**Section 17:**

We use JAVA 8 because it is a JAVA SE 8 (LTS) long term support version, it will last long compare to other versions, and other LTS are JAVA SE 11(LTS), JAVA SE 17 (LTS), and JAVA SE 21(LTS)

**Download Open JDK By Microsoft:**

**Open JDK 17:** [**Download**](https://aka.ms/download-jdk/microsoft-jdk-17.0.14-windows-x64.msi)

**Download Apache Tomcat:**

**Download Tomcat 10:**

Here they changed Java EE to Jakarta EE, javax.\* to Jakarta.\*, Previous version (Tomcat 9, 8) uses Javax know they use Jakarta.

**Download Link:** [**Download**](https://dlcdn.apache.org/tomcat/tomcat-10/v10.1.36/bin/apache-tomcat-10.1.36.exe)

**Servlet Life Cycle:**

The entire life cycle of a Servlet is managed by the **Servlet container** which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it.

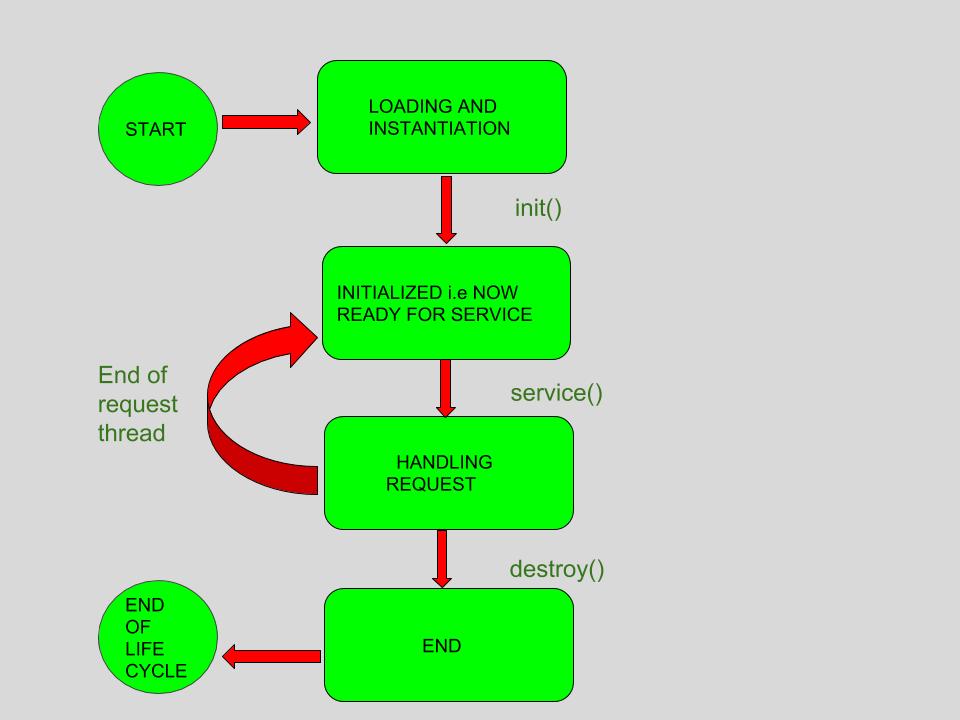
**Stages of the Servlet Life Cycle**: The Servlet life cycle mainly goes through four stages,

* Loading a Servlet.
* Initializing the Servlet.
* Request handling.
* Destroying the Servlet.

1. **Loading a Servlet**: The first stage of the Servlet lifecycle involves loading and initializing the Servlet by the Servlet container. The Web container or Servlet Container can load the Servlet at either of the following two stages :
   * Initializing the context, on configuring the Servlet with a zero or positive integer value.
   * If the Servlet is not preceding stage, it may delay the loading process until the Web container determines that this Servlet is needed to service a request.

The Servlet container performs two operations in this stage :

* + **Loading :** Loads the Servlet class.
  + **Instantiation :** Creates an instance of the Servlet. To create a new instance of the Servlet, the container uses the no-argument constructor.



1. **Initializing a Servlet**: After the Servlet is instantiated successfully, the Servlet container initializes the instantiated Servlet object. The container initializes the Servlet object by invoking the **Servlet.init(ServletConfig)** method which accepts ServletConfig object reference as parameter.

The Servlet container invokes the **Servlet.init(ServletConfig)** method only once, immediately after the **Servlet.init(ServletConfig)** object is instantiated successfully. This method is used to initialize the resources, such as JDBC datasource.

Now, if the Servlet fails to initialize, then it informs the Servlet container by throwing the **ServletException** or **UnavailableException**.

1. **Handling request**: After initialization, the Servlet instance is ready to serve the client requests. The Servlet container performs the following operations when the Servlet instance is located to service a request :
   * It creates the **ServletRequest** and **ServletResponse** objects. In this case, if this is a HTTP request, then the Web container creates **HttpServletRequest** and **HttpServletResponse** objects which are subtypes of the **ServletRequest** and **ServletResponse** objects respectively.
   * After creating the request and response objects it invokes the Servlet.service(ServletRequest, ServletResponse) method by passing the request and response objects.

The **service()** method while processing the request may throw the **ServletException** or **UnavailableException** or **IOException**.

