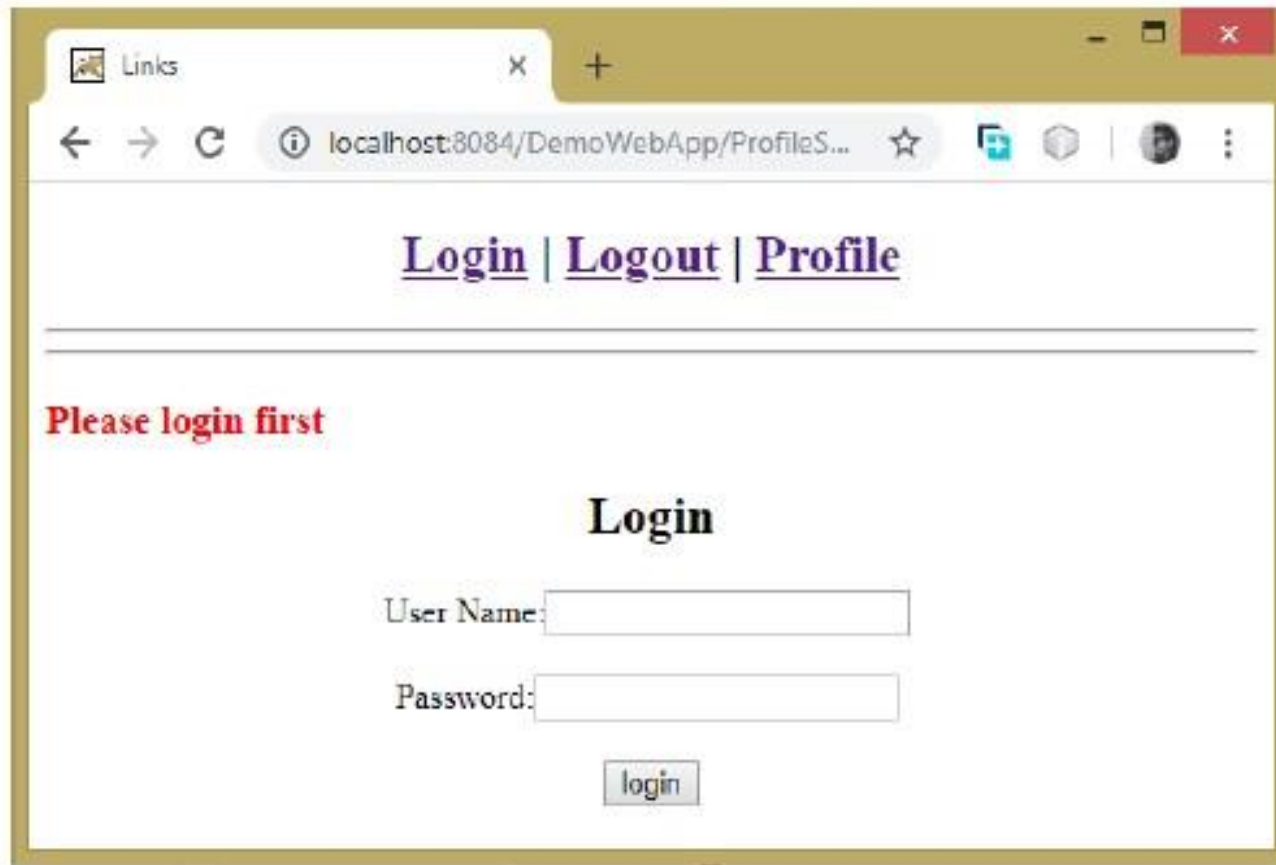
```
<url-pattern>/ProfileServlet</url-pattern>
```

```
</servlet-mapping>
```

```
</web-app>
```

When Profile is clicked without logging in



Links

localhost:8084/DemoWebApp/ProfileS...

