SERVLET JSP

DAY 10:

Up until now, we've been working with console-based Java applications. All of our inputs and outputs were handled through the console, and there was no user interface (UI). But in real-world applications, we rarely interact with users via the console. Instead, we create web applications where users interact through a browser-based UI.

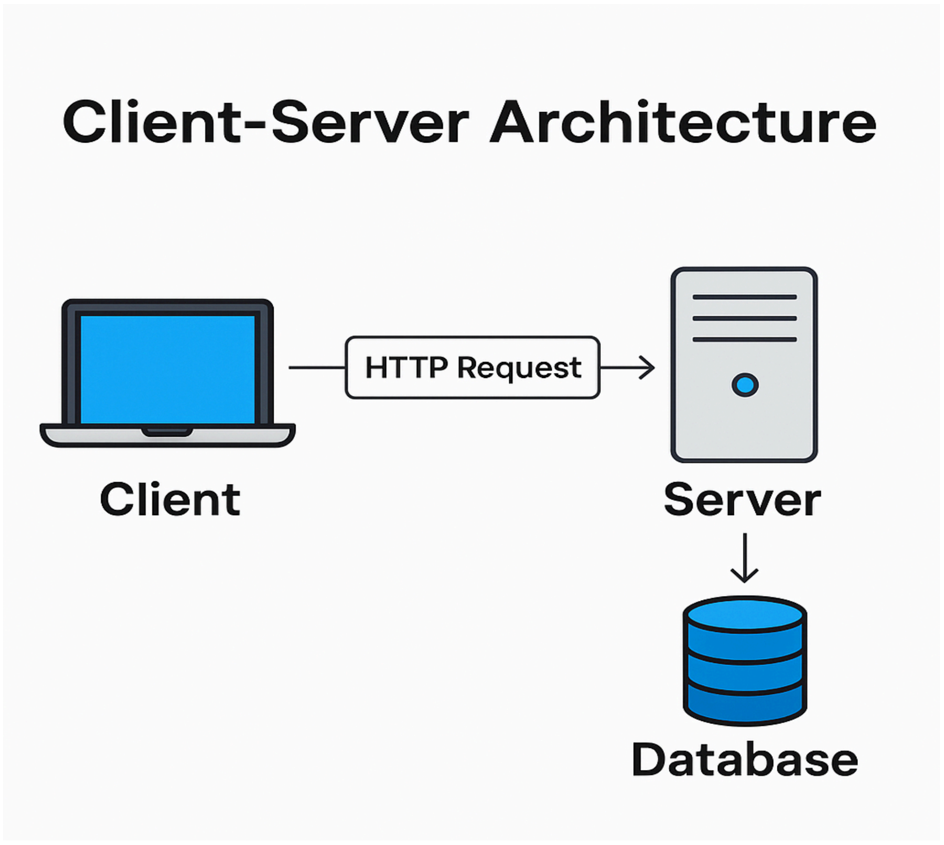
To make this possible, we need to understand how web applications work. We'll now enter the world of web development where we'll learn how to:

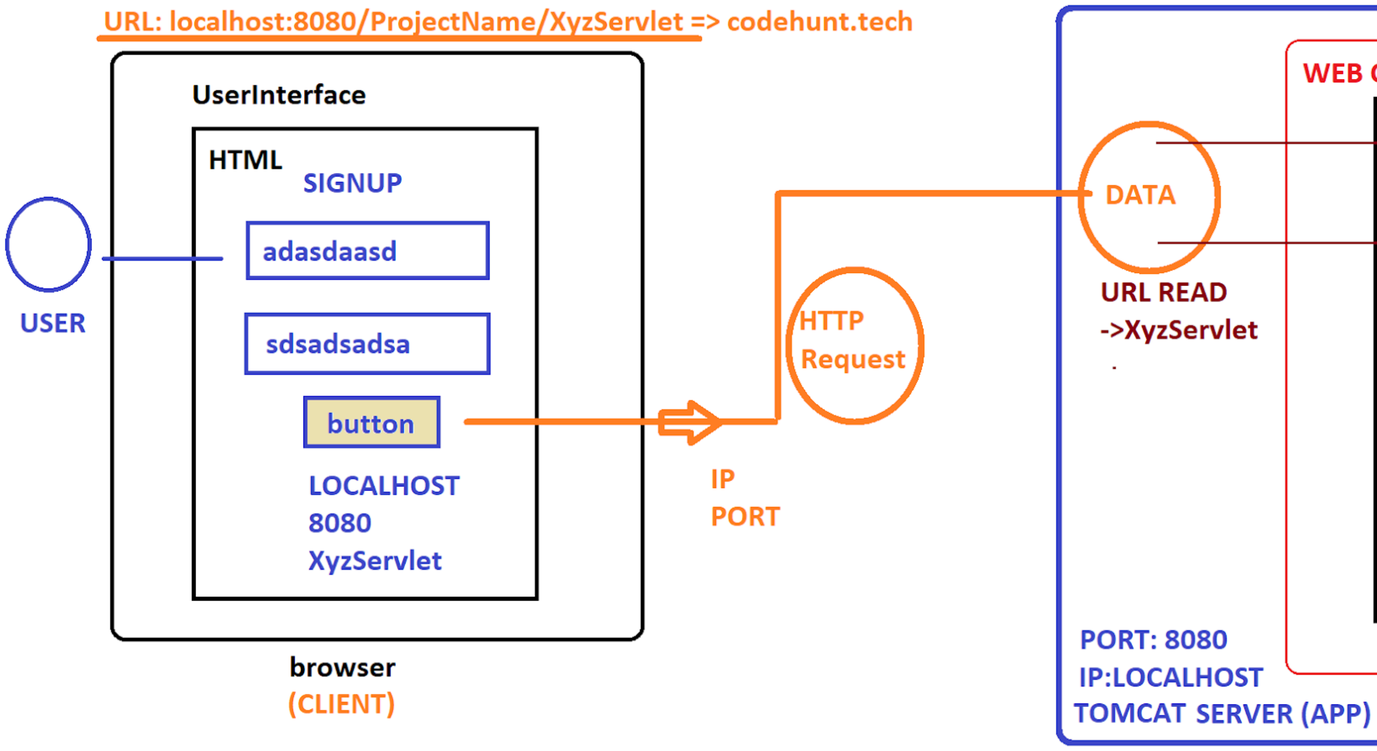
- Create a user interface using HTML (frontend)
- Send data from the browser to the backend
- Process that data using Java (backend)
- Store the data into a database

This brings us to an important concept: the Client-Server Architecture. Understanding this architecture is the foundation for building web applications. From this point onward, we'll start learning how web applications work.

What is Client-Server Architecture?

Whenever you use any website or app, you’re acting as the client. Your request goes to the server, which processes it and may talk to a database, then sends the result back to you. This is called Client-Server Architecture, and it’s the backbone of full stack development.


Client-Server Architecture
HTTP Request
Client
Server
Database


URL: localhost:8080/ProjectName/XyzServlet => codehunt.tech
UserInterface
HTML
SIGNUP
adasdaasd
sdsadsadsa
button
LOCALHOST
8080
XyzServlet
USER
browser
(CLIENT)
IP
PORT
HTTP
Request
DATA
URL READ
->XyzServlet
WEB
PORT: 8080
IP:LOCALHOST
TOMCAT SERVER (APP)

- A client is any device or program, like a web browser, that sends requests to a server to access information or services. When you type a website address or click a button on a webpage, the client(browser) creates and sends an HTTP request to the server asking for data or to perform an action. The client then waits for the server’s response and displays the result, such as showing a webpage or updating information on the screen. In web applications, the client is usually the user’s browser that interacts with the user and communicates with the backend through the server.
- Client request travels over the internet/network to the server’s IP address and port number where the web server is listening. The server continuously listens for incoming requests on that port. When the request arrives, the server reads and parses the HTTP request to understand what the

client wants. It then forwards the request to the right program (like a Servlet) that processes it, generates a response, and sends it back to the client.

- **A server is a special program or a separate computer that listens for requests from clients, like web browsers, over the internet or a network. When a user enters a website address or submits a form, the browser sends a request to the server. The server receives this request, processes it by finding the right program (such as a Servlet) to handle it, and then sends back the appropriate response, such as a web page or data.**
- **In web applications using Java, the server includes a web container that manages Servlets, runs them inside a Java Virtual Machine (JVM), and handles communication between the client and the backend code. Without a server, the browser cannot**

connect to the application or get any information.

✅ Proper Explanation:

- We create a UI (using HTML, CSS, JS) so that the client (browser) can interact with the application. Through this UI, users can send data to the application or request data from it.

- To handle these requests and process the data, we write backend code using Servlets. A Servlet is a Java program that runs on the server and handles HTTP requests like form submissions, login, data fetch, etc.

- However, a Servlet cannot run on its own. It needs a Server (like Apache Tomcat) to execute it. The server receives HTTP requests from the browser, identifies the correct Servlet based on the URL, and then calls the Servlet's method (doGet() or doPost()).

- Finally, the Servlet processes the request, sends a response back to the server, and the server returns that response to the client browser.

❓ When Will the Server Run a Servlet?

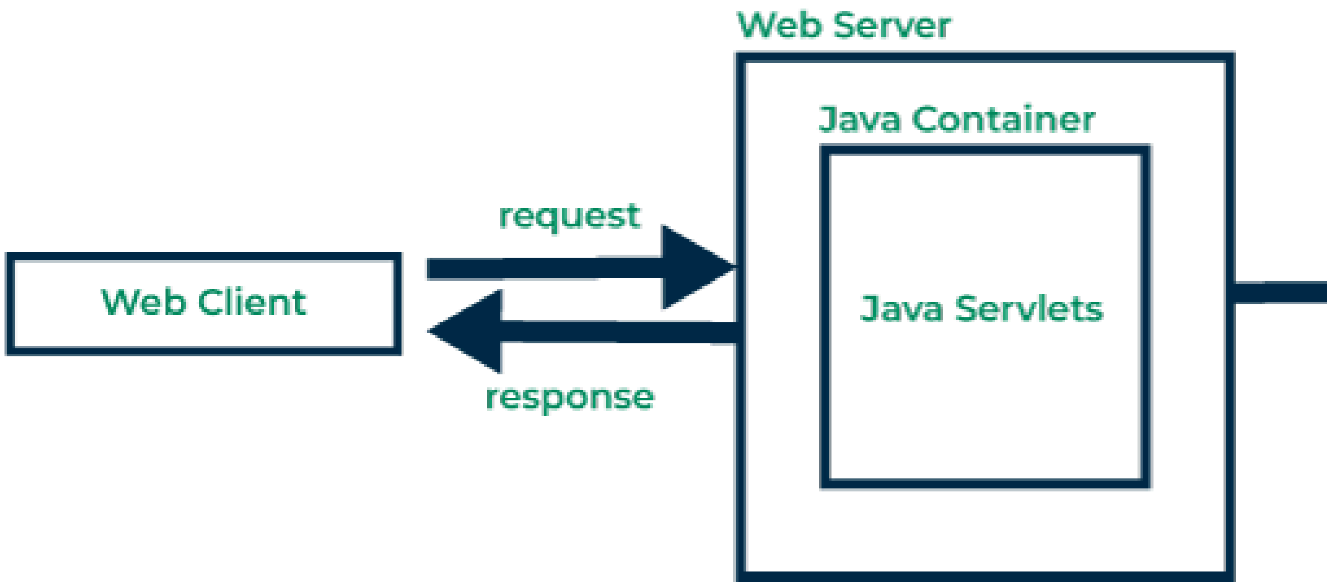
- The server runs a servlet only when a request comes from the browser that matches the servlet’s URL."

❓ Where Does a Servlet Run Inside a Server?

- A servlet runs inside a special part of the server called the Servlet Container or Web Container.

❓ What is this Servlet Container?

- It’s a part of the web server (like Tomcat, Jetty, or GlassFish).
- It manages the lifecycle of servlets: loading, Instantiation, initializing, calling methods, and destroying them.
- It also handles incoming HTTP requests and passes them to the correct servlet.



❓ How Web Container, Server, and JVM Work Together:

- The Web Container is a component inside the server (such as Apache Tomcat) that manages the execution of Servlets.
- When a request comes from the browser, the server along with the web container receives the request.
- The web container examines the URL in the request and determines which Servlet should handle it.
- The web container then asks the JVM to execute the Servlet’s Java code by invoking the appropriate method, such as doGet() or doPost().

CONCLUSION:

- ***A server acts like a connection point that any browser or client can connect to, as long as they know the server's URL (which includes IP address and port number) and have network access to it.***

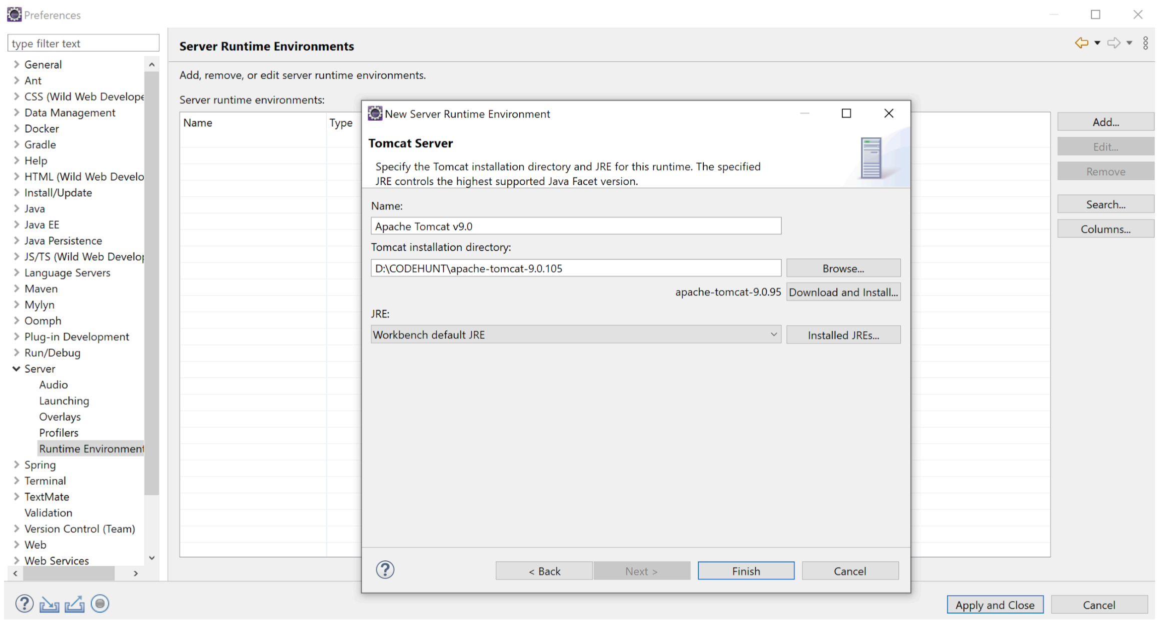
SERVER DOWNLOAD LINK:

https://tomcat.apache.org/download-90.cgi

Steps to Create a Dynamic Web Project in Eclipse

1. Change Perspective to Java EE

2. Configure Your Server



Add, remove, or edit server runtime environments.

Server runtime environments:

Tomcat Server

Specify the Tomcat installation directory and JRE for this runtime. The specified JRE controls the highest supported Java Facet version.

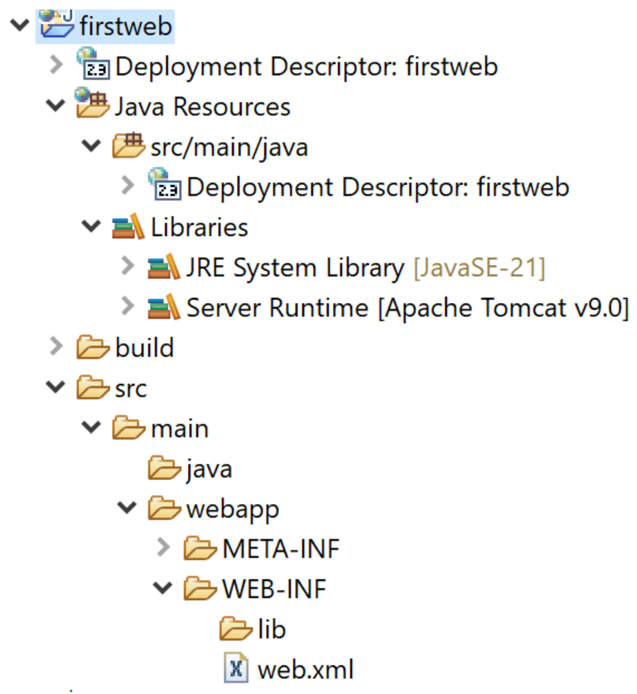
Name: Apache Tomcat v9.0

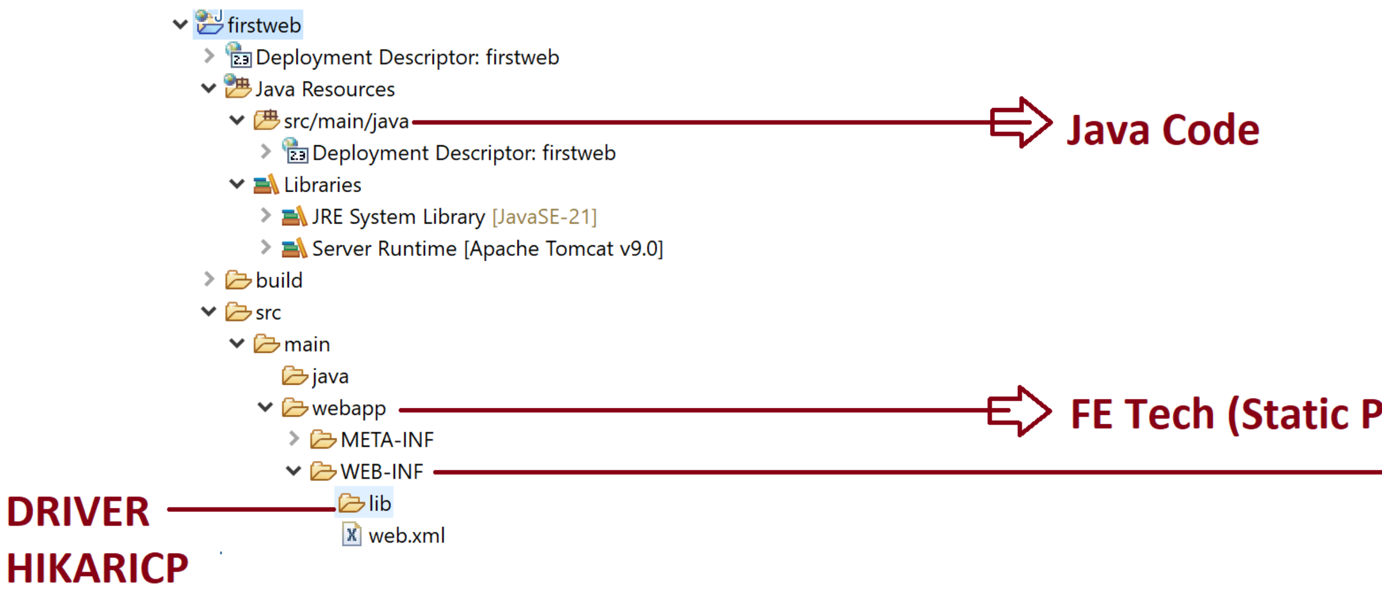
Tomcat installation directory: D:\CODEHUNT\apache-tomcat-9.0.105

apache-tomcat-9.0.95

JRE: Workbench default JRE

3. Create a Dynamic Web Project (web module 2.3 for xml support)


firstweb
Deployment Descriptor: firstweb
Java Resources
src/main/java
Deployment Descriptor: firstweb
Libraries
JRE System Library [JavaSE-21]
Server Runtime [Apache Tomcat v9.0]
build
src
main
java
webapp
META-INF
WEB-INF
lib
web.xml


firstweb
Deployment Descriptor: firstweb
Java Resources
src/main/java
Deployment Descriptor: firstweb
Libraries
JRE System Library [JavaSE-21]
Server Runtime [Apache Tomcat v9.0]
build
src
main
java
webapp
META-INF
WEB-INF
lib
web.xml

4. Create a Static HTML Page (e.g., index.html)

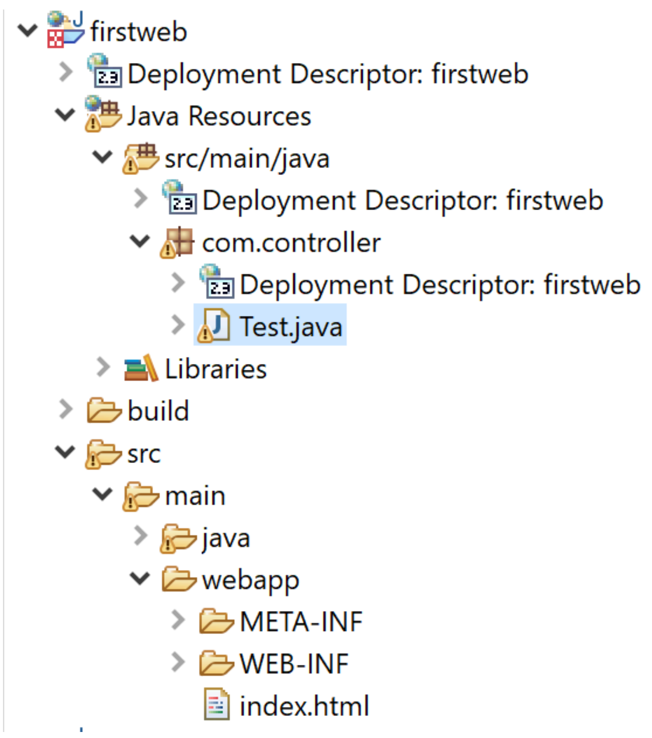
- **Right-click on the webapp folder.**

- Choose New → HTML File.
- Name it index.html.
- Add your HTML content inside the file.