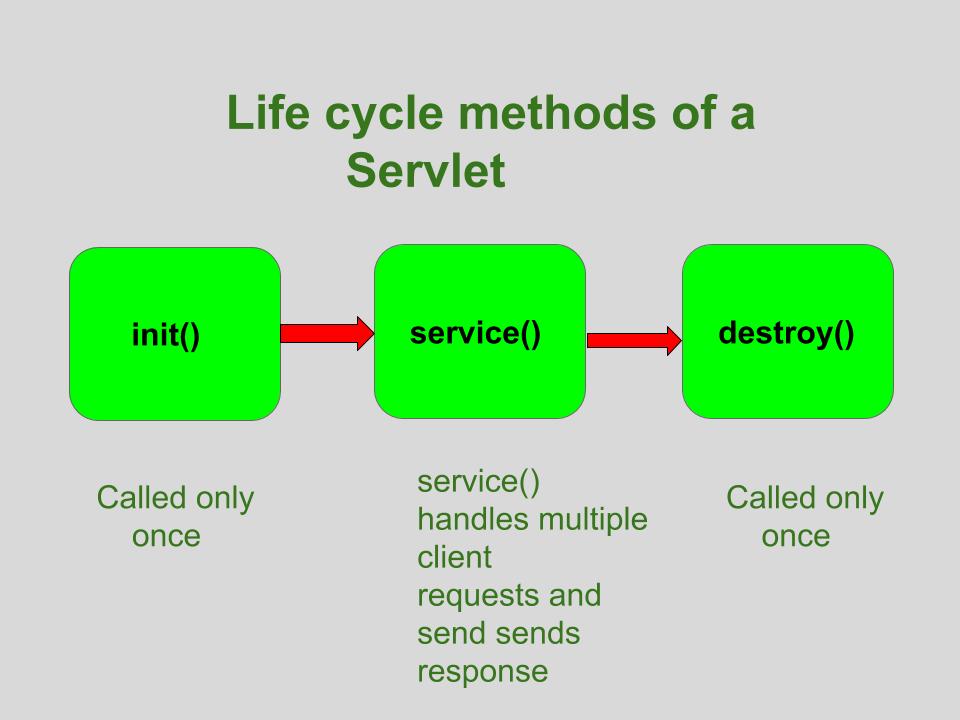
1. **Destroying a Servlet**: When a Servlet container decides to destroy the Servlet, it performs the following operations,
   * It allows all the threads currently running in the service method of the Servlet instance to complete their jobs and get released.
   * After currently running threads have completed their jobs, the Servlet container calls the **destroy()** method on the Servlet instance.

After the **destroy()** method is executed, the Servlet container releases all the references of this Servlet instance so that it becomes eligible for garbage collection.

**Servlet Life Cycle Methods**

There are three life cycle methods of a Servlet :

* init()
* service()
* destroy()



Let’s look at each of these methods in details:

1. **init() method**: The **Servlet.init()** method is called by the Servlet container to indicate that this Servlet instance is instantiated successfully and is about to put into service.
2. //init() method
3. public class MyServlet implements Servlet{
4. public void init(ServletConfig config) throws ServletException {
5. //initialization code
6. }
7. //rest of code
8. }
9. **service() method**: The **service()** method of the Servlet is invoked to inform the Servlet about the client requests.
   * This method uses **ServletRequest** object to collect the data requested by the client.
   * This method uses **ServletResponse** object to generate the output content.
10. // service() method
11. public class MyServlet implements Servlet{
12. public void service(ServletRequest res, ServletResponse res)
13. throws ServletException, IOException {
14. // request handling code
15. }
16. // rest of code
17. }
18. **destroy() method**: The **destroy()** method runs only once during the lifetime of a Servlet and signals the end of the Servlet instance.
19. //destroy() method
20. public void destroy()

As soon as the **destroy()** method is activated, the Servlet container releases the Servlet instance.

**Servlet Life Cycle:**  
Servlet life cycle can be defined as the stages through which the servlet passes from its creation to its destruction.  
The servlet life cycle consists these stages:

* Servlet is borned
* Servlet is initialized
* Servlet is ready to service
* Servlet is servicing
* Servlet is not ready to service
* Servlet is destroyed

**Life cycle methods:**  
Life cycle methods are those methods which are used to control the life cycle of the servlet. These methods are called in specific order during the servlets’s entire life cycle.  
The class **Servlet** provides the methods to control and supervise the life cycle of servlet. There are three life cycle methods in the Servlet interface. There are as follows:

* **init() method :**
  1. A servlet’s life begins here .
  2. This method is called only once to load the servlet.Since it is called only once in it’s lifetime,therefore “connected architecture” code is written inside it because we only want once to get connected with the database.  
     **Now Question Arises is that:-**  
     Q.Why can’t we write connected architecture code inside the constructor, since constructor also run only once in it’s entire life?  
     Ans. Suppose if the connection doesn’t get established, then we can throw an exception from init() and the rest of the steps stop executing. But in the constructor we can’t use, throw in it’s prototype otherwise it is an error.
  3. This method receives only one parameter, i.e **ServletConfig** object.
  4. This method has the possibility to throw the ServletException.
  5. Once the servlet is initialized, it is ready to handle the client request.
  6. The prototype for the init() method:

public void init(ServletConfig con)throws ServletException{ }

where **con** is ServletConfig object

* **NOTE:-** In programs of servlet,we use non parameterized version of init().
* **Now,Question Arises is that:-**  
  Q. Why it is recommended to use the non parameterized version of init() instead of parameterized version as seen above?  
  Ans. To answer this, we have to go into detail. Think like developers,i.e there must be some valid reason for this and the answer will blow your mind. Coming to answer:
* **APPROACH 1**  
  Whenever the lifecycle method of a servlet starts executing,i.e when public void init(ServletConfig con) throws ServletException gets call then our class public void init(ServletConfig con) throws ServletException gets called but we have to run the code which initializes servlet config object which is written inside “HttpServlet” method public void init(ServletConfig con) throws ServletException,i.e:  
  Coding of HttpServlet class be like:
* public void init(ServletConfig con) throws ServletException
* {
* //code to initialise ServletConfig object
* init(); //This HttpServlet has 2 init() one which is parameterized and the other one is non
* //parameterized.But this non parameterized version of init() has a blank body.
* //So this call is useless.
* }
* Now see the coding of our class
* public void init(ServletConfig con) throws ServletException
* {
* super.init(con); //Since,our class init() will run first,but to run HttpServlet init() we
* // have used super keyword.And Database connectivity code will be their
* }
* **NOTE:-** As we can see, total 3 init() calls we have to make.First init() gets called of our class then of HttpServlet class then non parameterized version of HttpServlet class.
* But now, we will achieve the same thing with less number of calls:
* **APPROACH 2**  
  Coding of HttpServlet parametrized and non parameterized versions of init() will remain the same. **But in our class instead of overriding parameterized version of init(), we will override non parameterized version of init()**.
* Let’s see the coding of our class non parameterized version of init():
* public void init() throws ServletException
* {
* //database connectivity code
* }
* **NOTE:** Since this method public void init() throws ServletException ,we have override from HttpServlet class whose coding is like:
* public void init() throws ServletException
* {
* //empty body
* }
* Since it’s body is blank, therefore it is known as **“Helper method”** as it is used for overriding purpose.
* Now, as the servlet starts executing its methods, it will call the parameterized version of init(). Since we have not to override the parameterized version, therefore it will give a call to the HttpServlet parameterized version of init(). Since coding of a parameterized version of init() of HttpServlet is as same as above, therefore, from there on it will call init() (i.e non parameterized version of init). It will give a call to our class non parameterized version of init() and the code continues.  
  **Now, as you can see, total number of init() calls are 2 which is less than the first approach. Therefore, execution time is less in 2nd approach and less headache for CPU for maintaining stack and it’s speed increases as compared to 1st approach.**  
  Therefore, it is highly recommended to override non parameterized version of init().Although both will run but due to efficiency first approach is rarely used and also in first approach we have to use super keyword too.Therefore in below mentioned program,we have override non parameterized version of init().
* **service() method :**
  1. The service() method is the most important method to perform that provides the connection between client and server.
  2. The web server calls the service() method to handle requests coming from the client( web browsers) and to send response back to the client.
  3. This method determines the type of Http request (GET, POST, PUT, DELETE, etc.) .
  4. This method also calls various other methods such as doGet(), doPost(), doPut(), doDelete(), etc. as required.
  5. This method accepts two parameters.
  6. The prototype for this method:
  7. public void service(ServletRequest req, ServletResponse resp)

throws ServletException, IOException { }

where

* + - **req** is the ServletRequest object which encapsulates the connection from client to server
    - **resp** is the ServletResponse object which encapsulates the connection from server back to the client
* **destroy() method :**
  1. The destroy() method is called only once.
  2. It is called at the end of the life cycle of the servlet.
  3. This method performs various tasks such as closing connection with the database, releasing memory allocated to the servlet, releasing resources that are allocated to the servlet and other cleanup activities.
  4. When this method is called, the garbage collector comes into action.
  5. The prototype for this method is:

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

Below is a sample program to illustrate Servlet in Java:

|  |
| --- |
| // Java program to show servlet example  // Importing required Java libraries  import java.io.\*;  import javax.servlet.\*;  import javax.servlet.http.\*;  // Extend HttpServlet class  public class AdvanceJavaConcepts extends HttpServlet  {  private String output;  // Initializing servlet  public void init() throws ServletException  {  output = "Advance Java Concepts";  }  // Requesting and printing the output  public void doGet(HttpServletRequest req,  HttpServletResponse resp)  throws ServletException, IOException  {  resp.setContentType("text/html");  PrintWriter out = resp.getWriter();  out.println(output);  }  public void destroy()  {  System.out.println("Over");  }  } |

**Section 17: Video 235:**

**main -> webapp -> WEB-INF -> lib -> web.xml**

This file contain welcome file, if didn’tmark url, then also by default it will redirect to default pages.

**JSP Scripting elements:**

In JSP, java code can be written inside the jsp page using the scriptlet tag.

**JSP Scripting elements**

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

* scriptlet tag
* expression tag
* declaration tag

**JSP scriptlet tag**

A scriptlet tag is used to execute java source code in JSP. Syntax is as follows:

1. <%  java source code %>

**Example of JSP scriptlet tag**

In this example, we are displaying a welcome message.

1. **<**html**>**
2. **<**body**>**
3. **<**% out.print("welcome to jsp"); %**>**
4. **</**body**>**
5. **</**html**>**

Example of JSP scriptlet tag that prints the user name

In this example, we have created two files index.html and welcome.jsp. The index.html file gets the username from the user and the welcome.jsp file prints the username with the welcome message.