[Login](#) | [Logout](#) | [Profile](#)

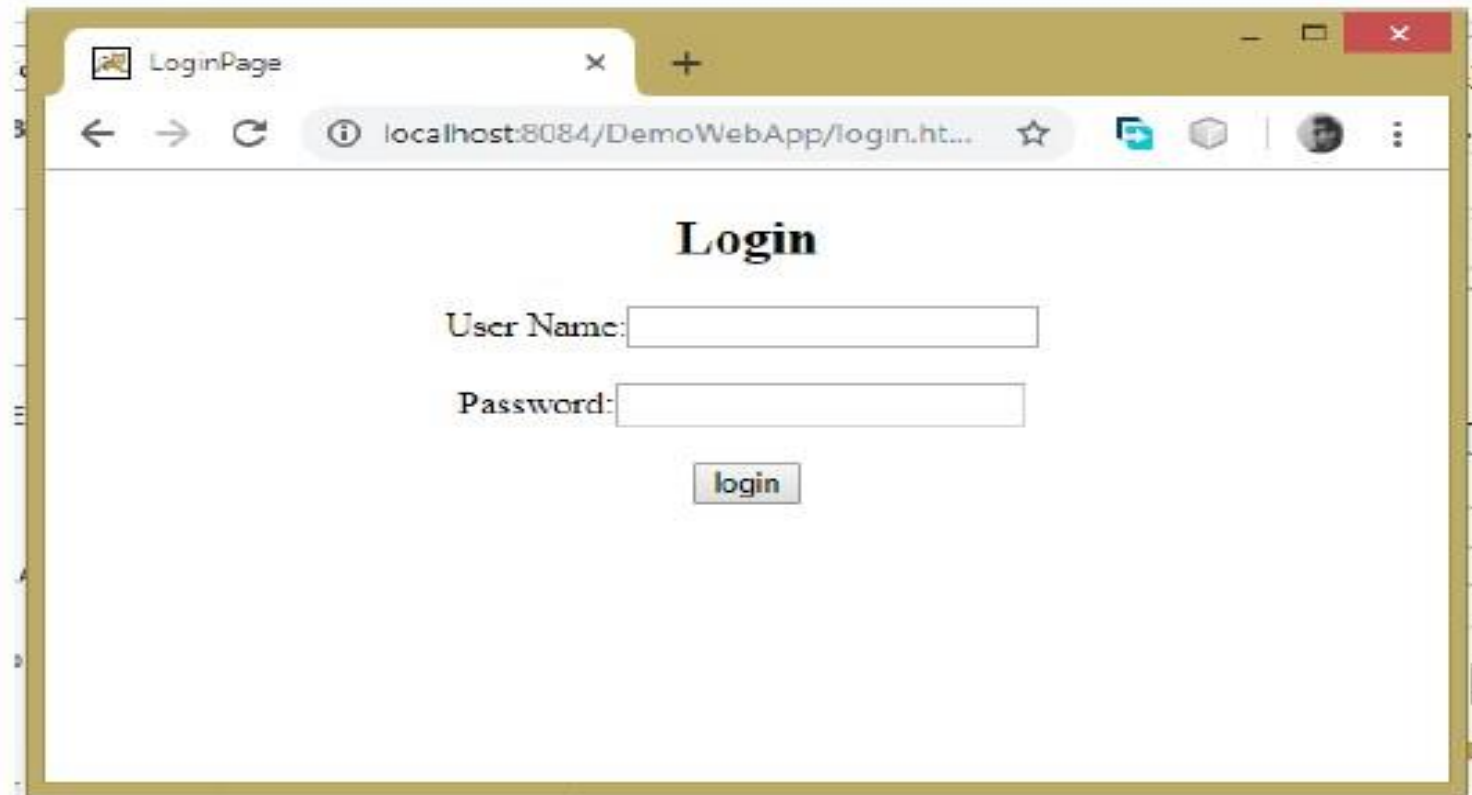
Please login first

Login

User Name:

Password:

When login is clicked



The screenshot shows a web browser window with a single tab titled "LoginPage". The address bar displays "localhost:8084/DemoWebApp/login.ht...". The page content features a centered heading "Login" in a bold serif font. Below the heading are two input fields: "User Name:" followed by a text box, and "Password:" followed by a text box. A "login" button is positioned centrally below the password field. The browser's interface includes standard navigation buttons (back, forward, refresh), a star icon for bookmarks, and a profile icon in the top right corner.