5. Create a Servlet (Override doGet method)

- Right-click on the Java Resources/src folder.
- Choose New → Servlet.
- Enter the servlet class name (e.g., TestServlet).
- Define the package (optional).
- Click Finish.

```
================================PROG
RAM: FIRST WEB
PROJET==============================
=============
```



```
firstweb
  Deployment Descriptor: firstweb
  Java Resources
    src/main/java
      Deployment Descriptor: firstweb
      com.controller
        Deployment Descriptor: firstweb
        Test.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
        index.html
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h2>THIS IS MY WELCOME PAGE</h2>

    <form action="Test" method="get">
       <input type="text" name="fullname" placeholder="enter full
name"><br>
       <input type="text" name="address" placeholder="enter
address"><br>
       <button>send</button>
    </form>


</body>
</html>
```

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class Test extends HttpServlet {

	//IT CAN HANDLE THE DATA FROM HTTP REQUEST(GET)
	protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

		String fullname = request.getParameter("fullname");
		String address = request.getParameter("address");

		System.out.println(fullname+" "+address);

	}

}
```

==================================

==================================

=============================

DAY 11:

What is the Servlet?

- **The Servlet API is a set of interfaces, classes, and methods provided by Java EE (Jakarta EE) that allows developers to write server-side programs (Servlets) that handle HTTP requests and responses.**

Packages in the Servlet API:
javax.servlet (or jakarta.servlet in newer versions)

- The implementations of the Servlet API are provided by Servlet Containers (also called Web Servers or Application Servers).
- A Servlet is a Java class used to handle HTTP requests and responses in a web application. It runs on a Java-enabled web server (like Apache Tomcat) and follows the Java EE (Jakarta EE) standard.

Types of HTTP Request

There are many types of HTTP requests, but the two most important are GET and POST.

GET Request

- Used to retrieve data from the server.
- Sends data via URL query parameters.

- Can be bookmarked because all data is in the URL.
- Can be cached by browsers and proxies.
- Not suitable for sensitive data (data visible in URL).

POST Request

- Used to send data to the server to create or update resources.
- Sends data in the request body.
- Cannot be bookmarked because data is not in the URL.
- Not cached by browsers by default.
- Better for sensitive data.

Q: What is the difference between the doGet() and doPost() methods in a servlet?

- The doGet() method in a servlet handles HTTP GET requests. It is used when the client wants to retrieve data from the server without changing anything. For example, when you open a webpage or request some information, the servlet’s doGet() method processes that request.

- The doPost() method handles HTTP POST requests, which are used when the client wants to send data to the server to create something, such as submitting a form or uploading data.

Q: What is HttpServletRequest object in a servlet?

- The HttpServletRequest object is used to get data from the client (browser). For example, if a user submits a form, you can use request.getParameter("name") to get the value entered in the "name" field. It basically gives the servlet access to all the information the client has sent.

Q: What is HttpServletResponse object in a servlet?

- The HttpServletResponse object is used to send data back to the client. It allows the servlet to create a response, set the content type (like HTML or JSON), write output to the browser, or even redirect the user to another page. For example, response.getWriter().println("Hello") sends text to be displayed on the browser. It controls what the user sees as the result of their request.

Q: Why do I get a blank page if my servlet doesn’t use the response object?

- When a servlet is hit, it is expected to send something back to the browser using the HttpServletResponse object. If the servlet doesn't write anything to the response (like response.getWriter().println(...)), then the server sends an empty HTTP response body. That’s why the browser shows a blank page — it received no content to display.

Here’s why response.getWriter().print(...)

directly is not the best approach:

Mixes logic and view:

- Writing HTML directly in Java (inside servlets) mixes the controller with the view, which breaks the separation of concerns. This makes the code messy and hard to manage as your project grows.

Hard to maintain or update UI:

- Updating the UI means editing Java code, which is inefficient and error-prone?

Better alternatives exist:

- In a proper MVC setup, servlets should process data, then forward the result to a JSP page or frontend view for rendering.

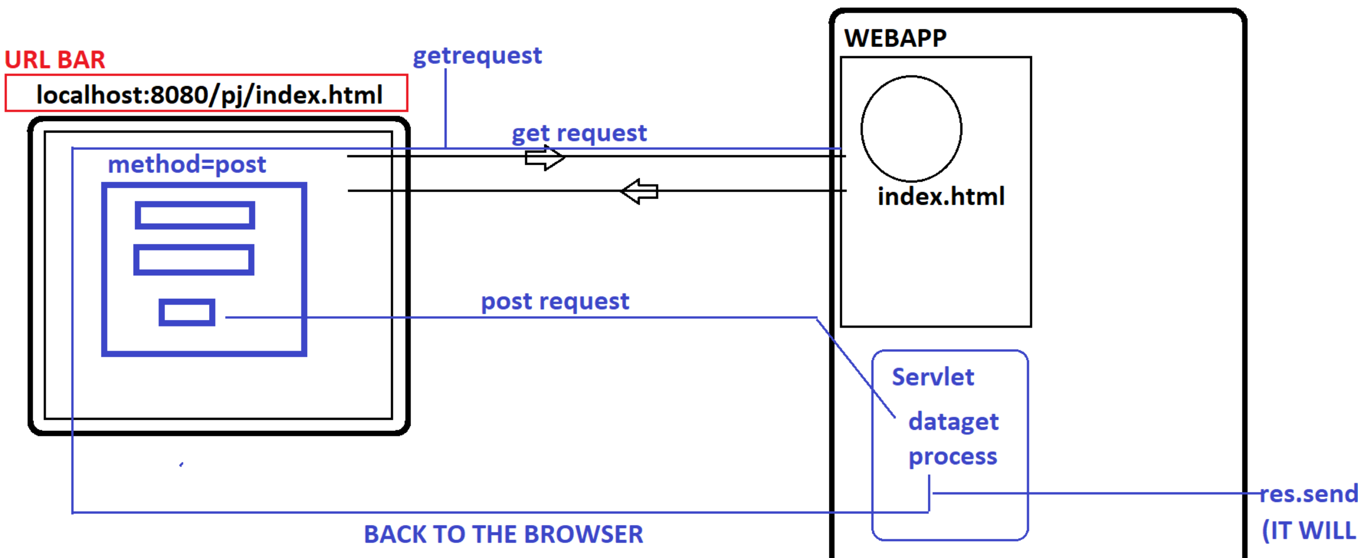
NOTE:

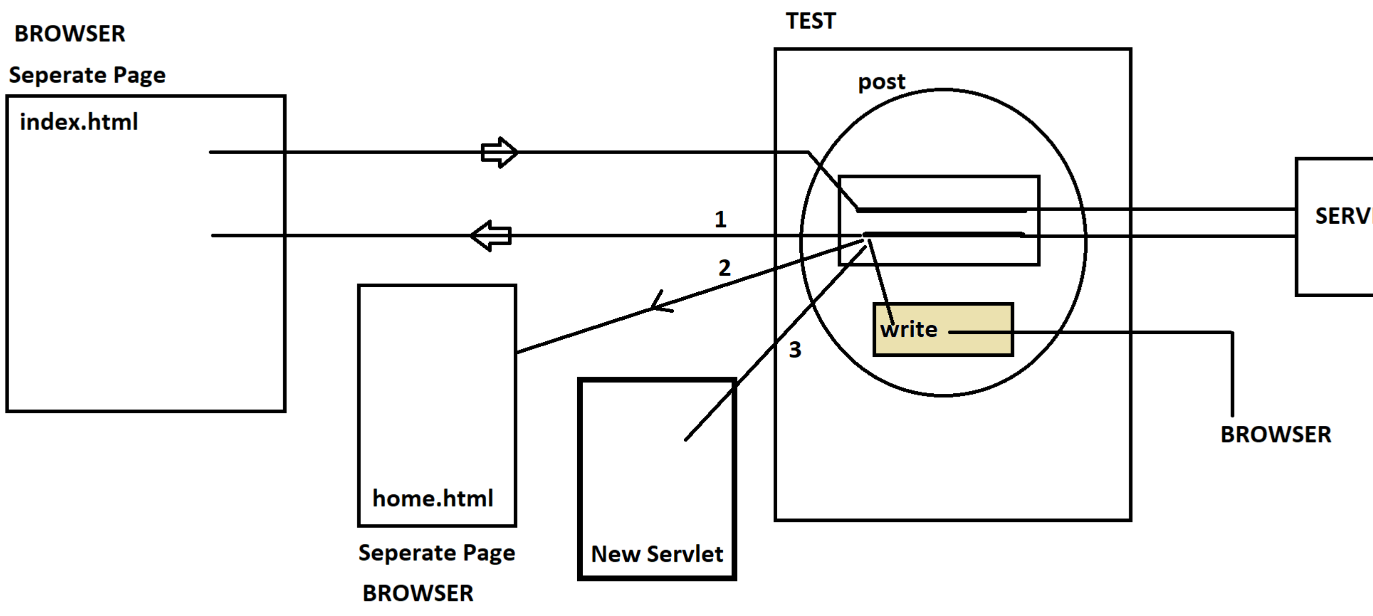
You can use the response.getWriter().print(...) method when you're working on very small applications, demos, or simply testing servlet behavior. It's useful when you just want to quickly display a message or a simple HTML response without setting up a full JSP or frontend. However, this approach is only suitable for basic output and not recommended for larger or real-world projects, as it mixes Java code with HTML, making the application harder to maintain.

response.sendRedirect()

- It sends a redirect response to the client (browser), telling it to make a new request to a different URL.
- Because it’s a new request, the URL in the browser changes to the new address.
- It can redirect to any URL, including sites outside your application.

- **The original request and response objects are not preserved; a brand new request is created.**
- **It causes an extra round-trip between client and server (client has to send a new request).**



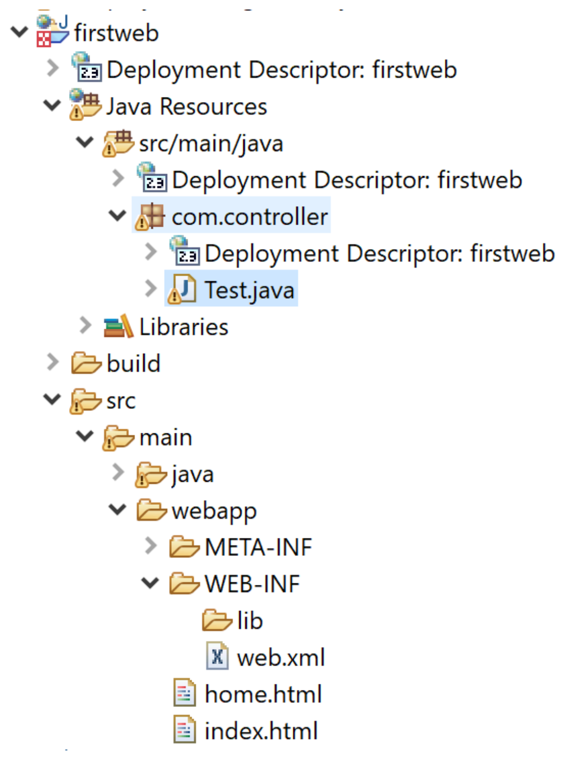


When you call res.sendRedirect("MyServlet") in a Servlet, the request does NOT directly go to

MyServlet on the server side. Instead, here's what happens:

1. **The server sends an HTTP redirect response back to the client (browser), telling it to go to the URL MyServlet.**
2. **The client browser then makes a new HTTP request to MyServlet.**
3. **The server receives this new request and processes it by calling the MyServlet.**

```
==============================PROG
RAM:
sendRedirect()=====================
==================
```



```
firstweb
  Deployment Descriptor: firstweb
  Java Resources
    src/main/java
      Deployment Descriptor: firstweb
      com.controller
        Deployment Descriptor: firstweb
        Test.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
          web.xml
        home.html
        index.html
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
    <h2>THIS IS MY WELCOME PAGE</h2>

    <form action="Test" method="post">
       <input type="text" name="fullname" placeholder="enter full
name"><br>
       <input type="text" name="address" placeholder="enter
address"><br>
       <button>send</button>
    </form>

</body>
</html>
```

Test.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
```

```java
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Test extends HttpServlet {

      //IT CAN HANDLE THE DATA FROM HTTP REQUEST(GET)
      @Override
      protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {

            String fullname = request.getParameter("fullname");
            String address = request.getParameter("address");
            System.out.println(fullname+" "+address);

            //TO PRINT ON BROWSER
//          PrintWriter writer = response.getWriter();
//          writer.print("<h1>SUCCESS</h1>");

            //ArrayList
            response.sendRedirect("home.html");
      }
}
```

home.html

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<h1>SUCCESS</h1>
</body>
</html>
```

==================================

==================================

=======================

DAY 12:

Servlet life cycle

Servlet Life Cycle is the process that a servlet goes through from the time it is loaded by the server until it is destroyed. It is managed by the servlet container (like Apache Tomcat) and involves several stages that allow the servlet to handle requests and release resources properly.

The main phases of the Servlet Life Cycle are:

Loading and Instantiation:

- **The servlet container loads the servlet class and creates an instance (object) of the servlet when the servlet is first requested or when the server starts (depending on configuration).**

Initialization (init() method):

- After creating the servlet instance, the container calls the init() method once. This method is used to perform any servlet initialization tasks, such as setting up resources (database connections, reading config, etc.).

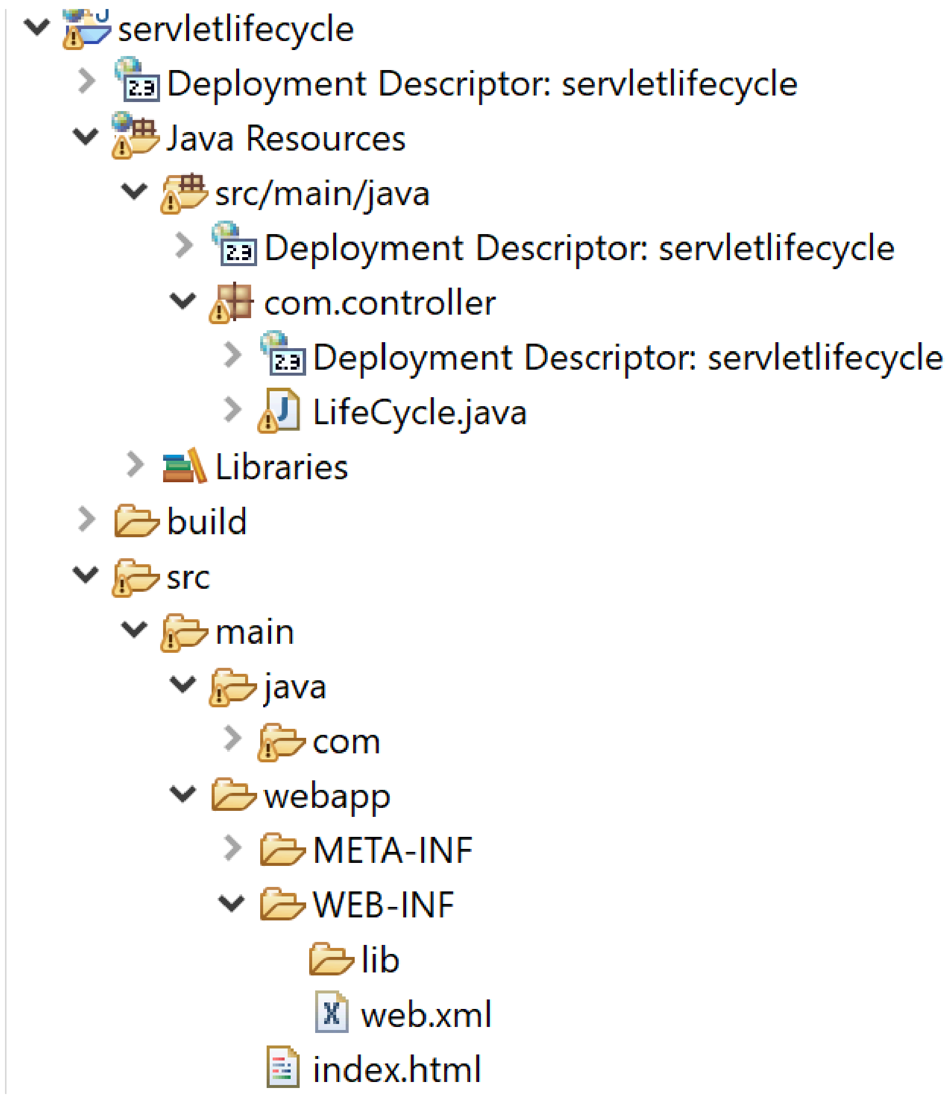
Request Handling (service() method):

- For each client request, the servlet container calls the service() method. This method determines the type of HTTP request (GET, POST, etc.) and calls the appropriate method (doGet(), doPost(), etc.) to process the request.

Destruction (destroy() method):

- When the servlet is no longer needed (usually when the server is shutting down or the application is undeployed), the container calls the destroy() method once. This method is used to clean up resources before the servlet is removed from memory.

=============================PROGRAM: Servlet Life Cycle=====================================


```
servletlifecycle
  Deployment Descriptor: servletlifecycle
  Java Resources
    src/main/java
      Deployment Descriptor: servletlifecycle
      com.controller
        Deployment Descriptor: servletlifecycle
        LifeCycle.java
    Libraries
  build
  src
    main
      java
        com
      webapp
        META-INF
        WEB-INF
          lib
          web.xml
        index.html
```

```
<!DOCTYPE html>
```

```html
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <a  href="LifeCycle" >INVOKE MY SERVLET</a>