File: index.html

1. **<**html**>**
2. **<**body**>**
3. **<**form action="welcome.jsp"**>**
4. **<**input type="text" name="uname"**>**
5. **<**input type="submit" value="go"**><**br**/>**
6. **</**form**>**
7. **</**body**>**
8. **</**html**>**

File: welcome.jsp

1. <html>
2. <body>
3. <%
4. String name=request.getParameter("uname");
5. out.print("welcome "+name);
6. %>
7. </form>
8. </body>
9. </html>

# Declaration Tag

Declaration tag is one of the scripting elements in JSP.  
This Tag is used for declare the variables. Along with this, Declaration Tag can also declare method and classes. Jsp initializer scans the code and find the declaration tag and initializes all the variables, methods, and classes. JSP container keeps this code outside of the service method (\_JSPService()) to make them class level variables and methods.

**Syntax of JSP-Declaration Tag**

* HTML

|  |
| --- |
| <%!  inside this tag we can initialise  our variables, methods and classes  %> |

**Example of JSP Declaration Tag which initialize a string**

* HTML

|  |
| --- |
| <%@ page language="java" contentType="text/html;   charset=ISO-8859-1"pageEncoding="ISO-8859-1"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01  Transitional//EN" "<http://www.w3.org/TR/html4/loose.dtd>">  <html>    <head>  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  <title>GeeksforGeeks</title>  </head>    <body>  <!--declaration of username variable....  -->  <%! String username="Geeks"; %>    <!--In expression tag a string is initialised as Geeks -->  <%="Hello : "+username %>    <!-- Displaying expression using Expression Tag -->  </body>  </html> |

**JSP comment**

JSP comment marks to text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out", a part of your JSP page.

Following is the syntax of the JSP comments −

<%-- This is JSP comment --%>

Following example shows the JSP Comments −

<html>

<head>

<title>A Comment Test</title>

</head>

<body>

<h2>A Test of Comments</h2>

<%-- This comment will not be visible in the page source --%>

</body>

</html>

The above code will generate the following result −

A Test of Comments

There are a small number of special constructs you can use in various cases to insert comments or characters that would otherwise be treated specially. Here's a summary −

| **S.No.** | **Syntax & Purpose** |
| --- | --- |
| 1 | **<%-- comment --%>** A JSP comment. Ignored by the JSP engine. |
| 2 | **<!-- comment -->** An HTML comment. Ignored by the browser. |
| 3 | **<\%** Represents static <% literal. |
| 4 | **%\>** Represents static %> literal. |
| 5 | **\'** A single quote in an attribute that uses single quotes. |
| 6 | **\"** A double quote in an attribute that uses double quotes |

**Deployment description and annotations:**

As we learned in previous chapters, the Servlet life cycle will be managed by the Servlet container that is under the Web/Application server. Whenever an HttpRequest comes from the client browser, the servlet container will map the request to the corresponding servlet based on the URL mappings provided in the deployment descriptor file – web.xml. **For example:** Consider the below servlet mappings in the web.xml file.

* XML

|  |
| --- |
| <servlet>     <servlet-name>HelloServlet</servlet-name>     <servlet-class>HelloServlet</servlet-class>  </servlet>    <servlet-mapping>     <servlet-name>HelloServlet</servlet-name>     <url-pattern>/hello</url-pattern>  </servlet-mapping> |

Here, whenever there is a “**/hello**” URL coming from the client request, we are mapping it to the “**HelloServlet**” class. Instead of providing these mappings in the **web.xml** file, we can simply provide an annotation in the Servlet as below,

* Java

|  |
| --- |
| @WebServlet("/hello")  public class HelloServlet extends HttpServlet {      // Code to be executed...  } |

### Servlet Annotations

Java Servlets provides a number of annotations to allow users to declare servlets, filters, listeners in the **javax.servlet.annotation** package. It also provides annotations to specify metadata for the components that are declared.

**Annotation Type WebServlet:**

* Java

|  |
| --- |
| public @interface WebServlet |

* Available in javax.servlet.annotation package.
* Used to declare a Servlet.
* At the time of deployment, the servlet container will identify and process the annotation and makes the corresponding servlet available at the specified URL patterns.

**Elements of @WebServlet Annotation:**

| **Name** | **Modifier and Type** | **Description** | **Default value** |
| --- | --- | --- | --- |
| asyncSupported | boolean | To specify whether the servlet supports asynchronous mode or not. | false |
| description | String | To provide a description of the servlet. | “” |
| displayName | String | To provide the Display name of the Servlet. | “” |
| initParams | WebInitParam[] | To specify initialization parameters of the servlet. | {} |
| name | String | To provide the name of the servlet. | “” |
| value | String[] | To specify the URL pattern of the servlet that is to be mapped. | {} |
| urlPatterns | String[] | To provide the URL patterns of the servlet that are to be mapped. | {} |
| smallIcon | String | To specify the small icon name of the servlet. | “” |
| loadOnStartup | int | To provide the load on the startup order of the servlet. | -1 |
| largeIcon | String | To specify the large icon name of the servlet. | “” |

The elements displayName, description, smallIcon and largeIcon are basically used by IDEs, tools or servlet containers. These do not affect the execution of the servlet.

### Servlet with Annotation Examples

For better understanding, we will create a simple HTML page to map the Servlet with the URL.