LoginPage

localhost:8084/DemoWebApp/login.ht...

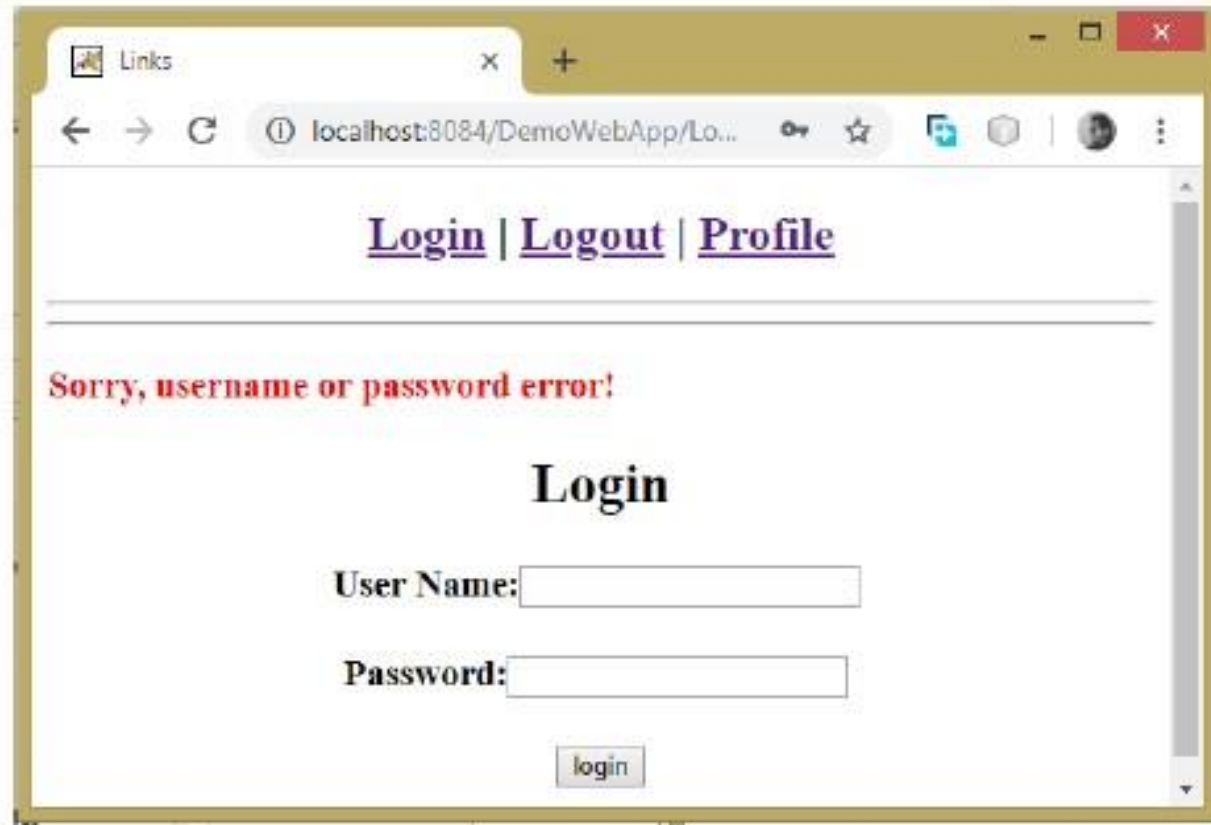
Login

User Name:

Password:

login

When incorrect password is entered



The screenshot shows a web browser window with a single tab titled "Links". The address bar displays "localhost:8084/DemoWebApp/Lo...". The page content includes a navigation bar with links [Login](#), [Logout](#), and [Profile](#). Below the navigation bar, a red error message states "Sorry, username or password error!". Underneath the error message is a large heading "Login". Below the heading are two input fields: "User Name:" and "Password:". At the bottom of the form is a "login" button.