</body>
</html>
```

```java
package com.controller;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LifeCycle extends HttpServlet {

	static{
		System.out.println("SERVLET LOADING");
	}

    public LifeCycle() {
      System.out.println("SERVLET INSTANTIATION");
    }

	public void init(ServletConfig config) throws ServletException
{
		//CONNECTION->
		System.out.println("RESOURCE ALLOCATION");
	}

	protected void service(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
		System.out.println("SERVICE");
	}

	public void destroy() {
		System.out.println("RESOURCE DE-ALLOCATION");
	}

}
```

====================================
====================================
========================

Servlet Collaboration

- Servlet Collaboration means communication or interaction between multiple servlets to share data or forward a request. In real-world web applications, it's very common that one servlet performs part of the work and then passes the request to another servlet or resource (like a JSP) to complete the response.

- In Servlet Collaboration, RequestDispatcher is a key interface that helps one servlet communicate or work with another servlet, JSP, or resource within the same web application.

- The RequestDispatcher interface in Servlet is used for servlet collaboration, and it provides two important methods:
   1. forward(request, response)
   2. include(request, response)

1. forward(request, response)

What it does:

- Transfers control completely from the current servlet or JSP to another resource (like another servlet, JSP, or HTML). Only the target resource generates the response.

How it works:

- Once forward() is called, the current servlet stops executing, and the target resource takes over and generates the full response.

Browser URL:

- The URL does not change because everything happens on the server side.

2. include(request, response)

What it does:

- Includes the output of another resource (servlet, JSP, HTML) inside the current response. Both resources contribute to the final output.

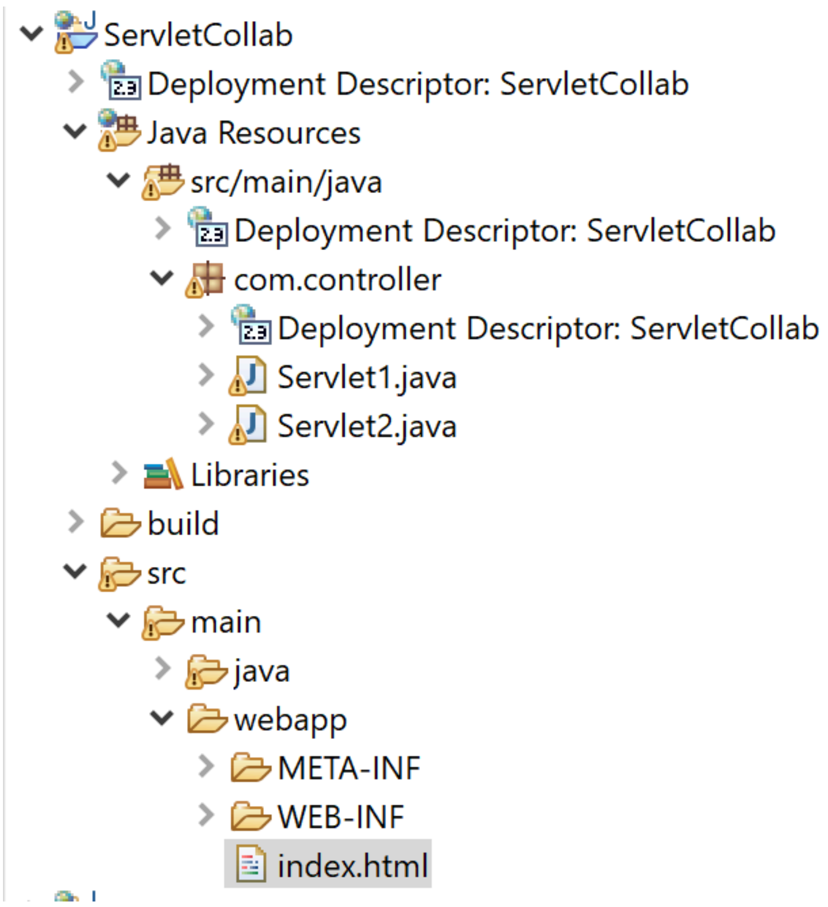
How it works:

- The included resource runs, and its output is inserted at the point where include() is called. After that, the current servlet continues executing.

Browser URL:

- Again, the URL does not change, as it is server-side inclusion.

============================PROGRAM: Servlet Collaboration==================================



```
ServletCollab
  Deployment Descriptor: ServletCollab
  Java Resources
    src/main/java
      Deployment Descriptor: ServletCollab
      com.controller
        Deployment Descriptor: ServletCollab
        Servlet1.java
        Servlet2.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
        index.html
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <form  action="Servlet1" method="post">
        <input type="text" name="name" placeholder="ENTER
NAME"><br>
        <input type="text" name="address"
placeholder="ENTER ADDRESS"><br>
        <button>send</button>
   </form>

</body>
</html>
```

```java
package com.controller;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Servlet1 extends HttpServlet {

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

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

		System.out.println("VALIDATE NAME &
ADDRESS"+name+"||"+address);

		response.getWriter().print("VALIDATION
SUCCESSFULL<br>");

		RequestDispatcher rd =
request.getRequestDispatcher("Servlet2");
		//rd.forward(request, response);
		rd.include(request, response);

	}
}


package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Servlet2 extends HttpServlet {
    protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {

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

            System.out.println("DATABASE LOGIC
CALLED"+name+"||"+address);

            response.getWriter().print("DATA PERSISTENCY
SUCCESSFULL");
    }

}
```

====================================
====================================
=========================

ServletConfig vs ServletContext

- **In a Java web application, both ServletConfig and ServletContext are used to pass configuration information, but they serve different purposes.**

1. ServletConfig

What is it?

- It is specific to a single servlet.
- Used to pass initialization parameters to a particular servlet through the web.xml.
- It's created by the container when the servlet is initialized.
- USE CASE: When you want to pass some custom configuration to only one servlet.

2. ServletContext

What is it?

- It is common to all servlets in a web application.
- Used to store and share global data and configuration accessible by all servlets.
- Created once per application when the web app is deployed.

- **Use Case: When you want to share data between multiple servlets, or define global configuration**
- **You can also set attributes dynamically: context.setAttribute("count", 1);**

==========================PROGRAM: ServletContext & ServletConfig==========================

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <a href="Servlet1" >SERVLET1</a>
    <a href="Servlet2" >SERVLET2</a>

</body>
</html>
```

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class Servlet1 extends HttpServlet {

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

		ServletConfig servletConfig = getServletConfig();
		String value1 =
servletConfig.getInitParameter("url");
		System.out.println(value1);


		ServletContext servletContext =
getServletContext();
		String value2 =
servletContext.getInitParameter("key");
		System.out.println(value2);

		servletContext.setAttribute("username", "root");

		response.sendRedirect("index.html");

	}
}
```

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Servlet2 extends HttpServlet {

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

```
         ServletConfig servletConfig = getServletConfig();
         String value1 =
servletConfig.getInitParameter("url");
         System.out.println(value1);

         ServletContext servletContext =
getServletContext();
         String value2 =
servletContext.getInitParameter("key");
         System.out.println(value2);

         String value3 = (String)
servletContext.getAttribute("username");
         System.out.println(value3);

         response.sendRedirect("index.html");

     }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD
Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
     <display-name>ContextConfig</display-name>

     <context-param>
           <param-name>key</param-name>
            <param-value>#^$%^TYGGHJHJHJ</param-value>
     </context-param>

     <servlet>
          <servlet-name>Servlet1</servlet-name>
          <display-name>Servlet1</display-name>
          <description></description>

<servlet-class>com.controller.Servlet1</servlet-class>

          <init-param>
            <param-name>url</param-name>
```

```
<param-value>jdbc:mysql://localhost:3306</param-value>
        </init-param>

    </servlet>
    <servlet>
        <servlet-name>Servlet2</servlet-name>
        <display-name>Servlet2</display-name>
        <description></description>

<servlet-class>com.controller.Servlet2</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Servlet1</servlet-name>
        <url-pattern>/Servlet1</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>Servlet2</servlet-name>
        <url-pattern>/Servlet2</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

==

==

=============================

DAY 13:

Servlet object explained simply:

- When your web application starts, the Servlet container (like Tomcat) creates one instance of each Servlet class (including the servlet generated from your JSP).
- This single Servlet object is shared for all incoming requests.
- For every user request, the server calls the servlet's service method (or _jspService for JSPs) — multiple threads handle multiple requests concurrently, all using that same Servlet object.

Q: Is the service() method always called by the servlet container even if I have only implemented doPost() in my servlet?

- Yes, the servlet container always calls the service() method to handle every HTTP request.

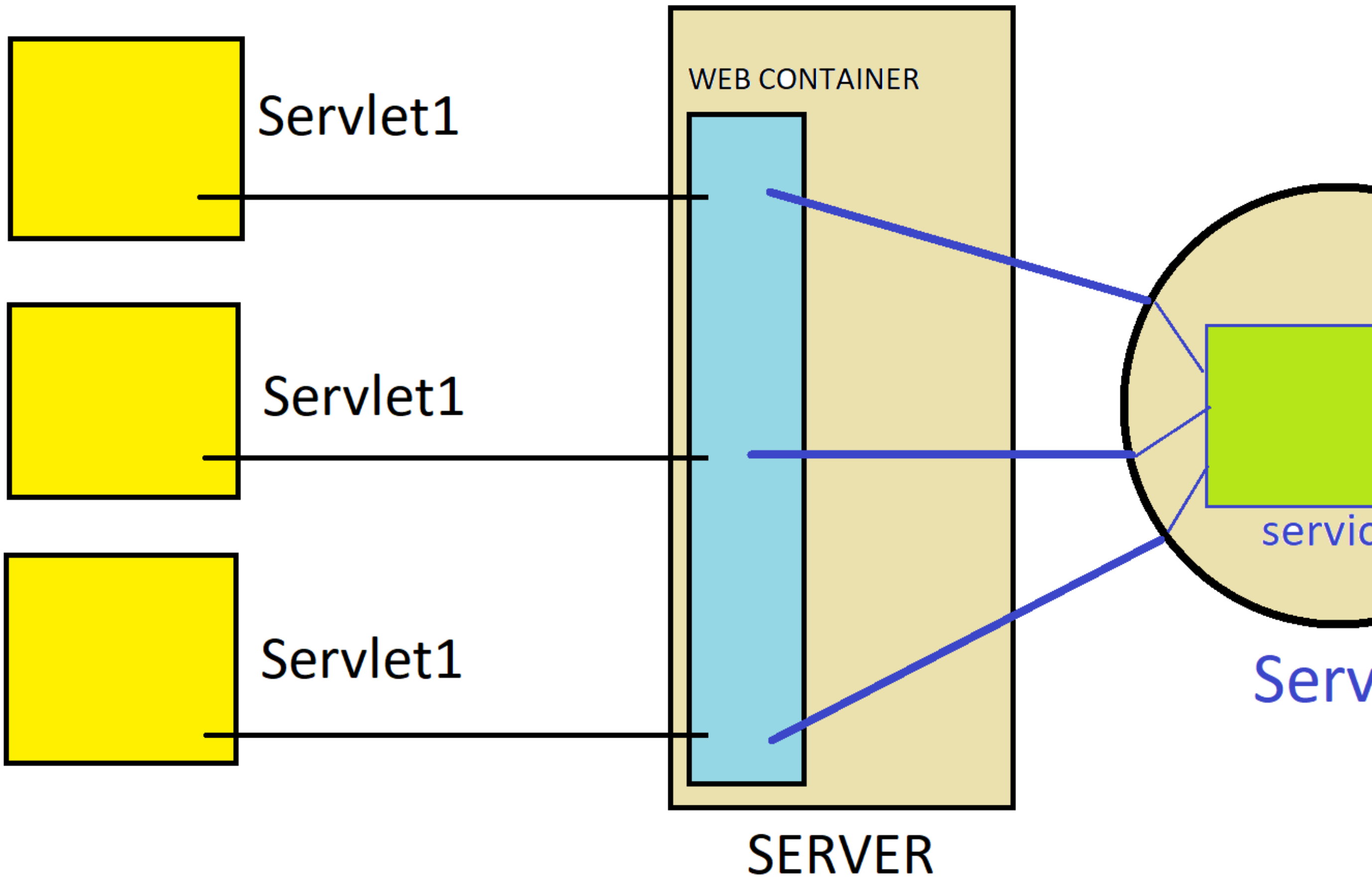
How it works:

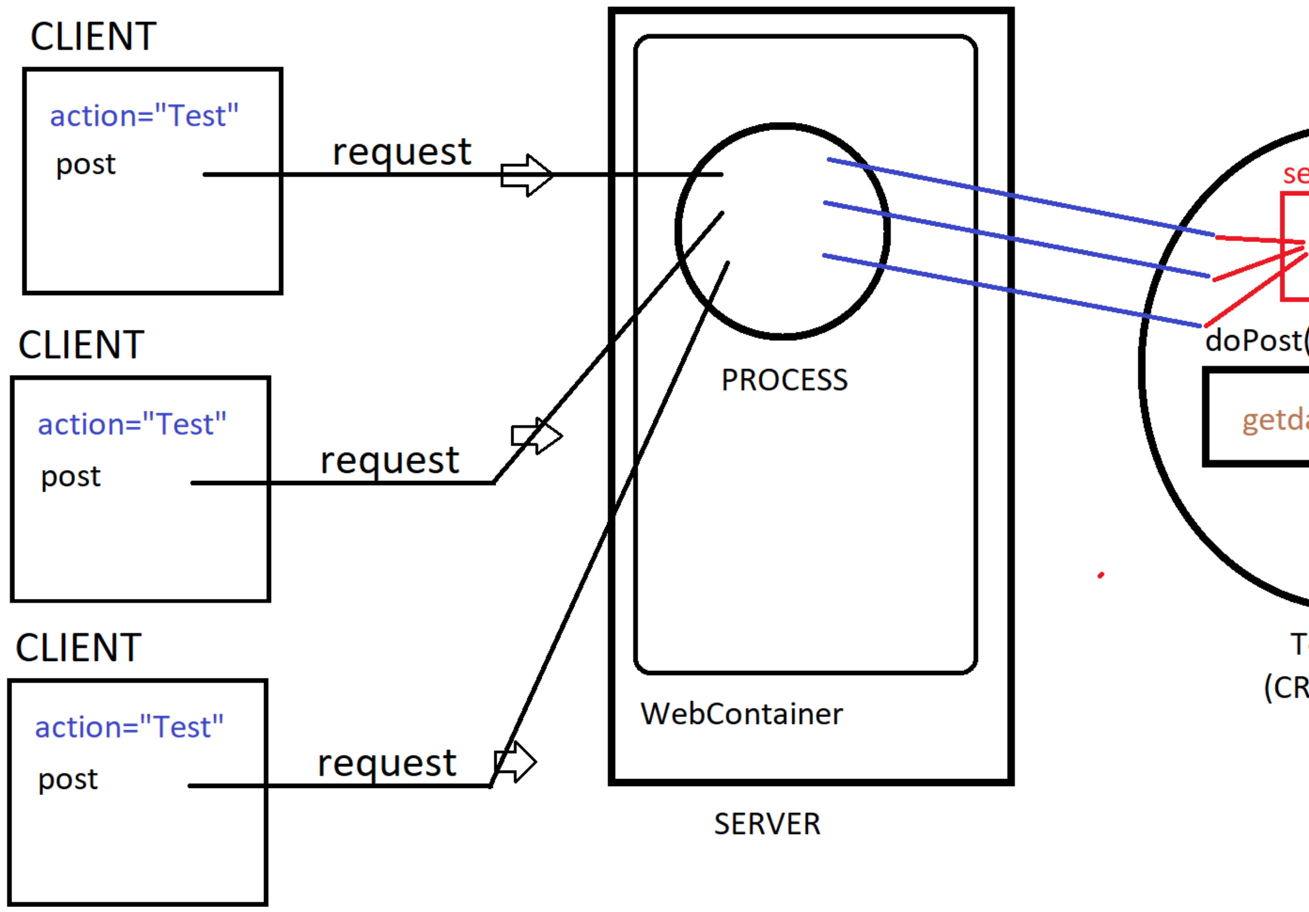
- When a request arrives, the container invokes the servlet’s service() method.
- The default service() method in HttpServlet checks the HTTP method (GET, POST, etc.) of the request.
- It then calls the corresponding method like doGet(), doPost(), doPut(), etc.
- If you have only implemented doPost(), only POST requests will be handled by your code.

- Requests with other methods (like GET) will be handled by the default implementations in HttpServlet, which usually return an HTTP 405 error (Method Not Allowed).

Who creates the threads for the servlet?

- The Servlet container (e.g., Tomcat, Jetty) is responsible for creating and managing threads.
- When the server receives multiple HTTP requests, it assigns each request to a separate thread from its internal thread pool.
- These threads call the same servlet instance's service method concurrently.
- The service() method (and for JSPs, the generated _jspService() method) is NOT synchronized by default.

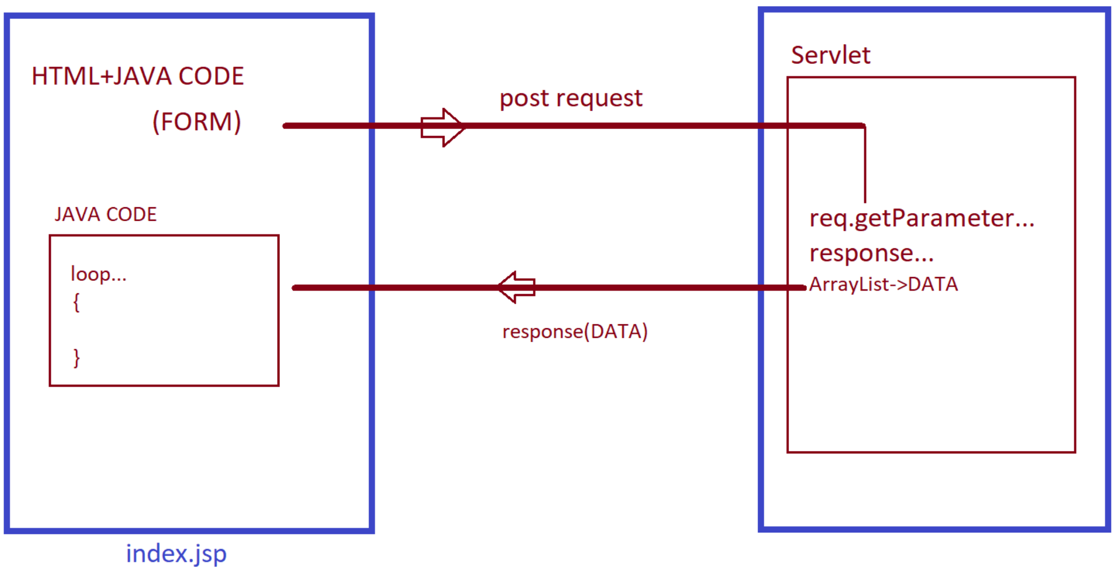

Servlet1
Servlet1
Servlet1
WEB CONTAINER
SERVER
servic
Serv



JAVA SERVER PAGE (JSP)

What is JSP?

JSP (JavaServer Pages) is a server-side technology that allows you to write HTML + Java code in the same file. It's used to create dynamic web pages.



When you hit the URL: http://localhost:8080/myproject/index.jsp

- **It looks like you're accessing a JSP file directly from the browser, but here's what's actually happening behind the scenes:**

What Actually Happens?