**Example 1:**

**Servlet annotated with only URL pattern:**

**index.html**

* HTML

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Home</title>  </head>  <body>      <form action="hello" method="post">            Welcome Page: <input type="submit" />        </form>  </body>  </html> |

Html page which maps the servlet with URL “/hello”.

**HelloServlet.java**

* Java

|  |
| --- |
| import java.io.IOException;  import java.io.PrintWriter;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;    @WebServlet("/hello")  public class HelloServlet extends HttpServlet {      private static final long serialVersionUID = 1L;        protected void doPost(HttpServletRequest request,                            HttpServletResponse response)          throws ServletException, IOException      {            // set the response content type          response.setContentType("text/html");          PrintWriter out = response.getWriter();            // Print hello message to the client browser in          // response object          out.println(              "<h3>Hello, Welcome to GeeksforGeeks!!</h3>");          out.close();      }  } |

* When user clicks submit for the welcome page, container will process the “**@WebServlet(“/hello”)**” annotation and maps the “**HelloServlet**” class.
* As the method is mentioned as “**post**” in Html page, container will execute “**doPost()**” method in “**HelloServlet**” class.

**File Include:**

<**h1**>Include File</**h1**>

<%-- Include file1.txt and file2.txt

This is a static include directive. used to include static files.

--%>

<%@ **include** file = *"file1.txt"* %> <**br**>

<%@ **include** file = *"file2.txt"* %><**br**>

<%--including file using jsp:include tag

This is a Dynamic include. Used to include dynamic files.

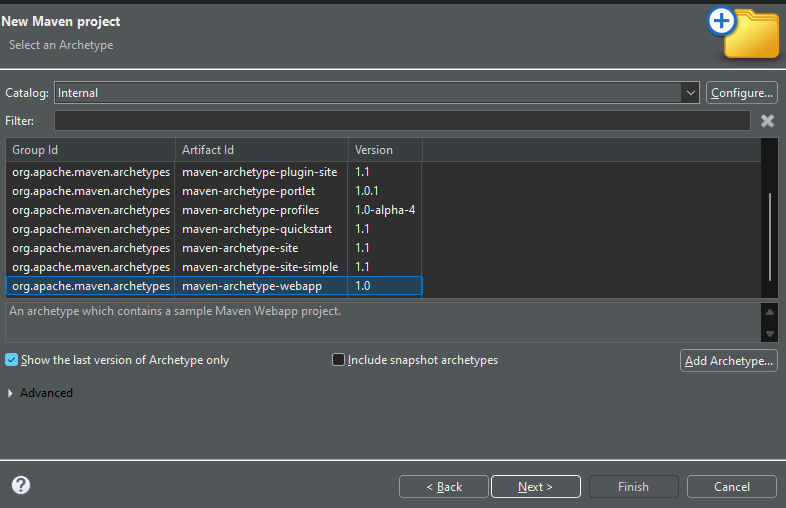
--%>

<**jsp:include** page=*"file1.txt"*></**jsp:include**>

</**body**>

**Creating Maven Project:**

These all are Archetypes boiler code (boiler plate) given by maven, we use org.apache.maven.archetypes -> web app

****

Pom.xml:

Dependencies: <https://mvnrepository.com/artifact/jakarta.servlet/jakarta.servlet-api/6.0.0>

Add below code in pom.xml

<!-- https://mvnrepository.com/artifact/jakarta.servlet/jakarta.servlet-api -->

<dependency>

<groupId>jakarta.servlet</groupId>

<artifactId>jakarta.servlet-api</artifactId>

<version>6.0.0</version>

<scope>provided</scope>

</dependency>

Still, getting error in HTML tag, go for Maven Update

Right Click on Project -> Maven -> Project update

**Forward and Redirect**

In JSP, "forward" refers to sending a request to another resource within the same web application without changing the URL in the browser, while "redirect" means sending the user to a new URL, causing the browser to make a separate request to that new location

forwarding.

Code

<%

RequestDispatcher rd = request.getRequestDispatcher("newPage.jsp");

rd.forward(request, response);

%>

Redirecting.

Code

<%

response.sendRedirect("newPage.jsp");

%>

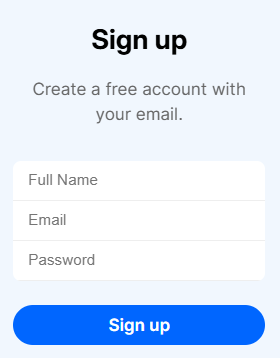
**The Concept of MVC:**

* **Model –** A model can be an object or collection of objects which basically contains the data of the application.
* **View –** A view is used for displaying the information to the user in a specific format. Spring supports various technologies like **freemarker**, **velocity**, and **thymeleaf**.
* **Controller –** It contains the logical part of the application. *@Controller* annotation is used to mark that class as a controller.Stores the data in variable and passed it to the model.
* **Front Controller –** It remains responsible for managing the flow of the web application. Dispatcher Servlet acts as a front controller in Spring MVC.