[Login](#) | [Logout](#) | [Profile](#)

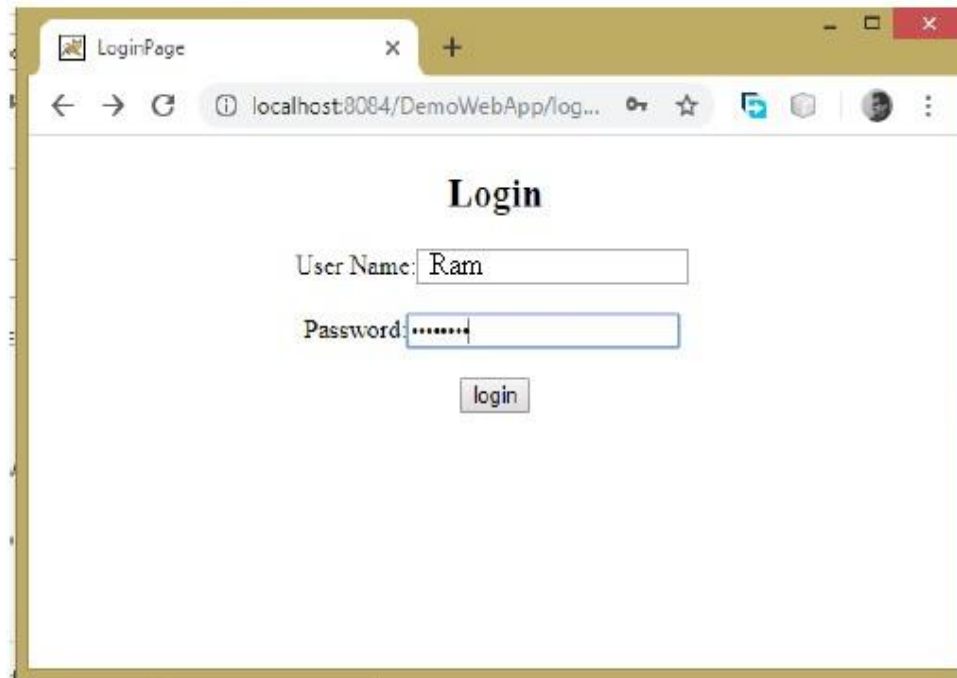
Sorry, username or password error!

Login

User Name:

Password:

When correct password is entered



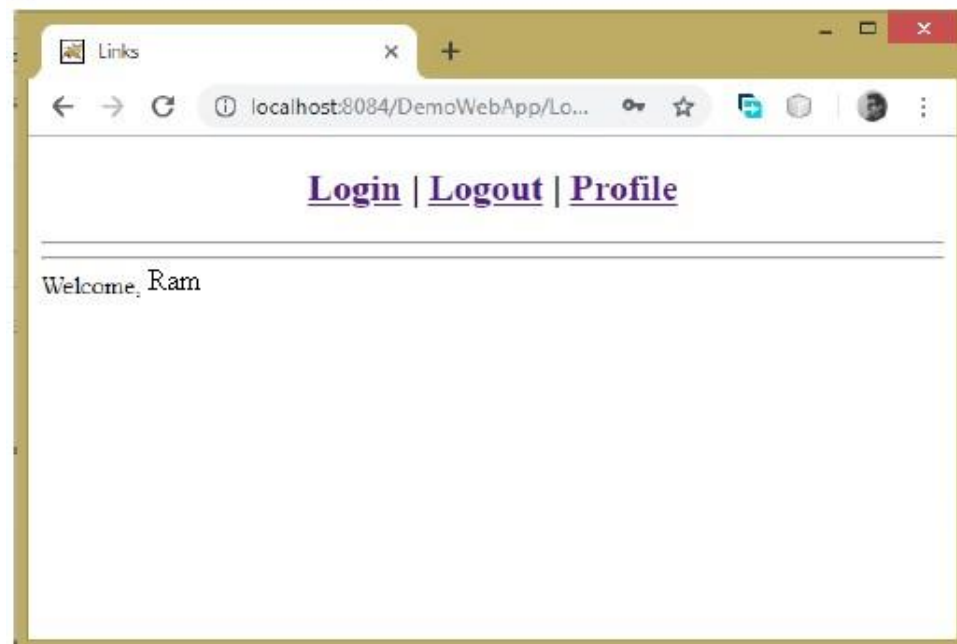
LoginPage

localhost:8084/DemoWebApp/log...

Login

User Name:

Password:



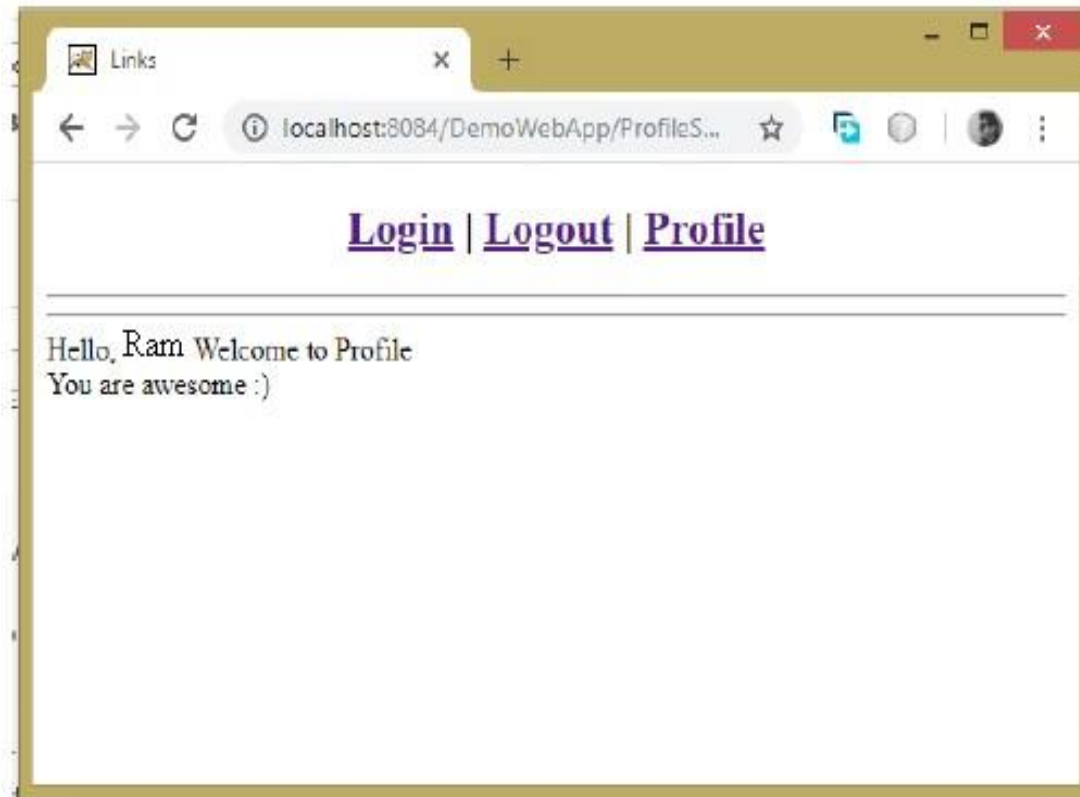
Links

localhost:8084/DemoWebApp/Lo...

[Login](#) | [Logout](#) | [Profile](#)

Welcome, Ram

Clicking on profile after logging in



After clicking on logout