1.The browser sends an HTTP request(GET)

- **Example: http://localhost:8080/myproject/index.jsp is requested by the browser.**

2.Tomcat (Servlet Container) receives the request

- **Tomcat handles all web components like Servlets and JSPs.**

3.The JSP container (part of the Servlet container) takes over

- **It is specifically responsible for managing JSP files.**

4.The JSP file is located and translated into a Java Servlet class

- **This happens only the first time or if the JSP was modified.**
- **The generated servlet class includes a method called _jspService(), which contains all the logic written in the JSP file.**

5.The translated Java Servlet is compiled into bytecode (a .class file)

- **This servlet class now behaves like any other servlet.**

6.The Servlet (from the JSP) is executed by the container

- **The container creates a thread and calls the _jspService(HttpServletRequest req, HttpServletResponse res) method.**
- **This method contains all your scriptlets, expressions, HTML, and logic from the JSP page.**

NOTE: When your JSP is translated into a servlet by the JSP container, all the HTML code you wrote in the JSP file becomes Java print statements (like out.print() calls) inside the _jspService() method.

```
So, for example:

<html>
```

```
  <body>
    <p>Hello <%=
request.getParameter("name") %></p>
  </body>
</html>

Becomes something like:

public void _jspService(HttpServletRequest
request, HttpServletResponse response) {
    JspWriter out = ...;
    out.write("<html>\n");
    out.write("  <body>\n");
    out.write("    <p>Hello ");

out.write(request.getParameter("name"));
    out.write("</p>\n");
    out.write("  </body>\n");
    out.write("</html>");
}
```

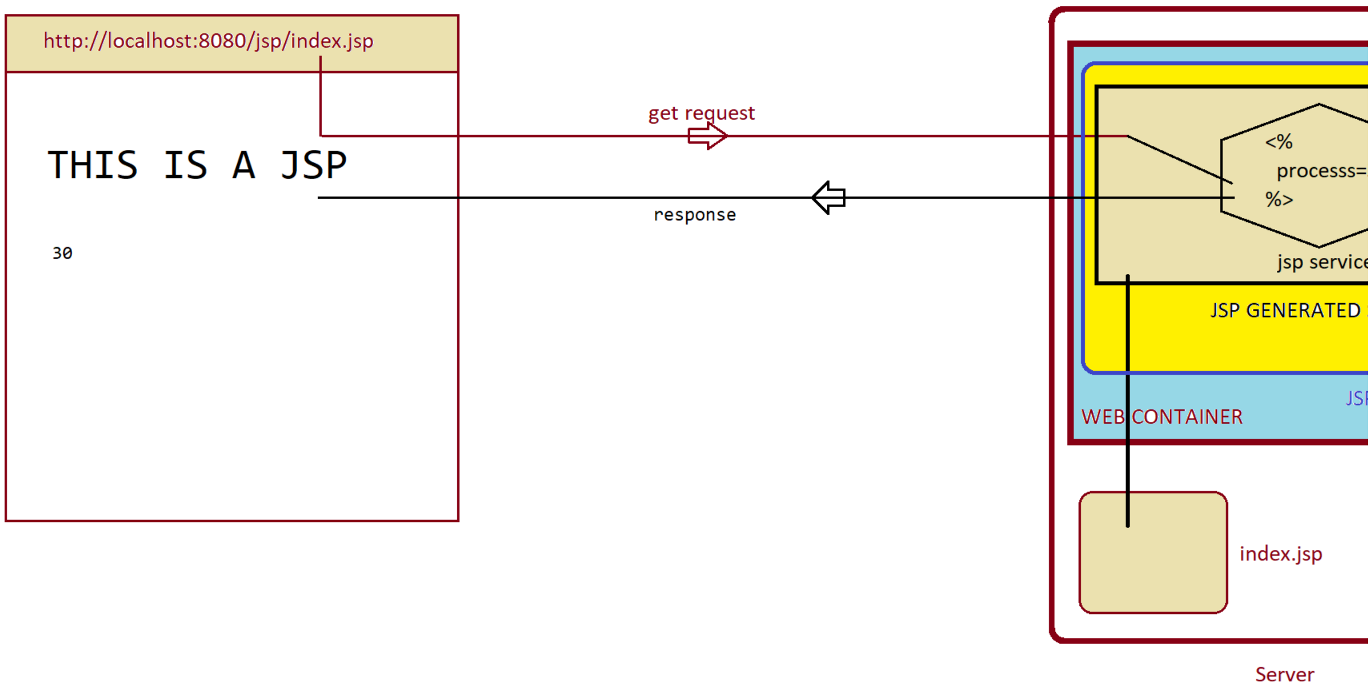
7.The out object inside _jspService() writes HTML output to the response

- **Any dynamic content created using Java is added to the HTML.**

9.The server sends the generated HTML as a response back to the browser

10.The browser renders only the HTML

- It never sees the JSP file or the Java code — only the final output.



JSP TAGS

In JSP, we use different types of tags to mix Java code with HTML and control how the page behaves. There are many tags available, but here are the main ones you should know: *directives, scriptlets, expressions, and declarations*. Each of these plays a specific role in how the JSP page is processed and how Java code interacts with the page content.

1. Directives (LATER)

Directives tell the JSP container how to process the page during compilation. For example, you can import Java classes or set page-level settings. They do not generate any output themselves but configure your JSP.

- Page Directive : Used to define page-level settings like imports, content type, error handling, etc.
  ```
  <%@ page import="java.util.*" %>
  ```
- Include Directive: Used to include static content (like HTML, JSP, or Java code) at translation time.
  ```
  <%@ include file="header.jsp" %>
  ```

- **Taglib Directive: Used to include custom tags or JSTL (JSP Standard Tag Library).**
  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

2. Scriptlets

Scriptlets let you write Java code that runs when the page is requested. You write them inside <% ... %> tags. This is where you can perform calculations, set variables, or control the flow with loops and conditions.

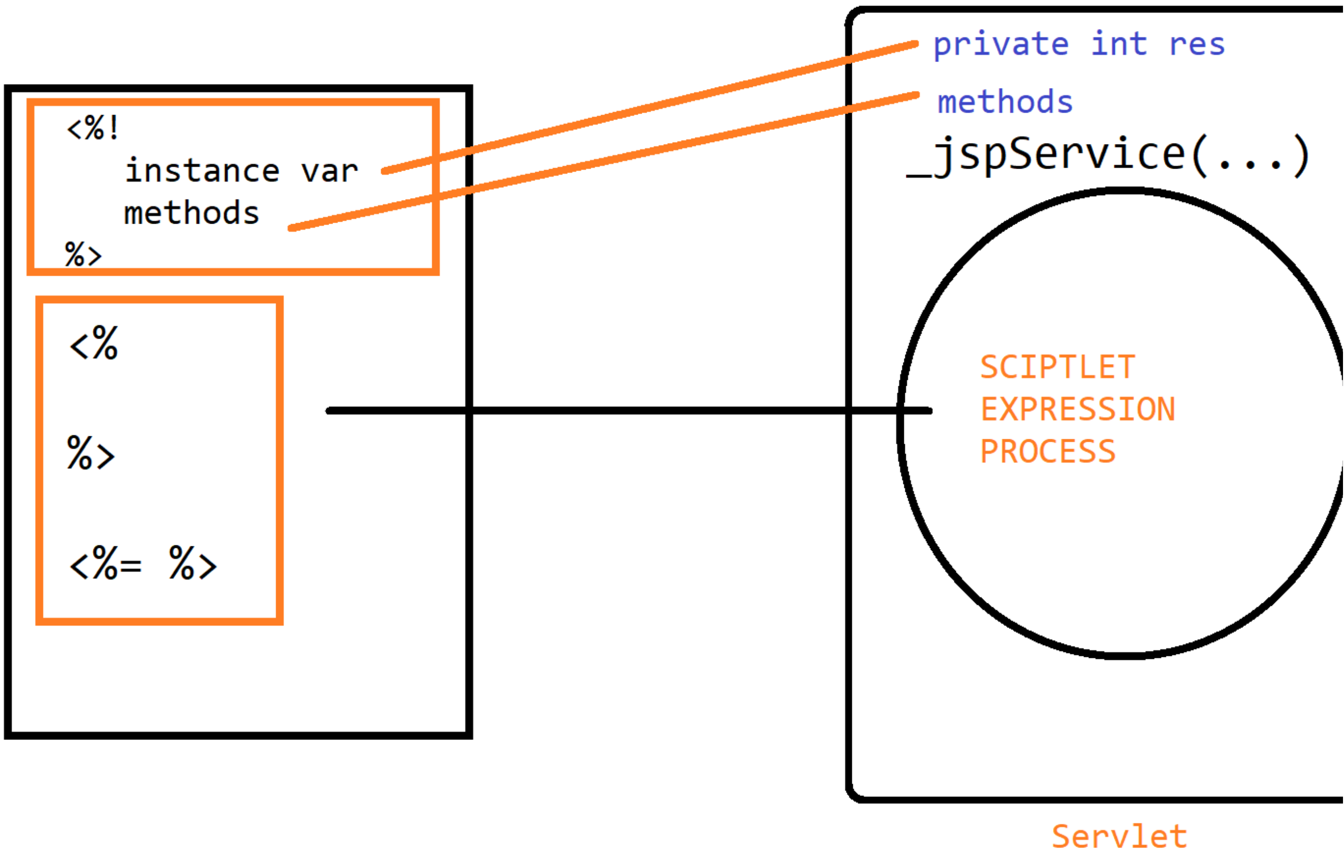
3. Expressions

Expressions output the result of Java code directly into the HTML page. They are shorter and easier for printing values and are written inside <%= ... %> tags.

4. Declarations/Declarative

Declarations allow you to define variables and methods that become part of the servlet class

generated from your JSP. These are shared across all requests, so use them carefully.



- **JSP declarations are blocks of Java code inside <%! ... %> tags that let you declare variables and methods at the class level in the servlet generated from your JSP.**
- **Unlike scriptlets (which go inside the _jspService() method), declarations become fields or methods at the class level of the generated servlet. This means they are defined once when the servlet class loads.**

- Scriptlets are used to write Java statements that go inside the _jspService() method of the servlet generated from your JSP so you can write statements, expressions, loops, and conditionals — but you cannot declare new methods inside another method in Java.
- These variables and methods become instance members of the servlet class — meaning they belong to the single servlet object instance that the server creates to handle all requests.
- Since the servlet container creates only one instance of the servlet class, these declaration variables and methods are shared across all user requests and sessions.
- This sharing means if you use declarations to store mutable data (like counters or lists), you risk concurrency issues(RACE CONDITION) such as inconsistent values or data corruption unless you carefully manage thread safety.

DAY 14:

❓ Question:

If the servlet container creates only one servlet object, then how are 5 different requests handled?

Even though the servlet container creates only one instance of a servlet, it can handle multiple requests simultaneously using multi-threading.

Here’s how it works:

Single Servlet Object:

- The servlet container creates one servlet object per servlet class.

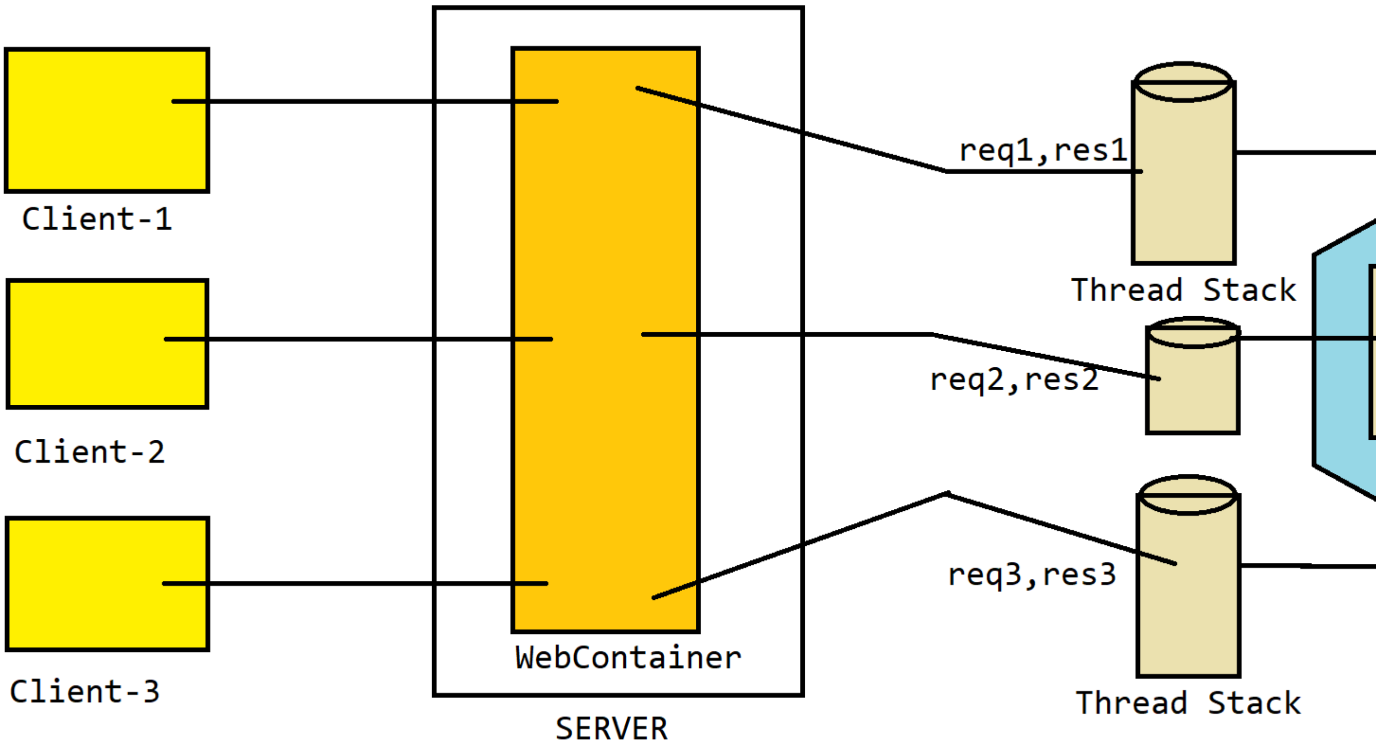
Multiple Threads:

- **When 5 requests come in, the container creates 5 separate threads. Each thread calls the service() method on the same servlet object.**

- **Although the service() method is part of the same object, it is executed in separate memory areas (called thread stacks). So, each request is handled independently and in parallel.**

Unique Request and Response Objects:

- **Each request gets its own HttpServletRequest and HttpServletResponse objects. These objects contain data specific to that request and are not shared.**



What is MVC?

MVC stands for:

- **Model – View – Controller, a design pattern used to separate business logic, UI, and request handling in web applications.**

1. Model

- Handles data, Handles Database Interaction, BusinessLogic
- Example: DTO, Entity

2. View
   - Represents the user interface.
   - Example: JSP, Thymeleaf, HTML pages.
3. Controller
   - Accepts incoming requests, processes them and forwards data to the view.
   - Example: Servlet

❓ Can you call a Dynamic JSP directly from the browser URL?

Technically: Yes

- You can call a JSP directly using a URL like: http://localhost:8080/your-app/view/home.jsp

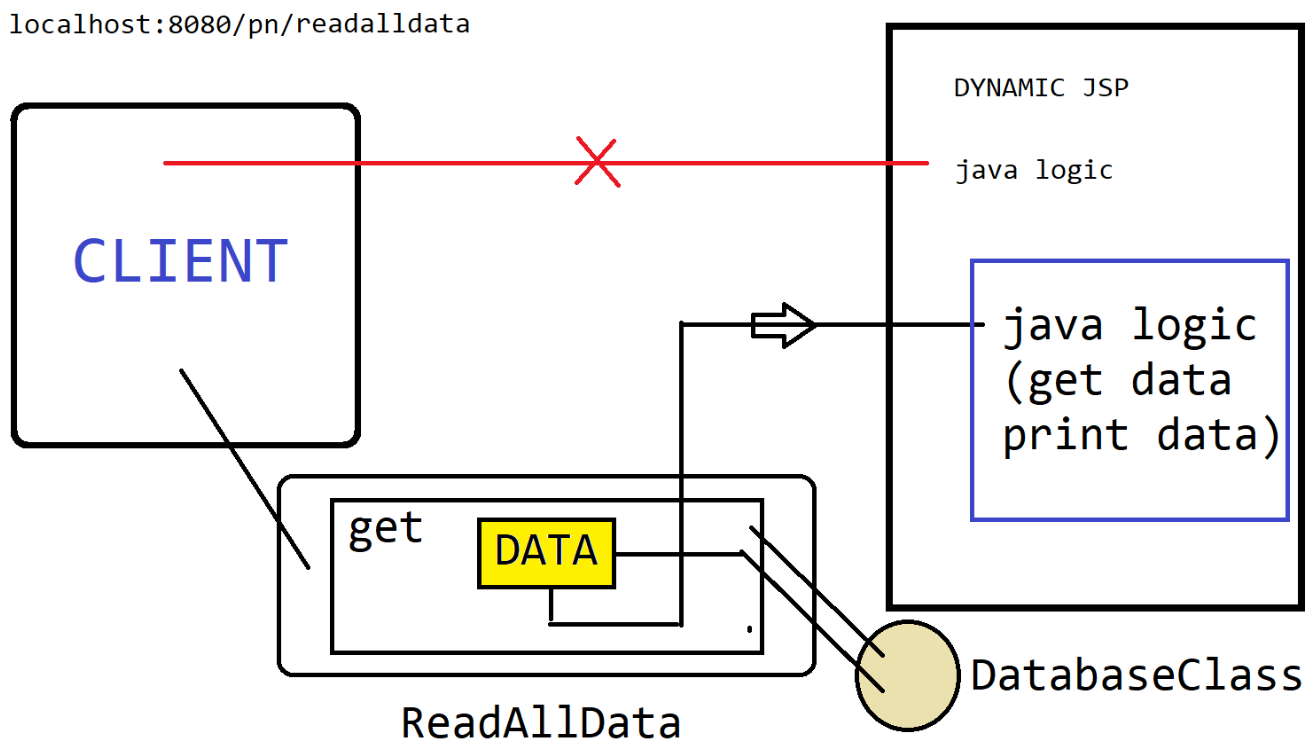
But...

Best Practice: No

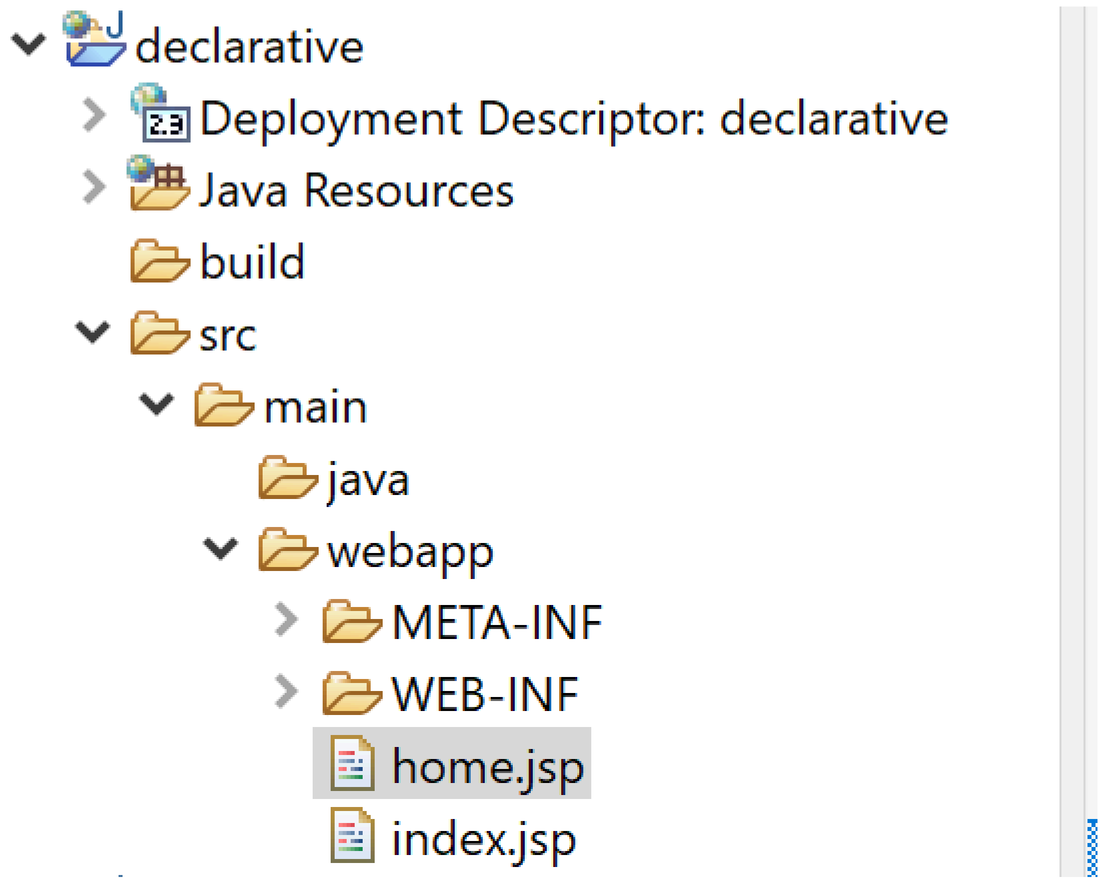
- In MVC architecture, JSP should not be accessed directly. Instead:
- Requests should go to the Controller (Servlet or Spring Controller).
- The controller should process the logic and then forward to the JSP as a view.
- This keeps your application structured, maintainable.

Why avoid direct DYNAMIC JSP access?