Controller

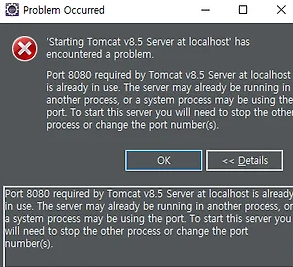


Model



Tomcat Error :

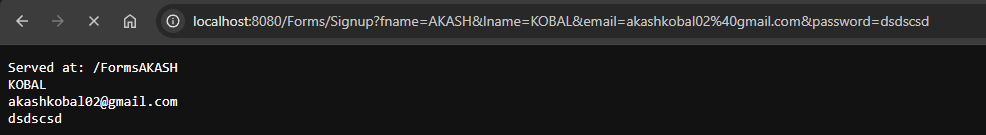
# **[Tomcat\_Error] Server Tomcat v9.0 Server at localhost failed to start](https://devlimk1.tistory.com/10" \t "_blank)**



Solution: Section 18, Video 250, at 6:30 timestamp

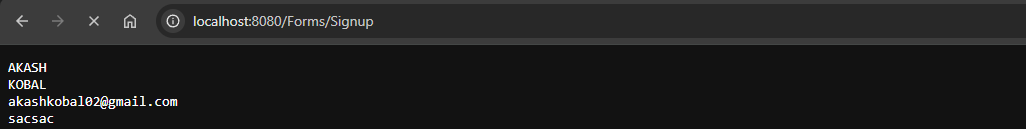
**Form:**

**When we use get method to submit form**

****

User is able to see the details in the URL and submitted to the controller or server.

**When we use post method to submit form**

****

User can’t see the details in the URL, only details is submitted to the controller or server.

**Section 19: Java Beans**

JavaBeans are [classes](https://www.geeksforgeeks.org/classes-objects-java/)that [encapsulate](https://www.geeksforgeeks.org/encapsulation-in-java/) many objects into a single object (the bean). It is a Java class that should follow the following conventions:

1. Must implement [Serializable](https://www.geeksforgeeks.org/serialization-in-java/).
2. It should have a public no-arg constructor.
3. All properties in java bean must be private with public getters and setter methods.

A simple example of JavaBean Class is mentioned below:

* JAVA

|  |
| --- |
| // Java program to illustrate the  // structure of JavaBean class  public class TestBean {      private String name;        public void setName(String name) {        this.name = name;      }        public String getName() { return name; }  } |

Getter and Setter have important roles in the topic. So, let us check on Getter and Setter below:

**Setter and Getter Methods in Java**

Setter and Getter Methods in Java properties are mentioned below:

**Properties for setter methods:**

1. It should be public in nature.
2. The return type **a**should be void.
3. The setter method should be prefixed with the set.
4. It should take some argument i.e. it should not be a no-arg method.

**Properties for getter methods:**

1. It should be public in nature.
2. The return type should not be void i.e. according to our requirement, **return type** we have to give the return type.
3. The getter method should be prefixed with get.
4. It should not take any argument.

For Boolean properties getter method name can be prefixed with either “get” or “is”. But recommended to use “is”.

* JAVA

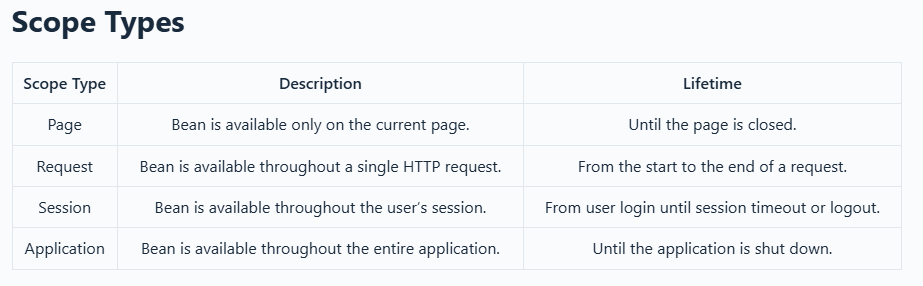
|  |
| --- |
| // Java program to illustrate the  // getName() method on boolean type attribute  public class Test {      private boolean empty;        public boolean getName(){        return empty;      }          public boolean isempty(){        return empty;      }  } |

JavaBeans are reusable software components that follow specific conventions, making them easy to manage and manipulate within various Java applications. One of the critical aspects of JavaBeans is their scope, which determines the lifecycle and visibility of the bean within an application. Understanding the different scopes is essential for developers to manage state and data effectively.

**Java Beans Scopes Overview**

JavaBeans can exist in different scopes, each defining their lifespan and accessibility within a web application. The main scopes available are:

1. **Page Scope**
2. **Request Scope**
3. **Session Scope**
4. **Application Scope**

****

**Session Scope**

**Session Scope** means the bean is tied to a user’s session. This is useful for maintaining user-specific data across multiple requests and pages within the application.

**Application Scope**

**Application Scope** extends the bean’s availability across the entire application, making it accessible to all users and sessions. This scope is ideal for shared resources or configuration settings that remain constant throughout the application’s lifecycle.

set Property : scope session

<**jsp:useBean** id=*"user"* class=*"User.User"* scope=*"session"*>

</**jsp:useBean**>

First Name : <**jsp:setProperty** property=*"fname"* name=*"user"* value=*"john"*/><**br**>

Last Name : <**jsp:setProperty** property=*"lname"* name=*"user"* value=*"monk"*/><**br**>

get Property: scope session

<**jsp:useBean** id=*"user"* class=*"User.User"* scope=*"session"*></**jsp:useBean**>

First Name : <**jsp:getProperty** property=*"fname"* name=*"user"*/><**br**>

Last Name : <**jsp:getProperty** property=*"lname"* name=*"user"*/>

When a user sets the properties via **setProperty.jsp**, the updated values are stored in the session. Refreshing or navigating within the same session (same browser) retains these values. However, accessing the application from a different browser or tab initiates a new session, and the previously set values are not visible.