- Breaks MVC flow.
- Makes your app harder to maintain and test.
- In Servlet (DYNAMIC WEB PROJECT), JSPs are usually placed in /WEB-INF/ so they are not directly accessible.


localhost:8080/pn/readalldata
CLIENT
DYNAMIC JSP
java logic
java logic (get data print data)
get
DATA
DatabaseClass
ReadAllData

==================================PROGRAM: Declarative Tag======================================



index.jsp

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <form action="home.jsp" method="post">
       <input type="text" name="name" ><br>
       <button>send</button>
   </form>

</body>
</html>
```

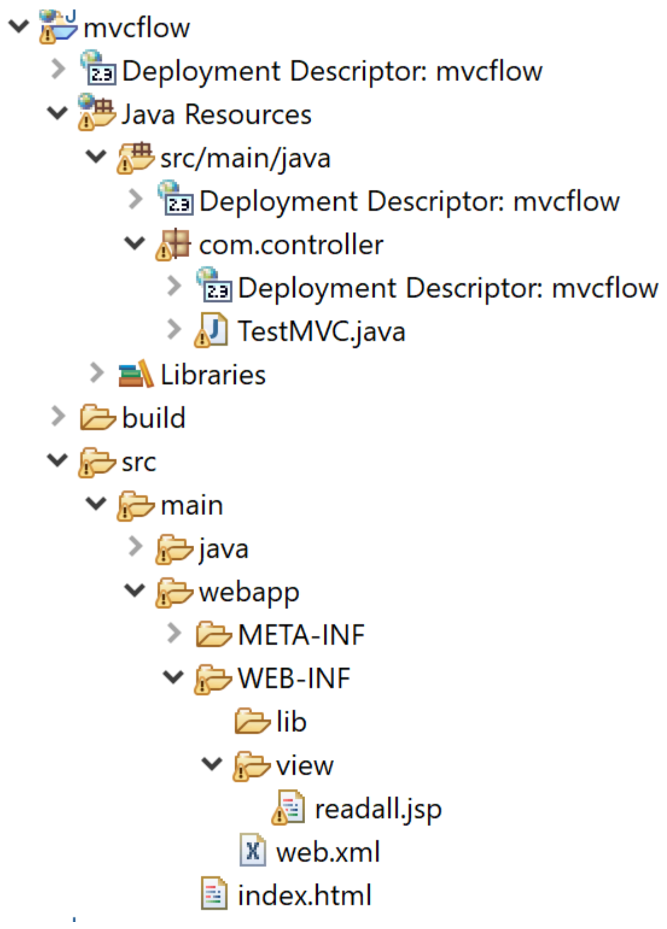
home.jsp

```
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

    <%!
       private String name;
    %>

     <%
        String name= request.getParameter("name");
        this.name=this.name+name;
     %>
     <h3>NAME:<%=this.name%></h3>


</body>
</html>
```

==

======================================PROGRAM: MVC DESIGN PATTERN===================================



```
mvcflow
  Deployment Descriptor: mvcflow
  Java Resources
    src/main/java
      Deployment Descriptor: mvcflow
      com.controller
        Deployment Descriptor: mvcflow
        TestMVC.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
          view
            readall.jsp
          web.xml
        index.html
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <a href="readall">READ ALL DATA</a>

</body>
</html>
```

readall.jsp

```
<%@page import="java.util.List"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
```

```
<title>Insert title here</title>
</head>
<body>

<h3>READ ALL DATA(DYNAMIC)</h3>

<%
     List<String> list = ( List<String>)
session.getAttribute("data");
     if(list!=null){

        for(String data : list){
%>
           <h2>NAME: <%=data %></h2>
<%
        }
     }
%>
</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
      <display-name>mvcflow</display-name>
      <servlet>
            <servlet-name>TestMVC</servlet-name>
            <display-name>TestMVC</display-name>
            <description></description>
            <servlet-class>com.controller.TestMVC</servlet-class>
      </servlet>
      <servlet-mapping>
            <servlet-name>TestMVC</servlet-name>
            <url-pattern>/readall</url-pattern>
      </servlet-mapping>
      <welcome-file-list>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>default.html</welcome-file>
            <welcome-file>default.htm</welcome-file>
            <welcome-file>default.jsp</welcome-file>
      </welcome-file-list>
</web-app>
```

TestMVC.java

```java
package com.controller;
import java.io.IOException;
import java.util.Arrays;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
public class TestMVC extends HttpServlet {

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


		List<String> list = Arrays.asList("emp1","emp2");

		HttpSession session = request.getSession();
		session.setAttribute("data", list);

		RequestDispatcher requestDispatcher = 
request.getRequestDispatcher("WEB-INF/view/readall.jsp");
		requestDispatcher.forward(request, response);

	}
}
```

==================================

==================================

========================

DAY 15

✅ Q: What is Session Management in Servlets?

A:

Session Management refers to the process of tracking and maintaining user state and data across multiple HTTP requests during a user's interaction with a web application.

Since HTTP is stateless, the server doesn't remember the user between two requests. Session management helps the server recognize the same client (user) across multiple requests and store user-specific information like login status, shopping cart, preferences, etc.

✅ Q: Why is HTTP considered stateless?

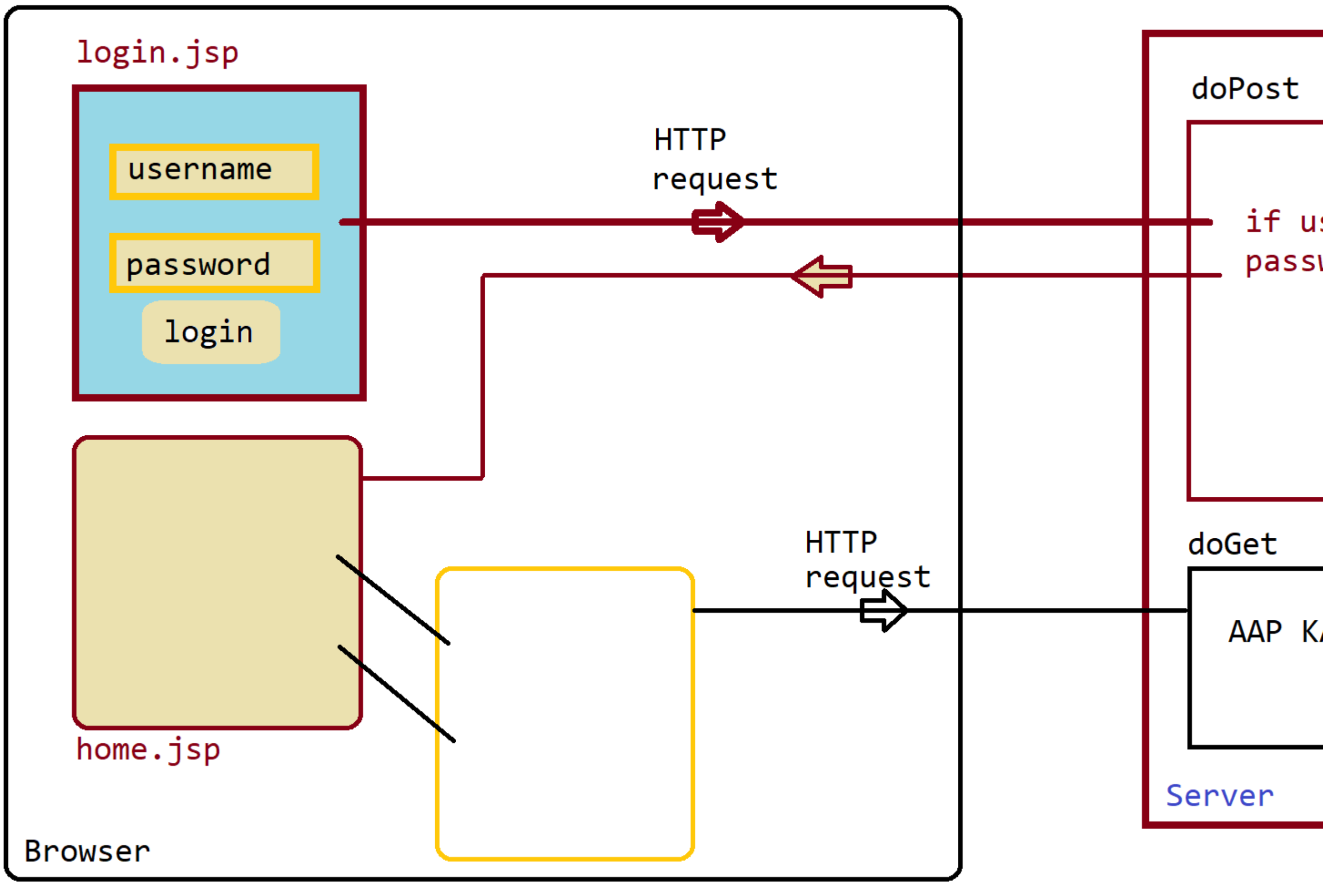
A:

HTTP is considered stateless because each request sent from the client to the server is

independent and has no memory of previous interactions. The server does not retain any information about past requests made by the same client.

What does "stateless" mean?

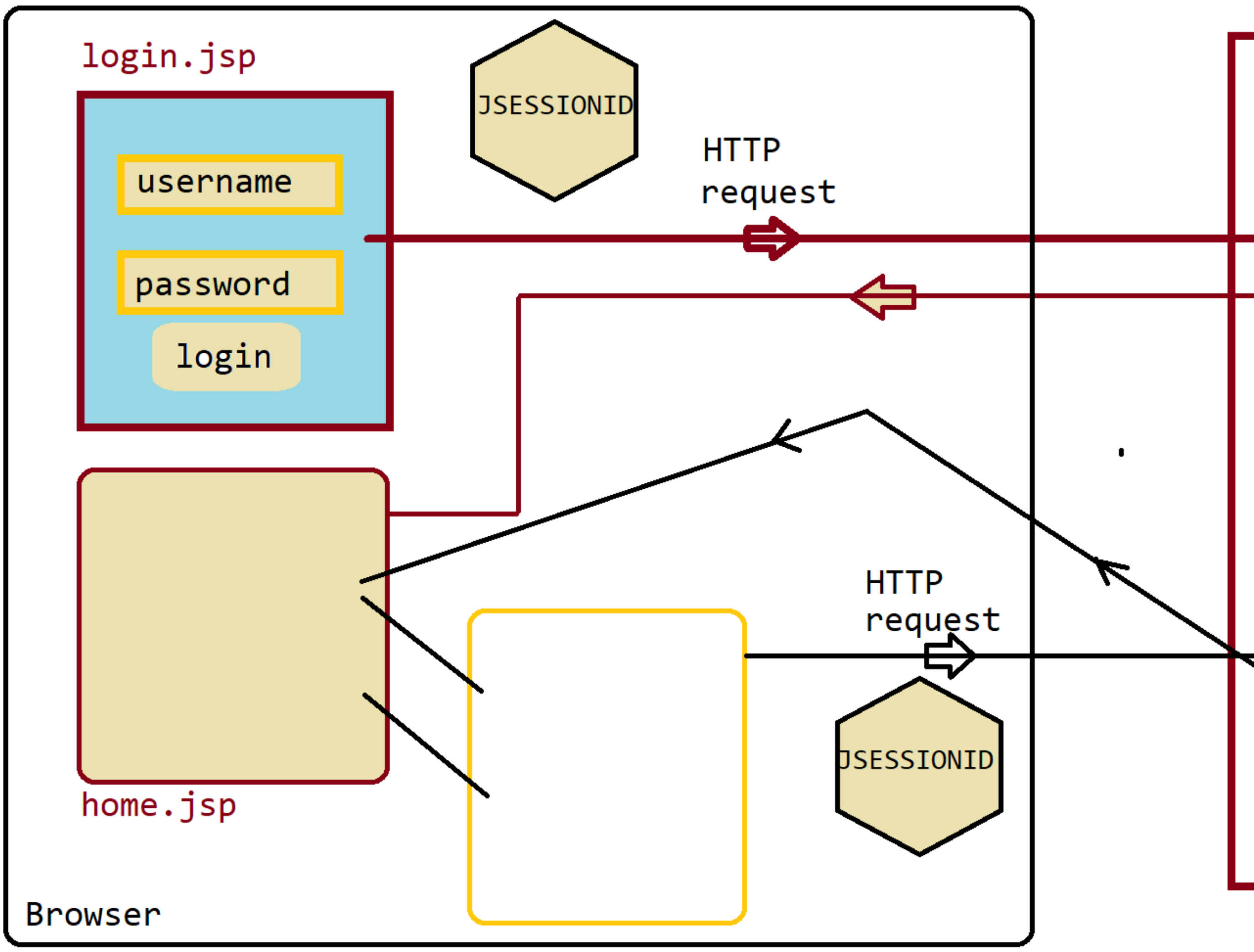
- "Stateless" means no session or memory is maintained between requests.
- Each HTTP request is treated as new and unrelated to any previous request.



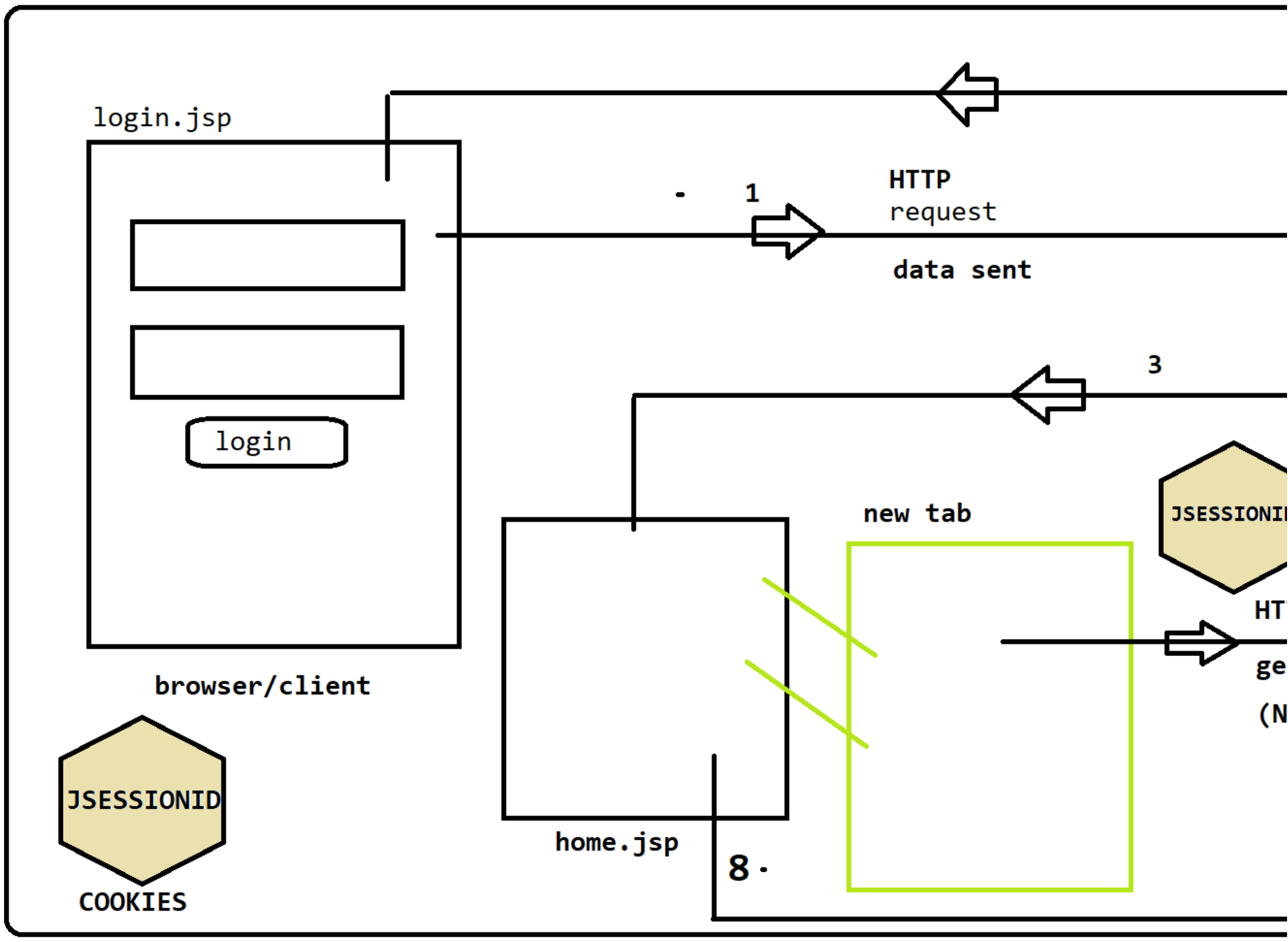
- **Session management in Java can be implemented in several ways However, the recommended and most widely used approach is through HttpSession, as it provides a simple, powerful, and secure way to store and manage user-specific data on the server side(STATEFULL).**

✅ What is HttpSession?

- HttpSession is a Java interface provided by the Servlet API (used in Servlet, JSP, Spring apps) that allows you to store user-specific data on the server between multiple requests.

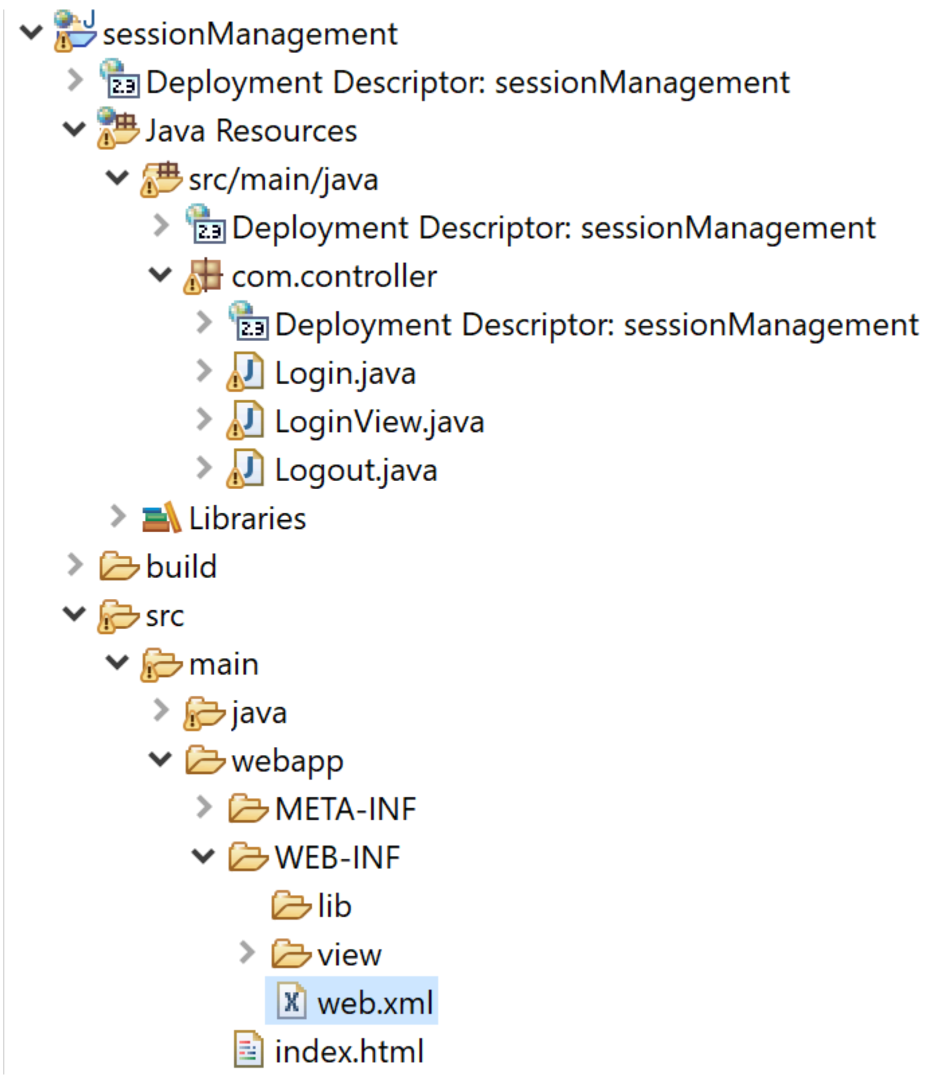


OR



- When the user fills in the username and password on login.jsp and submits the form, a POST request is sent to the server. If the credentials are valid (e.g., username is raju and password is 123456), the server creates a session and stores the username as a session attribute. The server then sends a JSESSIONID to the client as a cookie. This ID uniquely identifies the user’s session. On subsequent requests (like when the user visits home.jsp), the browser sends this JSESSIONID back automatically in the HTTP request. The server uses this ID to fetch the user’s session and retrieve stored data like the username, allowing the server to maintain user-specific state and personalize the response — even though HTTP itself is stateless.

```
==============================PRO
GRAM: SESSION
MANAGEMENT======================
==========
```



```
sessionManagement
  Deployment Descriptor: sessionManagement
  Java Resources
    src/main/java
      Deployment Descriptor: sessionManagement
      com.controller
        Deployment Descriptor: sessionManagement
        Login.java
        LoginView.java
        Logout.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
          view
          web.xml
        index.html
```

index.html

```html
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <a href="login-view">LOGIN HERE</a>

</body>
</html>
```

LoginView.java

```java
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginView extends HttpServlet {

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


request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
      }
}
```

Login.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Login extends HttpServlet {

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

           HttpSession session = request.getSession();
          String data = (String)session.getAttribute("data");

          if(data!=null) {

request.getRequestDispatcher("/WEB-INF/view/home.jsp").forward(reque
st, response);
          }
          else {

request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
          }
      }

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

           String username = request.getParameter("username");
           String password = request.getParameter("password");
           if(username.equals("raju") &&  password.equals("123456"))
{

                HttpSession session = request.getSession();
                session.setAttribute("data", username);


request.getRequestDispatcher("/WEB-INF/view/home.jsp").forward(reque
st, response);
           }else {

request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
           }

     }
}
```

Logout.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Logout extends HttpServlet {

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

           HttpSession session = request.getSession();
           session.removeAttribute("data");


request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);

     }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
     <display-name>sessionManagement</display-name>
```

```
	<servlet>
		<servlet-name>LoginView</servlet-name>
		<display-name>LoginView</display-name>
		<description></description>
		<servlet-class>com.controller.LoginView</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>Login</servlet-name>
		<display-name>Login</display-name>
		<description></description>
		<servlet-class>com.controller.Login</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>Logout</servlet-name>
		<display-name>Logout</display-name>
		<description></description>
		<servlet-class>com.controller.Logout</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>LoginView</servlet-name>
		<url-pattern>/login-view</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>Login</servlet-name>
		<url-pattern>/login</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>Logout</servlet-name>
		<url-pattern>/logout</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.html</welcome-file>
		<welcome-file>index.htm</welcome-file>
		<welcome-file>index.jsp</welcome-file>
		<welcome-file>default.html</welcome-file>
		<welcome-file>default.htm</welcome-file>
		<welcome-file>default.jsp</welcome-file>
	</welcome-file-list>
</web-app>
```

==================================

==================================

========================

DAY 16

Listener

- A Servlet Listener in Java EE (Jakarta EE) is a special component that listens for specific events in a web application lifecycle, such as when the application starts or stops, when a session is created or destroyed, or when a request is initialized or completed.
- Servlet listeners are used to perform tasks automatically at certain points in the application's life cycle, such as logging, initializing resources, cleaning up, tracking user sessions, etc.
- We have many types of Servlet listeners, but in this topic, we'll focus on the ServletContextListener.
- It is mainly used for executing code when the web application starts or stops.
- For example: it has two methods

1. Initializing resources like table creation, logging or configuration files when the application starts.
2. Cleaning up resources when the application shuts down.

Filter

- A Servlet Filter is a powerful component in Java web applications that allows you to intercept requests and responses before they reach a servlet or JSP, or before the response is sent back to the client.
- A Filter is part of the Servlet specification and is used to:
  1. Pre-process requests (e.g., authentication, request modification)
  2. Post-process responses (e.g., modifying response headers, compression)

Where Does a Filter Sit?

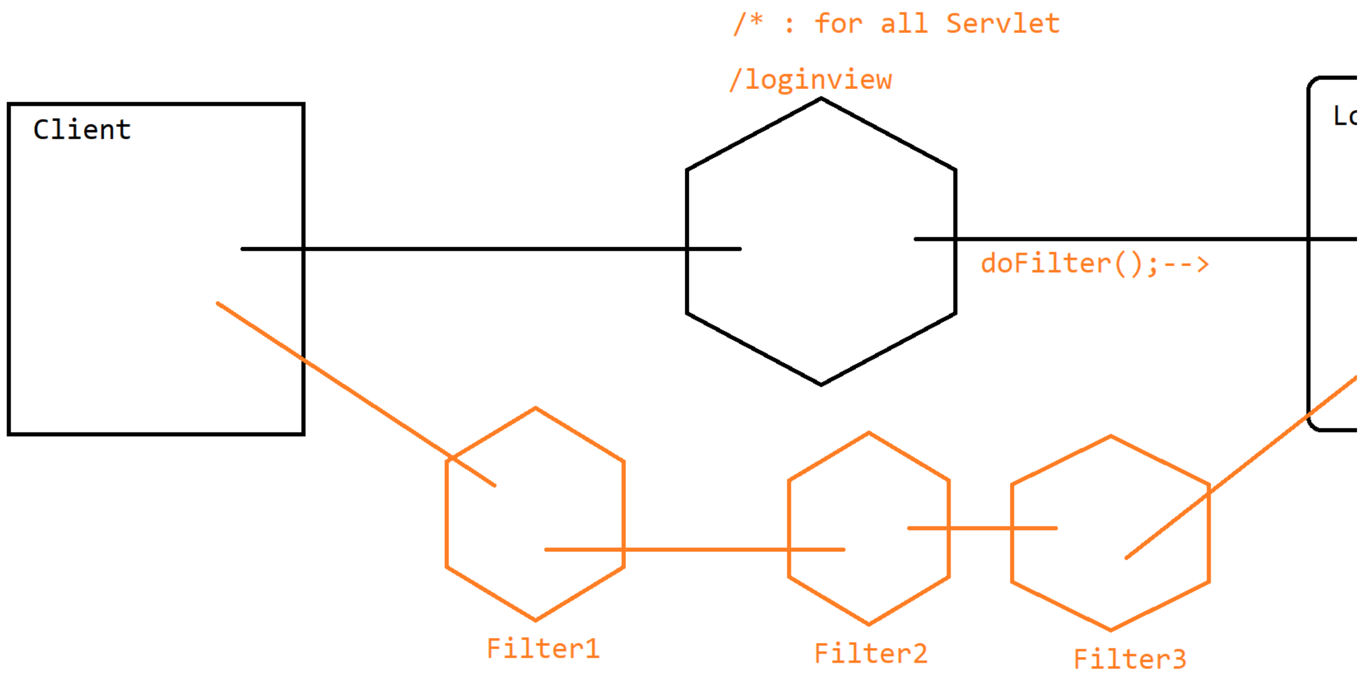
- Think of a filter as a middleware that sits between the client and the servlet
  Client --> Filter(PRE) --> Servlet --> Response --> Filter(POST) --> Client

- Pre-processing code in a filter is written before the line chain.doFilter(request, response); inside the doFilter method. This means you put your logic to handle or inspect the request before it gets passed along to the next filter or servlet. Conversely, post-processing code is written after the chain.doFilter (request, response); call, which allows you to work with the response after the servlet and other filters have finished processing but before the response goes back to the client.

What does this line do?

- chain.doFilter(request, response); means:

- **Pass this request and response to the next filter in the chain, or if there are no more filters, pass it to the target servlet or resource.**



What are the purposes of the init and destroy methods in a servlet filter, and when are they called?

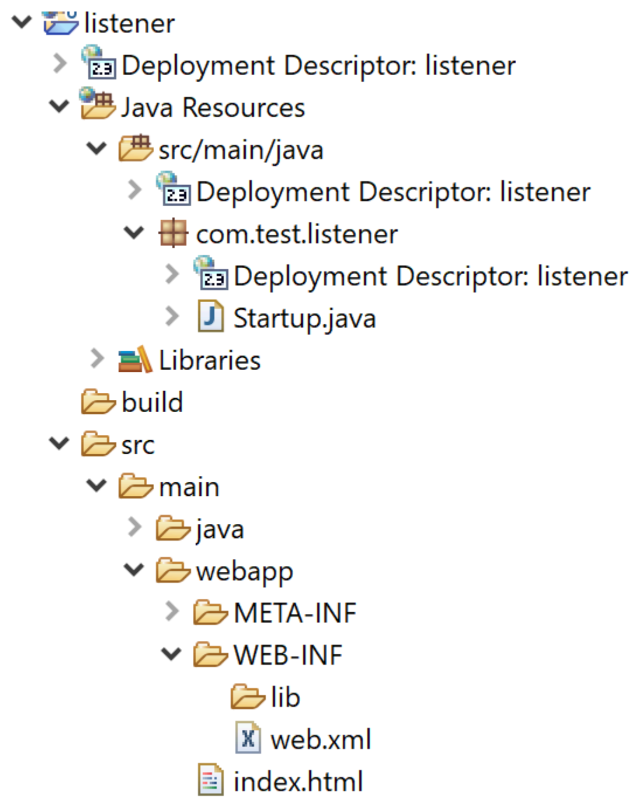
- **The init method in a filter is called once by the servlet container when the filter is first** created and is used to perform any one-time setup or initialization tasks, such as reading configuration parameters. The destroy method is called when the filter is taken out of service, allowing you to release resources or perform cleanup activities before the filter is removed. Together, they manage the filter’s lifecycle beyond just processing requests.

What does filter mapping mean in a Java web application?

- Filter mapping defines which requests a filter should apply to by specifying URL patterns or servlet names. It tells the servlet container when and for which URLs or servlets the filter’s code should run before reaching the target resource. This configuration can be done in web.xml or via annotations.

==================================

PROGRAM: Listener==



index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

WELCOME

</body>
</html>
```

Startup.java

```
package com.test.listener;
```

```java
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class Startup implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce)  {
       System.out.println("TABLE DELETED");
    }

    public void contextInitialized(ServletContextEvent sce)  {
        System.out.println("TABLE CREATED");
    }
}
```

web.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
      <display-name>listener</display-name>
      <listener>

<listener-class>com.test.listener.Startup</listener-class>
      </listener>
      <welcome-file-list>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>default.html</welcome-file>
            <welcome-file>default.htm</welcome-file>
            <welcome-file>default.jsp</welcome-file>
      </welcome-file-list>
</web-app>
```

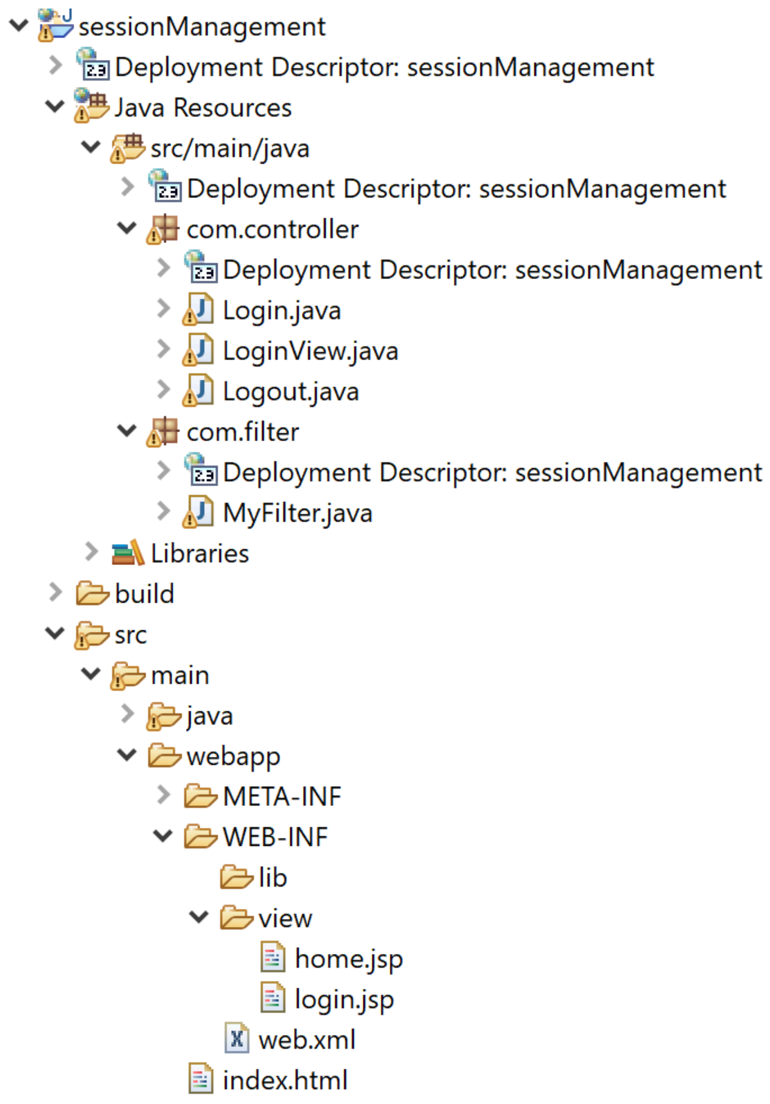
==================================

==================================

=======================

===================================

PROGRAM: Filter===



```
sessionManagement
  Deployment Descriptor: sessionManagement
  Java Resources
    src/main/java
      Deployment Descriptor: sessionManagement
      com.controller
        Deployment Descriptor: sessionManagement
        Login.java
        LoginView.java
        Logout.java
      com.filter
        Deployment Descriptor: sessionManagement
        MyFilter.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
          view
            home.jsp
            login.jsp
          web.xml
        index.html
```

index.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
```

```
   <a href="login-view">LOGIN HERE</a>

</body>
</html>
```

home.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
THIS IS HOME PAGE

<a href="logout" >LOGOUT</a>

</body>
</html>
```

login.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <form action="login" method="post">
      <input type="text" name="username" placeholder="enter
username" ><br>
      <input type="password" name="password" placeholder="enter
password" ><br>
      <button>login</button>
   </form>

</body>
</html>
```

Login.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Login extends HttpServlet {

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

           HttpSession session = request.getSession();
          String data = (String)session.getAttribute("data");

          if(data!=null) {

request.getRequestDispatcher("/WEB-INF/view/home.jsp").forward(reque
st, response);
          }
          else {

request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
          }
      }

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

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

           if(username.equals("raju") &&  password.equals("123456"))
{

                   HttpSession session = request.getSession();
                   session.setAttribute("data", username);


request.getRequestDispatcher("/WEB-INF/view/home.jsp").forward(reque
st, response);
           }else {

request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
```

```
          }

     }
}
```

LoginView.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginView extends HttpServlet {

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

          System.out.println("LOGIN VIEW");

request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);
     }
}
```

Logout.java

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class Logout extends HttpServlet {

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

          HttpSession session = request.getSession();
          session.removeAttribute("data");
request.getRequestDispatcher("/WEB-INF/view/login.jsp").forward(requ
est, response);

      }
}
```

MyFilter.java

```
package com.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class MyFilter extends HttpFilter implements Filter {

      public void destroy() {
            System.out.println("FILTER DESTROYED...");
      }

      public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {

            HttpServletRequest req=(HttpServletRequest) request;
            HttpServletResponse res=(HttpServletResponse) response;

            String header = req.getHeader("User-Agent");
            System.out.println(header);

            if(header.contains("Chrome")) {
                  res.getWriter().print("INVALID BROWSER");
                  return;
            }

            long t1 = System.nanoTime();

            //PRE
            chain.doFilter(request, response); //forward-> EX.SERVLET
            //POST

            long t2 = System.nanoTime();
```

```
		System.out.println("TIME TAKEN:"+ (t2-t1)+"ns");

	}

	public void init(FilterConfig fConfig) throws ServletException
{
		System.out.println("FILTER INITIALIZED...");
	}

}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app id="WebApp_ID">
	<display-name>sessionManagement</display-name>
	<filter>
		<filter-name>MyFilter</filter-name>
		<display-name>MyFilter</display-name>
		<description></description>
		<filter-class>com.filter.MyFilter</filter-class>
	</filter>
	<filter-mapping>
		<filter-name>MyFilter</filter-name>
		<url-pattern>/*</url-pattern>
	</filter-mapping>
	<servlet>
		<servlet-name>LoginView</servlet-name>
		<display-name>LoginView</display-name>
		<description></description>
		<servlet-class>com.controller.LoginView</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>Login</servlet-name>
		<display-name>Login</display-name>
		<description></description>
		<servlet-class>com.controller.Login</servlet-class>
	</servlet>
	<servlet>
		<servlet-name>Logout</servlet-name>
		<display-name>Logout</display-name>
		<description></description>
		<servlet-class>com.controller.Logout</servlet-class>
	</servlet>
	<servlet-mapping>
		<servlet-name>LoginView</servlet-name>
		<url-pattern>/login-view</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>Login</servlet-name>
		<url-pattern>/login</url-pattern>
	</servlet-mapping>
	<servlet-mapping>
		<servlet-name>Logout</servlet-name>
		<url-pattern>/logout</url-pattern>
	</servlet-mapping>
	<welcome-file-list>
		<welcome-file>index.html</welcome-file>
		<welcome-file>index.htm</welcome-file>
		<welcome-file>index.jsp</welcome-file>
		<welcome-file>default.html</welcome-file>
		<welcome-file>default.htm</welcome-file>
		<welcome-file>default.jsp</welcome-file>
	</welcome-file-list>
</web-app>
```

==================================

==================================

=======================

DAY 17

JSP DIRECTIVE TAGS

- **In this session, we will dive into JSP (JavaServer Pages) and learn about Directive Tags, one of the core components of JSP.**

- **JSP directive tags are special instructions that provide global information about the entire JSP page. These instructions affect how the JSP page is translated into a servlet.**

There are 3 types of directive tags in JSP:

1. **page directive – for setting page-level configurations**

2. include directive – for including static content during translation
3. taglib directive – for using custom or JSTL tag libraries

Directive Tag Syntax:

- All directive tags in JSP follow this basic syntax: `<%@ directiveName attribute="value" %>`

JSP Directive Tags Work Based on Their Attributes:

- Each directive tag works on the basis of the attributes provided to it. These attributes define the behavior, configuration, and features of the JSP page.

PAGE DIRECTIVE

- The page directive provides global settings for the JSP page. It controls the overall behavior and properties of the JSP during its translation and execution.
- Here are the most common attributes used with page directive tag:

❖ contentType – Sets the MIME type and character encoding for the response

❖ pageEncoding – Specifies the character encoding used to read the JSP source file itself (e.g., UTF-8).

❖ import – Allows importing Java classes or packages into the JSP

❖ session – Enables or disables session tracking on the page

❖ isELIgnored – Specifies whether to ignore Expression Language (EL)

❖ errorPage – Defines the error handler page to redirect on exceptions

❖ isErrorPage – Declares whether this page can handle exceptions

❖ language – Specifies the scripting language used in the JSP (default is "java").

❖ buffer – Sets the output buffer size

- ❖ autoFlush – Controls whether the buffer is automatically flushed when full (true or false).

contentType Attribute

- ❖ It tells the browser what type of content the JSP page will return and how to interpret the characters (i.e., the character encoding).
- ❖ the default value of the contentType attribute in JSP is text/html; charset=ISO-8859-1

What does text/html mean in contentType?

- It means the content returned is HTML, so the browser should render it as a web page.
- text/html is a MIME type, where MIME stands for Multipurpose Internet Mail Extensions.

What does charset=ISO-8859-1 mean?

- It specifies the character encoding.
- ISO-8859-1 (also called Latin-1) supports most Western European languages (English, French, German, etc.).

Why is the charset important?

- Because the server sends bytes, not characters.
- The browser needs to know how to decode those bytes into characters.

What is a better alternative to ISO-8859-1 today?

- UTF-8 is preferred because:
- It supports all characters and languages
- It’s the most widely used encoding today

- Recommended usage: <%@ page contentType="text/html; charset=UTF-8" %>

NOTE:

- In JSP, the default content type is "text/html; charset=ISO-8859-1", which tells the browser to treat the output as an HTML web page with Latin-1 encoding. However, we can change this default content type using the contentType attribute in the <%@ page %> directive. For example, if we want the page to return plain text instead of HTML, we can set contentType="text/plain". Similarly, to send JSON data (useful in AJAX or API responses), we can use contentType="application/json". By specifying the appropriate MIME type and character encoding (like UTF-8), we ensure the browser processes and displays the response correctly based on the type of content being returned.

pageEncoding Attribute

- The pageEncoding attribute specifies the character encoding used to read the JSP source file itself (the .jsp file).
- It tells the JSP engine how to interpret the bytes in the JSP file as characters when compiling it.

Why is pageEncoding important?

- JSP source files may contain non-ASCII characters (e.g.Unicode symbols).
- To correctly process and display these characters, the JSP engine needs to know the encoding of the JSP file.

- **If the encoding is not specified or mismatched, characters may appear garbled or cause errors.**

How to use it?

<%@ page pageEncoding="UTF-8" %>

import Attribute

- **The import attribute is used in the <%@ page %> directive to import Java classes or entire packages into a JSP page. This lets you use those classes directly in your JSP scriptlets, expressions, or declarations without writing the full package name every time.**