## Implementing Application Scope

### Application Scope Example

To maintain bean data across the entire application, regardless of user sessions, we use **Application Scope**. This ensures that the bean’s state persists until the application is shut down.

getProperty :scope = “*application”*

<**jsp:useBean** id=*"user"* class=*"User.User"* scope=*"application"*></**jsp:useBean**>

First Name : <**jsp:getProperty** property=*"fname"* name=*"user"*/><**br**>

Last Name : <**jsp:getProperty** property=*"lname"* name=*"user"*/>

setProperty :scope = “*application”*

<**jsp:useBean** id=*"user"* class=*"User.User"* scope=*"application"*></**jsp:useBean**>

First Name : <**jsp:getProperty** property=*"fname"* name=*"user"*/><**br**>

Last Name : <**jsp:getProperty** property=*"lname"* name=*"user"*/>

**When and Where to Use Session vs. Application Scope**

**When to Use Session Scope**

* **User Authentication**: Storing user login details during a session.
* **Shopping Carts**: Maintaining items selected by a user in an e-commerce application.
* **User Preferences**: Retaining user-specific settings across multiple pages.

**When to Use Application Scope**

* **Configuration Settings**: Storing application-wide settings accessible by all users.
* **Shared Resources**: Managing resources like connection pools or logging mechanisms.
* **Global Data**: Maintaining data that needs to be consistent across the entire application, such as application status or shared counters.

JavaBeans can have various scopes, determining their lifespan and visibility within a web application. Two commonly used scopes are **Page** and **Request**. Each scope offers distinct behaviors that cater to different aspects of web application functionality.

**Page Scope**

**Page Scope** confines the JavaBean to a single page (JSP). The bean is created and accessible only within the specific page it’s defined in. This scope ensures that the bean’s data is available during the processing of the page but is discarded once the page processing is complete.

**Key Characteristics:**

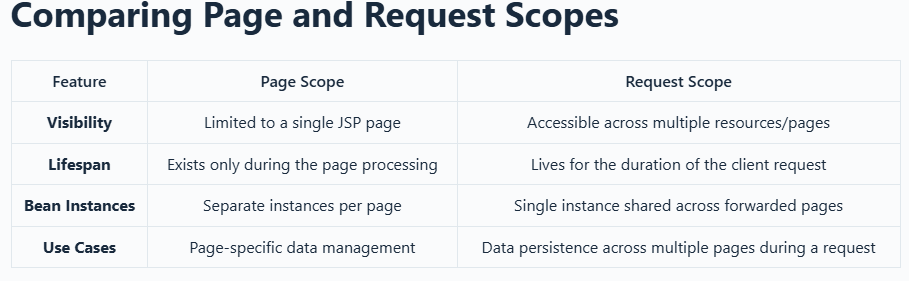
* **Visibility:** Limited to the page where it’s defined.
* **Lifespan:** Exists only during the page request.
* **Use Case:** Ideal for beans that manage data specific to a single page without needing to persist beyond it.

**Request Scope**

**Request Scope** extends the visibility of a JavaBean across multiple resources within a single client request. Unlike Page Scope, a Request Scoped bean remains available even if the request is forwarded to another page or resource.

**Key Characteristics:**

* **Visibility:** Accessible across multiple resources during a single request.
* **Lifespan:** Lives for the duration of the client request.
* **Use Case:** Suitable for scenarios where data needs to persist across multiple pages during a single interaction, such as form submissions that span multiple pages.



**Why Java Beans with Web Forms?**

* **Efficiency**: Simplifies data handling between client and server.
* **Maintainability**: Enhances code organization and readability.
* **Reusability**: Promotes reusable components, reducing redundancy.

**Pros and Cons**

****

**When and Where to Use Java Beans with Web Forms**

Java Beans with web forms are ideal for applications that require structured form data handling, such as user registration forms, feedback systems, and any scenario where data persistence and integrity are crucial.

**Understanding Java Beans**

Java Beans are reusable software components that follow specific conventions, making them ideal for encapsulating data and business logic. They play a pivotal role in Java EE applications, especially when dealing with user inputs through web forms.

**Key Characteristics of Java Beans**

* **Encapsulation**: Private properties with public getter and setter methods.
* **Serialization**: Implements **Serializable** interface.
* **No-Argument Constructor**: Facilitates easy instantiation.

form.jsp

<**body**>

<**jsp:useBean** id=*"user"* class=*"User.Form"* scope=*"session"*></**jsp:useBean**>

<**form** action=*"formValue.jsp"*>

Submit form <**br**>

First Name : <**input** type=*"text"* name=*"fname"* value=*"*<**jsp:getProperty** property=*"fname"* name=*"user"*/>*"*/><**br**>

Last Name : <**input** type=*"text"* name=*"lname"* value=*"*<**jsp:getProperty** property=*"lname"* name=*"user"*/>*"*/><**br**>

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

</**form**>

</**body**>

formValue.jsp

<**body**>

Values submitted to forms.<**br**>

<**jsp:useBean** id=*"user"* class=*"User.Form"* scope=*"session"*></**jsp:useBean**>

<**jsp:setProperty** property=*"\*"* name=*"user"*/>

First Name : <**jsp:getProperty** property=*"fname"* name=*"user"*/><**br**>

Last Name : <**jsp:getProperty** property=*"lname"* name=*"user"*/>

</**body**>

Form.java

package User;

public class Form {

private String fname;

private String lname;

public Form() {

fname = "first";

lname = "last";

}

public String getFname() {

return fname;

}

public void setFname(String fname) {

this.fname = fname;

}

public String getLname() {

return lname;

}

public void setLname(String lname) {

this.lname = lname;

}

}

### Session Management in JSP:

### Session management is a critical aspect of web development, ensuring that user interactions with a web application are seamless and personalized. In JavaServer Pages (JSP), managing sessions effectively allows developers to create dynamic and user-specific experiences.