```
<%@ page import="java.util.Date" %>
<%
   Date currentDate = new Date();
   out.println("Current Date is: " +
currentDate);
%>
```

session Attribute

- The session attribute in the <%@ page %> directive controls whether the JSP page participates in HTTP session tracking.

What does the session attribute do?

- session="true" (default):
- The JSP page has access to the implicit session object of type HttpSession.
- You can store or retrieve user-specific data like:
    1. session.setAttribute("username", "Saif");
    2. String name = (String) session.getAttribute("username");

- session="false":

- The JSP page will not create or use a session object.
- You cannot use the session implicit object in this page.

isELIgnored Attribute

- The isELIgnored attribute controls whether the JSP Expression Language (EL) is enabled or ignored on that JSP page.

What does isELIgnored do?

- isELIgnored="false" (default): EL expressions like ${...} are evaluated and the result is displayed.
- isELIgnored="true": EL expressions are ignored — treated as plain text, not evaluated.

errorPage & isErrorPage Attribute

errorPage

- Purpose: Specifies the JSP page that should handle errors or exceptions thrown on the current JSP page.

How it works:

- If an exception occurs on this JSP page, the request is forwarded to the page defined in errorPage.

Usage:

- <%@ page errorPage="errorHandler.jsp" %>
- Here, if an error occurs, the server redirects the user to errorHandler.jsp.

isErrorPage

- Purpose: Declares whether the current JSP page is an error-handling page.

How it works:

- When isErrorPage="true" is set, the JSP page automatically gets access to an implicit exception object named exception which contains the thrown exception.

Usage:

- <%@ page isErrorPage="true" %>
- This page can then display details about the exception, log it, or handle it gracefully.

language Attribute

- The language attribute in the <%@ page %> directive specifies the scripting language used in the JSP page.
- It tells the JSP engine what language the scriptlets and expressions are written in.
- The default and most commonly used value is: language="java"
- This means the JSP uses Java as the scripting language.

Can it be something else?

- In early JSP versions, other scripting languages were possible (like JavaScript), but modern JSPs almost always use Java.
- Most JSP containers only support Java for scripting.

Usage example:

```
<%@ page language="java" %>
```

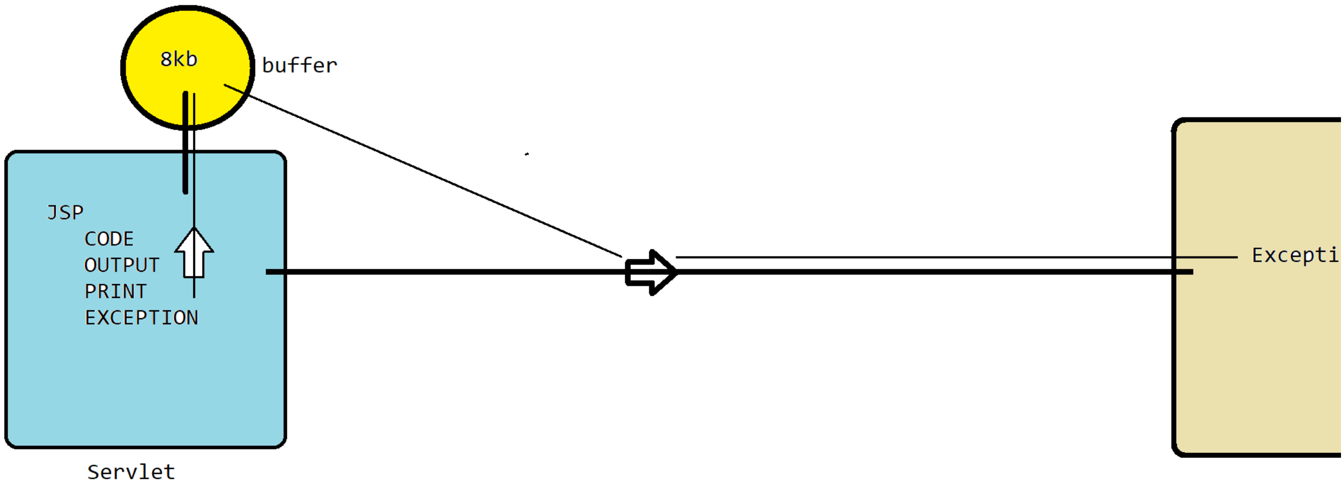
Why specify it?

- Usually, you don’t need to specify it because Java is the default.
- But it’s good practice to include it for clarity.

buffer Attribute

- The buffer attribute in the <%@ page %> directive controls how much output

(HTML/text) the JSP page stores temporarily in memory before sending it to the client (browser).



Why use a buffer?

- When your JSP generates output (like HTML), the server doesn't always send it to the browser immediately. Instead:
- It collects the output in a buffer (like a holding area).
- Once the buffer is full, or the page finishes executing, it flushes the content and sends it to the browser.

This allows:

- Efficient communication with the client
- Server to handle errors or redirect before any part of the response is sent

Syntax

```
<%@ page buffer="8kb" %>
```

You can set values like:

- "8kb" (default)
- "16kb", "32kb", etc.
- "none" – disables buffering completely

Maximum Buffer Size in Tomcat:

- There is no fixed maximum limit enforced by Tomcat like “you can’t go beyond X”. But in practice:
- You can set large values like 32KB, 64KB, or even 256KB
- Very large values (e.g., 10MB or more) might cause memory or performance issues

- The buffer is stored per request, so hundreds of large buffers can consume memory fast

INCLUDE DIRECTIVE

- The include directive is used to statically include the content of one JSP file into another at compile time.

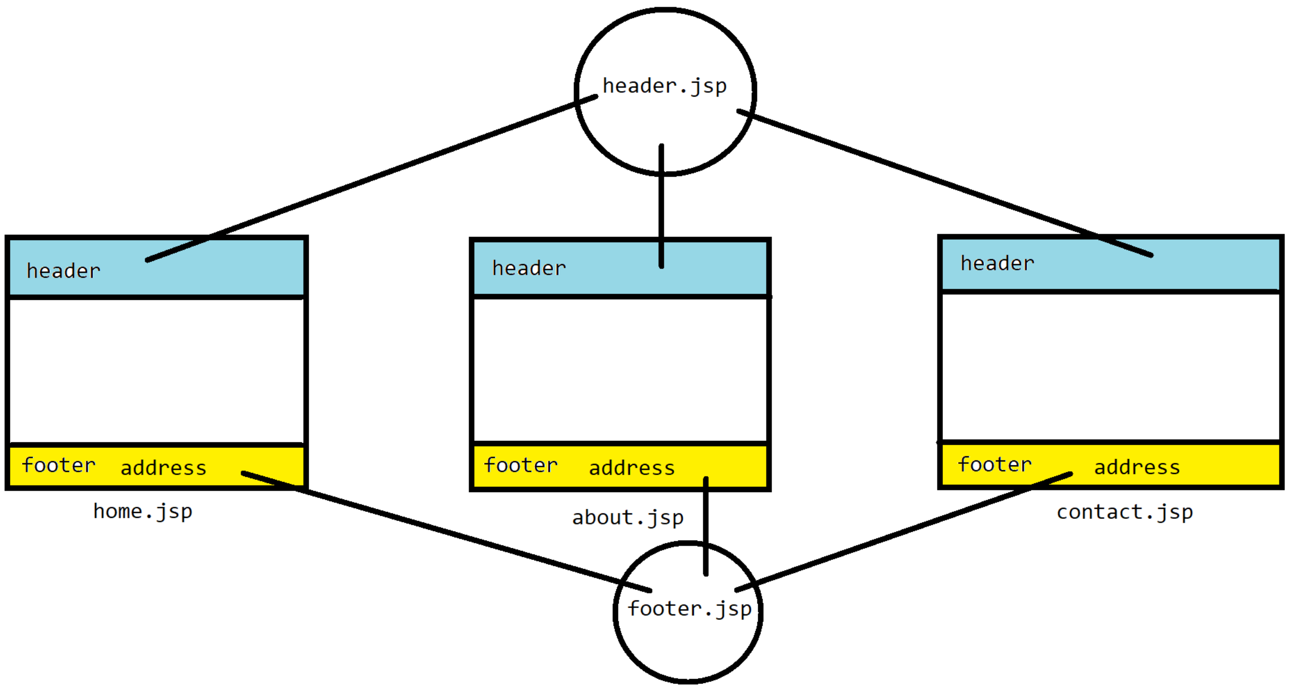
Syntax:

```
<%@ include file="header.jsp" %>
```

file is a required attribute that specifies the relative path of the file to be included.

How It Works:

- When the JSP page is compiled, the contents of the included file are physically copied into the main JSP.
- This is done before the JSP is converted to a servlet.
- It’s like copy-pasting the contents of the included file.



JSP IMPLICIT OBJECTS

- In JSP, implicit objects are predefined variables automatically available to you without declaring or initializing them. They make it easier to interact with the servlet environment — request, response, session, application, etc.

The 9 Standard JSP Implicit Objects:

1.request

- Represents the HttpServletRequest object for the current request. Used to get request parameters, headers, etc.

2.response

- Represents the HttpServletResponse object to send data back to the client.

3.out

- A JspWriter object used to send output to the client (similar to PrintWriter).

4.session

- Represents the HttpSession associated with the request. Used to track user data across requests.

5.application

- Represents the ServletContext shared by all JSPs and servlets in the web app.

6.config

- The ServletConfig object for the JSP’s servlet. Contains initialization info.

7.pageContext

- It helps you access different areas (namespaces) where data is stored in JSP, like page, request, session, and application.
- JSP has different namespaces (or scopes) where variables or attributes live separately:
    1. Page scope: Attributes visible only on the current JSP page.
    2. Request scope: Attributes visible throughout the current HTTP request.
    3. Session scope: Attributes visible throughout the user's session.
    4. Application scope: Attributes shared across the entire web app.

```
<%
  // Store attribute in different scopes
  pageContext.setAttribute("pageMsg", "Hello
Page", PageContext.PAGE_SCOPE);
  pageContext.setAttribute("requestMsg",
"Hello Request", PageContext.REQUEST_SCOPE);
  pageContext.setAttribute("sessionMsg",
"Hello Session", PageContext.SESSION_SCOPE);
  pageContext.setAttribute("appMsg", "Hello
Application", PageContext.APPLICATION_SCOPE);

  // Retrieve attributes
  out.println("Page Scope: " +
pageContext.getAttribute("pageMsg",
PageContext.PAGE_SCOPE) + "<br>");
  out.println("Request Scope: " +
pageContext.getAttribute("requestMsg",
PageContext.REQUEST_SCOPE) + "<br>");
  out.println("Session Scope: " +
pageContext.getAttribute("sessionMsg",
PageContext.SESSION_SCOPE) + "<br>");
  out.println("Application Scope: " +
pageContext.getAttribute("appMsg",
PageContext.APPLICATION_SCOPE) + "<br>");
%>
```

Why do we need pageContext when we already have request, response, session, and application?

Centralized Access Point:

- pageContext acts as a single “hub” or wrapper that gives you easy access to all these objects plus other useful stuff — so you don’t have to deal with each one separately all the time.

Other useful stuff:

Forward Requests (like RequestDispatcher)

- pageContext.forward("otherPage.jsp");

Include Other Resources

- pageContext.include("header.jsp");

Access Servlet Config or Servlet Context

- ServletConfig config = pageContext.getServletConfig();
- ServletContext ctx = pageContext.getServletContext();

You can say that pageContext is a central object that gives access to all key components of a JSP page.

8.page

- **Refers to the instance of the generated servlet for this JSP page (rarely used).**
  Example: Using page

```
<%
   out.println("This is the page
object: " + page.toString());
%>
Output:
This is the page object:
org.apache.jsp.example_jsp@15db9742
```

9.exception

- **Used only in JSP error pages, represents the uncaught exception causing the error page to be invoked.**

DAY 18

JSTL

What is JSTL?

- JSTL stands for JavaServer Page Standard Tag Library.
- It is a collection of custom JSP tags that provide common functionalities such as loops, conditionals, formatting, and database access.
- JSTL helps eliminate Java code (scriptlets) from JSP pages, making them cleaner, easier to read, and maintain. This allows front-end developers to work with JSP more comfortably, without needing deep Java knowledge.

- **Makes JSP pages front-end friendly by replacing Java logic with simple, tag-based syntax.**

JSTL Libraries Overview

- **JSTL provides several tag libraries, each serving a specific purpose. These libraries contain a wide range of ready-to-use tags that simplify common tasks in JSP.**
- **In this session, we'll focus on the three most commonly used JSTL libraries:**

  ▯ Core Library (c) – For flow control, iteration, and variable handling
  ▯ Function Library (fn) – For string operations like length, replace, split, case conversion, etc.
  ▯ SQL Library (sql) – For executing database queries directly from JSP (used mainly for quick demos or prototypes)

EL (Expression Language) in JSP:

- EL stands for Expression Language — it's a simple scripting language used in JSP to access data stored in JavaBeans, request parameters, session attributes, etc., without writing Java code.

Purpose of EL:

- To make JSP pages cleaner and more readable by removing the need for Java scriptlets (<% %>).

Example Use Case:

Without EL (using scriptlets):

```
<%
String name =
request.getParameter("name");
out.println(name);
%>
```

With EL:

```
${param.name}
```

List of Important JSTL Core Tags (c)

- <c:set> Sets a variable in a specified scope
- <c:if> Executes body if the test condition is true
- <c:choose> Acts like a switch-case block
- <c:when> Used within <c:choose> as a condition
- <c:otherwise> Default case inside <c:choose>
- <c:forEach> Iterates over a collection, array, or range
- <c:out> Displays a value with optional default (safe printing)

List of Important JSTL Function Tags:

- **fn:contains(str, substr)**
  Checks if str contains substr
- **fn:containsIgnoreCase(str, substr)**
  Case-insensitive version of contains()
- **fn:startsWith(str, prefix)**
  Checks if str starts with prefix
- **fn:endsWith(str, suffix)**
  Checks if str ends with suffix
- **fn:indexOf(str, substr)**
  Returns the index of substr in str (or -1)
- **fn:length(obj)**
  Returns the length of a string, array, or collection
- **fn:toLowerCase(str)**
  Converts string to lowercase
- **fn:toUpperCase(str)**
  Converts string to uppercase

- fn:trim(str)
  Removes leading and trailing spaces from a string

List of Important JSTL SQL Tags:

- <sql:setDataSource> Sets up the database connection (DataSource)
- <sql:query>
  Executes a SELECT SQL query
- <sql:update>
  Executes INSERT, UPDATE, or DELETE queries
- <sql:param> Sets parameter values in a query (used inside <sql:query> or <sql:update>)

DAY 19

In this section, we’ll understand the changes introduced in Servlet 3.0 related to configuration. Earlier, all servlet-related settings had to be done in the web.xml file. But starting from Servlet 3.0, a more flexible and developer-friendly way were introduced using annotations. So, is web.xml gone completely? Let’s explore this in a question-and-answer format for better clarity.

Q: What changed in Servlet 3.0 regarding configuration?

A: Servlet 3.0 introduced annotations (like @WebServlet, @WebFilter, @WebListener) which allow configuring servlets, filters, and listeners without using web.xml.

Q: Can I still use web.xml in Servlet 3.0 or later?

A: Yes, you can still use web.xml. It works as before and is useful in certain advanced or deployment-time configurations.

Q: What are the benefits of using annotations over web.xml?

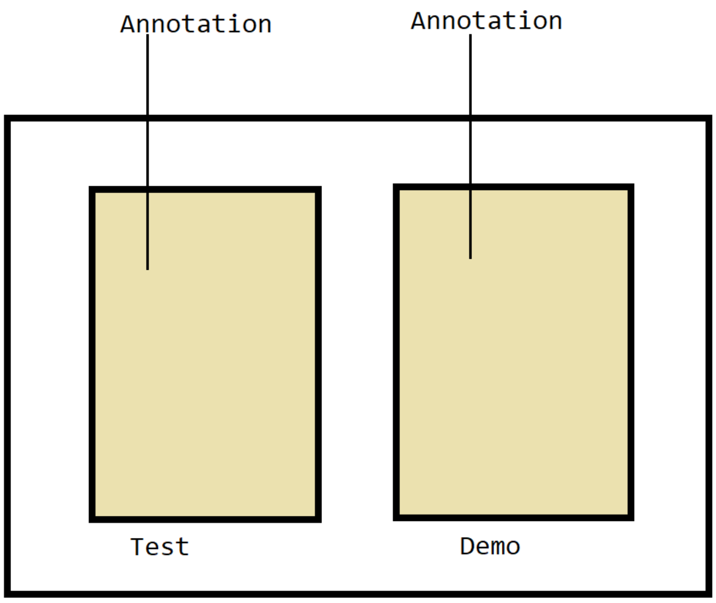
A:

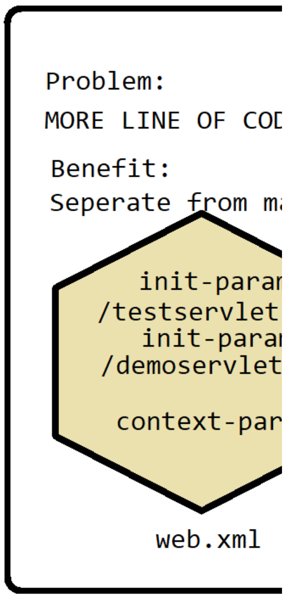
- Less boilerplate.
- Easier and faster to configure.

Q: When should I prefer web.xml instead of annotations?

A:

- When you need to configure without touching source code.
- When deploying to older servers.(LEGACY)





Note:

- **In Java, if a class implements the Serializable interface, its objects can be saved to a file or sent over a network — this process is called serialization.**
- **When Java saves and loads an object, it checks whether the class structure is the same. It does this using a number called serialVersionUID.**

Why Is This Important?

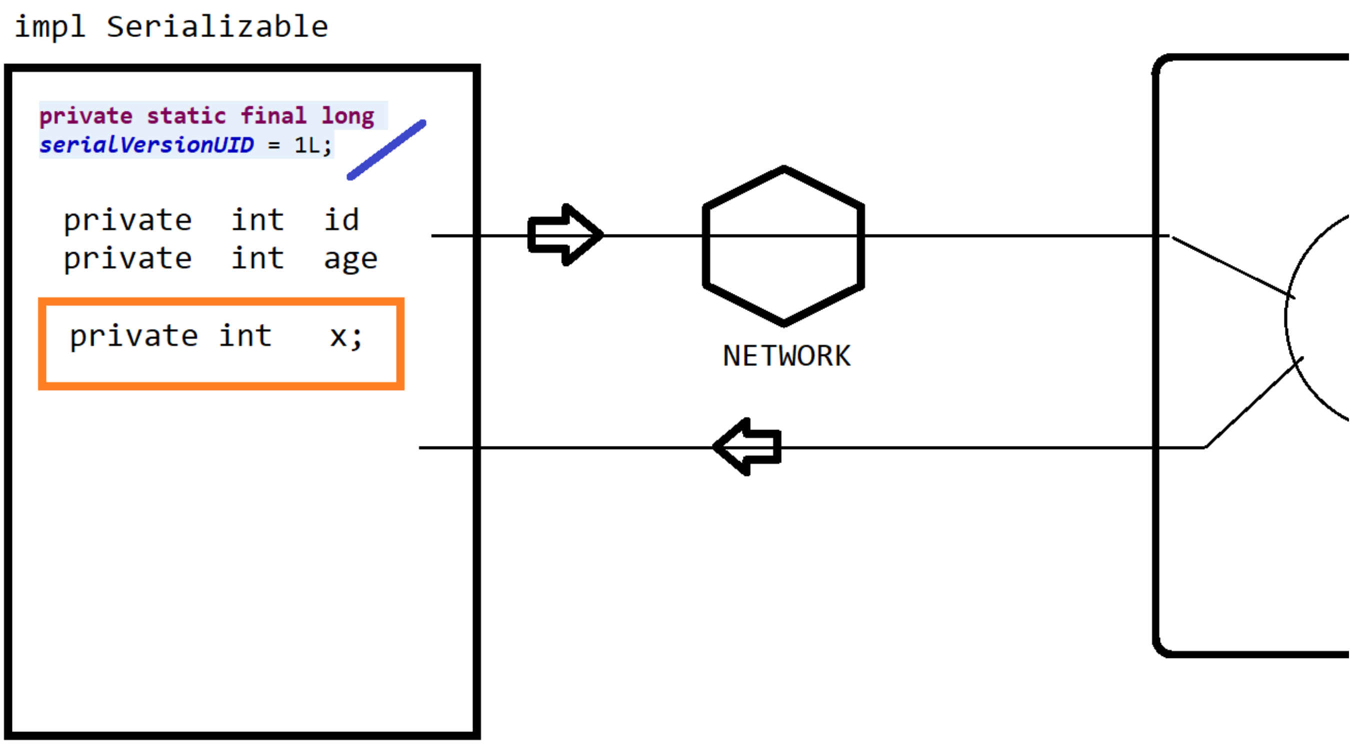
- Let’s say you saved a Student object today.
- After 2 months, you updated the Student class and added a new variable.
- If you try to load the old object — and your class has no serialVersionUID — Java sees a version mismatch and throws an error.

But if you defined:

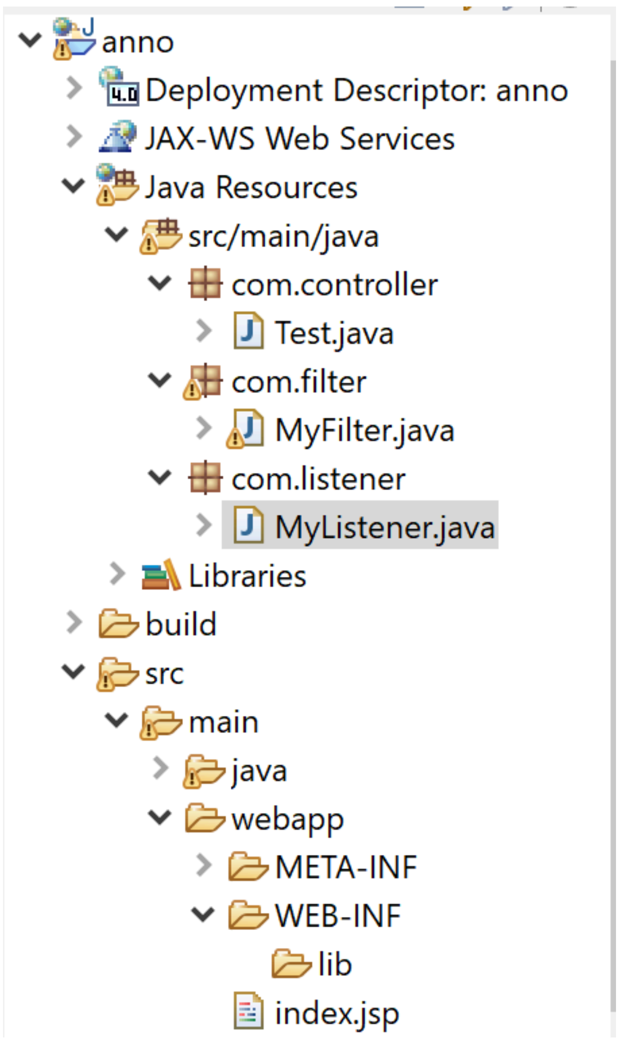
- private static final long serialVersionUID = 1L;
- Then Java says: “Version is same, even if class changed a little — go ahead and load it!”

Real-World Use Case:

- In large apps or banking systems, data is often stored in serialized form.
- To ensure future updates don’t break loading of old data, developers manually set serialVersionUID to avoid issues.



====================================PROGRAM: Annotation Approach====================================



```
anno
  Deployment Descriptor: anno
  JAX-WS Web Services
  Java Resources
    src/main/java
      com.controller
        Test.java
      com.filter
        MyFilter.java
      com.listener
        MyListener.java
    Libraries
  build
  src
    main
      java
      webapp
        META-INF
        WEB-INF
          lib
        index.jsp
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

<a href="testservlet" >1.TEST SERVLET</a><br>
<a href="test" >2.TEST SERVLET</a><br>
<a href="servlettest" >3.TEST SERVLET</a><br>

</body>
</html>
```

```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(
           name = "Test",
           urlPatterns = {"/testservlet","/test","/servlettest"},
           initParams = {
                      @WebInitParam(name="url1" , value =
"jdbc:xyz1"),
                      @WebInitParam(name="url2" , value =
"jdbc:xyz2")
           }

           )
public class Test extends HttpServlet {
      private static final long serialVersionUID = 1L;

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

           ServletConfig servletConfig = getServletConfig();
           ServletContext servletContext = getServletContext();

           response.getWriter().print("SERVLET CALLED: "
               +servletConfig.getInitParameter("url1")
               +" : "+servletContext.getInitParameter("admin"));
      }
}
```

```
package com.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpFilter;
@WebFilter(urlPatterns = {"/*"})
public class MyFilter extends HttpFilter implements Filter {

	public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain) throws IOException, ServletException {
		System.out.println("PRE");
		chain.doFilter(request, response);
		System.out.println("POST");
	}
}
```

```
package com.listener;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
@WebListener
public class MyListener implements ServletContextListener {


    public void contextDestroyed(ServletContextEvent sce)  {
      System.out.println("ON CLOSE");
    }

    public void contextInitialized(ServletContextEvent sce)  {

      System.out.println("ON STARTUP");
      ServletContext servletContext = sce.getServletContext();
      servletContext.setInitParameter("admin", "admin123");

    }
}
```

==================================

==================================

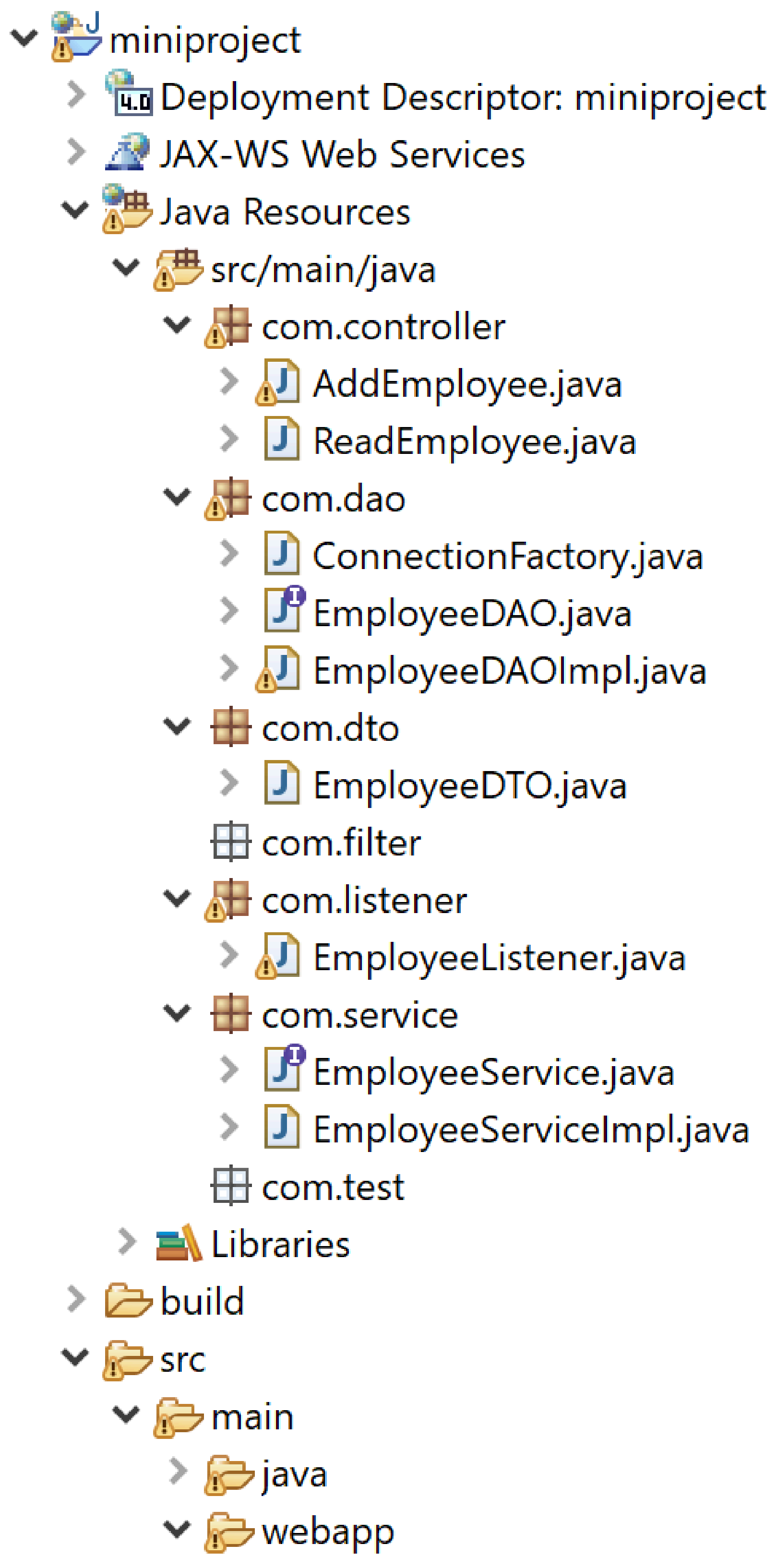
======================

DAY 20

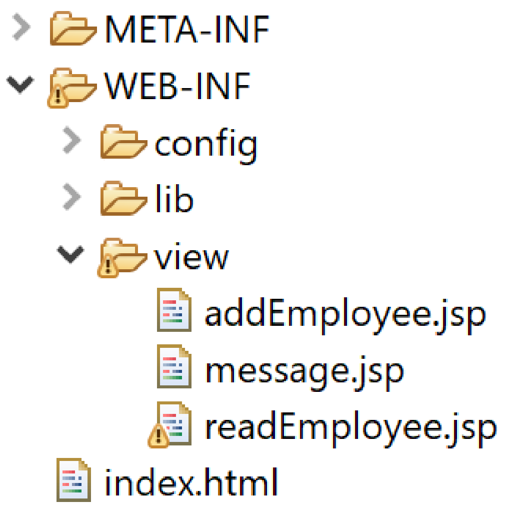
PROJECT PRACTICE

EMPLOYEE CRUD PROJECT

- Eid
- Ename
- Eaddress
- Salary



```
miniproject
  Deployment Descriptor: miniproject
  JAX-WS Web Services
  Java Resources
    src/main/java
      com.controller
        AddEmployee.java
        ReadEmployee.java
      com.dao
        ConnectionFactory.java
        EmployeeDAO.java
        EmployeeDAOImpl.java
      com.dto
        EmployeeDTO.java
      com.filter
      com.listener
        EmployeeListener.java
      com.service
        EmployeeService.java
        EmployeeServiceImpl.java
      com.test
    Libraries
  build
  src
    main
      java
      webapp
```



```
package com.controller;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.dto.EmployeeDTO;
import com.service.EmployeeService;
import com.service.EmployeeServiceImpl;

@WebServlet(urlPatterns = {"/viewAddEmployee","/addEmployee"})
public class AddEmployee extends HttpServlet {
      private static final long serialVersionUID = 1L;

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

request.getRequestDispatcher("/WEB-INF/view/addEmployee.jsp").forwar
d(request, response);
      }

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


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

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

		EmployeeDTO employeeDTO = new EmployeeDTO(name, address, 
Integer.parseInt(age), Integer.parseInt(salary));
		EmployeeService employeeService=new 
EmployeeServiceImpl();
		String result = employeeService.addEmployee(employeeDTO);

		request.setAttribute("msg", result);

request.getRequestDispatcher("/WEB-INF/view/addEmployee.jsp").forwar
d(request, response);

	}
}
```

```
package com.controller;
import java.io.IOException;
import java.util.ArrayList;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.dto.EmployeeDTO;
import com.service.EmployeeServiceImpl;

@WebServlet("/readEmployee")
public class ReadEmployee extends HttpServlet {
	private static final long serialVersionUID = 1L;

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

		EmployeeServiceImpl employeeServiceImpl = new 
EmployeeServiceImpl();
		ArrayList<EmployeeDTO> employee = 
employeeServiceImpl.readEmployee();

		request.setAttribute("data", employee);

request.getRequestDispatcher("/WEB-INF/view/readEmployee.jsp").forwa
rd(request, response);

	}
}
```

```
package com.dao;
import java.sql.Connection;
import java.sql.SQLException;
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;
public class ConnectionFactory {

      private static HikariDataSource  hikariDataSource;

      public static void init(String url,String user,String
pass,String driver) {

         try {
            Class.forName(driver);
            HikariConfig hikariConfig = new HikariConfig();
            hikariConfig.setJdbcUrl(url);
            hikariConfig.setUsername(user);
            hikariConfig.setPassword(pass);

            hikariDataSource= new HikariDataSource(hikariConfig);

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

      public static Connection getConnection() throws SQLException {
            return hikariDataSource.getConnection();
      }
}
```

```
package com.dao;
import java.util.ArrayList;
import com.dto.EmployeeDTO;

public interface EmployeeDAO {

      void createTable();
      String addEmployee(EmployeeDTO employeeDTO);
      ArrayList<EmployeeDTO> readEmployee();

}
```

```java
package com.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import com.dto.EmployeeDTO;

public class EmployeeDAOImpl implements EmployeeDAO {

	@Override
	public void createTable() {
		try {

			String sql="create table employee(sn int primary key 
auto_increment,"
					+ "name varchar(30),"
					+ "address varchar(100),"
					+ "age int,"
					+ "salary int)";

			Connection connection = 
ConnectionFactory.getConnection();
			Statement statement = connection.createStatement();
			statement.execute(sql);

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

	@Override
	public String addEmployee(EmployeeDTO employeeDTO) {

	 try {
			String sql="insert into 
employee(name,address,age,salary) values(?,?,?,?)";
			Connection connection = 
ConnectionFactory.getConnection();
			PreparedStatement ps = 
connection.prepareStatement(sql);

			ps.setString(1, employeeDTO.getName());
			ps.setString(2, employeeDTO.getAddress());
			ps.setInt(3, employeeDTO.getAge());
```

```
                ps.setInt(4, employeeDTO.getSalary());

                int row = ps.executeUpdate();
                if(row==1) {
                     return "SAVED";
                }
                else {
                     return "FAILED TO SAVED";
                }
           } catch (Exception e) {
                e.printStackTrace();
                return "FAILED TO SAVE";
           }
      }

      @Override
      public ArrayList<EmployeeDTO> readEmployee() {

           ArrayList<EmployeeDTO> arrayList = new
ArrayList<EmployeeDTO>();

           try {
                String sql="select * from employee";
                Connection connection =
ConnectionFactory.getConnection();
                PreparedStatement ps =
connection.prepareStatement(sql);
                ResultSet rs = ps.executeQuery();

                while(rs.next()) {
                     EmployeeDTO employeeDTO = new
EmployeeDTO(rs.getString("name"), rs.getString("address"),
                               rs.getInt("age"),
rs.getInt("salary"));
                     arrayList.add(employeeDTO);
                }

           } catch (SQLException e) {
                e.printStackTrace();
           }
           finally {
                return arrayList;
           }
      }
}
```

```
package com.dto;
public class EmployeeDTO {
	private String name;
	private String address;
	private int age;
	private int salary;

	public EmployeeDTO() {
		// TODO Auto-generated constructor stub
	}

	public EmployeeDTO(String name, String address, int age, int
salary) {
		super();
		this.name = name;
		this.address = address;
		this.age = age;
		this.salary = salary;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

	public int getAge() {
		return age;
	}

	public void setAge(int age) {
		this.age = age;
	}

	public int getSalary() {
		return salary;
	}

	public void setSalary(int salary) {
		this.salary = salary;
	}

	@Override
	public String toString() {
		return "EmployeeDTO [name=" + name + ", address=" +
address + ", age=" + age + ", salary=" + salary + "]";
	}

}
```

```
package com.listener;
import java.io.InputStream;
import java.util.Properties;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
import com.dao.ConnectionFactory;
import com.dao.EmployeeDAO;
import com.dao.EmployeeDAOImpl;

@WebListener
public class EmployeeListener implements ServletContextListener {

	public void contextDestroyed(ServletContextEvent sce) {

	}

	public void contextInitialized(ServletContextEvent sce) {

		try {

			ServletContext servletContext =
sce.getServletContext();

//			String realPath =
servletContext.getRealPath("/WEB-INF/config/config.properties");
//			System.out.println(realPath);

			InputStream is =
servletContext.getResourceAsStream("/WEB-INF/config/config.propertie
s");
			Properties properties = new Properties();

			properties.load(is);

			String url  = (String) properties.get("db.url");
			String user  = (String) properties.get("db.user");
```

```java
				String pass  = (String) properties.get("db.pass");
				String driver  = (String)
properties.get("db.driver");

				ConnectionFactory.init(url, user, pass, driver);

				EmployeeDAO employeeDAO= new EmployeeDAOImpl();
				employeeDAO.createTable();

			} catch (Exception e) {
				// TODO: handle exception
			}

	}

}
```

```java
package com.service;
import java.util.ArrayList;
import com.dto.EmployeeDTO;

public interface EmployeeService {

	String addEmployee(EmployeeDTO employeeDTO);

	ArrayList<EmployeeDTO> readEmployee();

}
```

```java
package com.service;
import java.util.ArrayList;
import com.dao.EmployeeDAOImpl;
import com.dto.EmployeeDTO;

public class EmployeeServiceImpl implements EmployeeService {

	@Override
	public String addEmployee(EmployeeDTO employeeDTO) {

		//DATA PROCESSING LIKE VALIDATION
		EmployeeDAOImpl employeeDAOImpl = new EmployeeDAOImpl();
		return employeeDAOImpl.addEmployee(employeeDTO);
	}

	@Override
	public ArrayList<EmployeeDTO> readEmployee() {
		EmployeeDAOImpl employeeDAOImpl = new EmployeeDAOImpl();
```

```
            return employeeDAOImpl.readEmployee();
      }
}
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

  <a href="index.html">HOME</a>

  <%@include file="message.jsp" %>

   <form action="addEmployee"  method="post">
       <input type="text" name="name" placeholder="enter name"><br>
       <input type="text" name="address" placeholder="enter
address"><br>
       <input type="text" name="age" placeholder="enter age"><br>
       <input type="text" name="salary" placeholder="enter
salary"><br>
       <button>add</button>
   </form>
</body>
</html>
```

```
<%
   String msg=(String)request.getAttribute("msg");
   if(msg!=null){
%>
<%=msg%>
<%
  request.removeAttribute("msg");
} %>
```

```
<%@page import="com.dto.EmployeeDTO"%>
<%@page import="java.util.ArrayList"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>
<a href="index.html" >home</a>

  <%
      ArrayList<EmployeeDTO>  arrayList=(ArrayList<EmployeeDTO>
)request.getAttribute("data");
      for( EmployeeDTO employeeDTO : arrayList){
  %>
      NAME: <%=employeeDTO.getName() %> <br>
      ADDRESS:  <%=employeeDTO.getAddress() %><br>
      AGE:   <%=employeeDTO.getAge() %><br>
      SALARY:    <%=employeeDTO.getSalary() %><br>
  <%} %>

</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

   <h1>THIS IS MY WELCOME PAGE</h1>

   <a href="viewAddEmployee">ADD EMPLOYEE</a>  <br>
   <a href="readEmployee">READ EMPLOYEE</a>  <br>
   <a href="#">UPDATE EMPLOYEE</a>  <br>
   <a href="#">DELETE EMPLOYEE</a>  <br>
   <a href="#">UPLOAD FILE</a>  <br>

</body>
</html>
```