### Importance of Session Management

* **Personalization**: Tailor content based on user preferences and behavior.
* **Security**: Maintain user authentication and authorization.
* **State Management**: Preserve user data across multiple requests and interactions.



### When and Where to Use Session Management

Session management is essential in scenarios where:

* User authentication is required.
* Personalized content delivery is necessary.
* Data persistence across multiple pages is needed.

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

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

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

if(name.equals("akash") && password.equals("123456")) {

request.getSession().invalidate(); // first invalidate the session if any exists before creating a new one

HttpSession newSession = request.getSession();

newSession.setMaxInactiveInterval(500); // 50 seconds

response.sendRedirect("member.jsp");

}

else {

response.sendRedirect("login.jsp");

}

}

**Session** is used to save user information momentarily on the server. It starts from the instance the user logs into the application and remains till the user logs out of the application or shuts down the machine. In both cases, the session values are deleted automatically.

## What is Session Management?

Keeping track of the user's preferences over a website or an application as long as the user is logged into the system for a specific time is called Session Management. This lasts until they log out or their session expires. In simpler terms, it's like keeping a record of user activities when they simply surf the application so that from next time onwards, the user doesn't have to specify the preferences again.

## Why is it Required?

Sessions are required because they perform multiple functions ranging from time management to security perspective. The following can be noted in this regard.

* Sessions store the **user authentication credentials** in an encrypted format which can be passed in case of auto login.
* Since the values are encrypted, it can store sensitive information as well.
* Session values can be accessed in a short period cause they are stored on the client's end and thus it helps the server to **respond faster**.
* It can restrict**application access** from multiple servers, thus securing the network.

## Role of Cookies and Other Tracking Mechanisms

Cookies and other tracking mechanisms play a crucial role in session management by helping applications remember the users through multiple interactions and their preferences and behavior. Below are some of the mechanisms through which they manage the task.

### 1. Session Cookies

* **Identification:**When the user logs into the system, an ID is generated. This is a unique identifier that is stored in the session cookie assigned to an individual user which identifies the user on their forthcoming visits.
* **Temporary Storage:**The details stored in temporary in nature, such as login status and other session-specific data.
* **Automatic Logins:**They store the user's authentication credentials which helps the user to not provide the credentials every a time they are logging into the system. It performs as an auto login where it will take the details from the session cookie and login to the platform. This can be performed with some time frame to identify the user or in simpler words, this can be stored for a specific time period to maintain security of the system.

### 2. Persistent Cookies

* **Extended Recognition:** Unlike session cookies which expires when the user closes browser, persistent cookies generally have a longer lifespan.
* **Personalization:** Persistent cookies can store user preferences providing a personalized experience across multiple visits of the user to the platform.

### 3. Tracking Mechanisms

* **User Activity Tracking:**Websites may use JavaScript or pixels as tracking mechanisms to monitor the user's activity enhancing user experience and analytics.
* **Third-Party Cookies:**For this very purpose, some websites prefer third-party cookies which are generally set by domains.

### 4. Security Measures

* **CSRF Tokens:**Cookies can also store anti-CSRF (Cross-Site Request Forgery) tokens to prevent unauthorized actions, thus enhancing the overall security of the platform.
* **Secure and HttpOnly Flags:** Cookies can also be marked with HttpOnly flag to prevent client-side scripts from accessing them which convey as secure to ensure they are only transmitted over secure (HTTPS) connections, thus reducing the risk of certain types of attacks.

## How to Get a Session?

We can keep track of a client's session with the **HttpSession** object. In Java Servlets, the HttpSession interface provides a way to regulate the state/information about a user varying across multiple requests. It is a part of **javax.servlet.http** package and allows storing and retrieving the attributes about the user's information, providing a mechanism for session management.

### Retrieving HttpSession in Servlets

We can use the getSession() method to create a HttpSession object. The getSession() method returns a session if it already exists or creates a new one, in case no session exists. The following snippet follows.

HttpSession session = request.getSession();

Using the parameterized getSession(boolean) method, it returns a new session if we pass the true parameter. The syntax will look something like this.

HttpSession session = request.getSession(true);

Further, to fetch a pre-existing session, one can pass false in the parameterized getSession() method.

HttpSession session = request.getSession(false);

## Common Methods

**1. setAttribute(String name, Object value)**

This associates the specified value to the specified name in a particular session. This can be used to store data that needs to be managed across multiple requests.

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

**2. getAttribute(String name)**

Returns the value which is associated with the specified name in the particular session. This is used to retrieve previously stored session attributes.

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

**3. removeAttribute(String name)**

It removes the attribute with the specified name from the session.

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

**4. invalidate()**

Once the services are completed, it is necessary to destroy the session object. The syntax follows.

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

### Example

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

@WebServlet("/api")

**public** **class** **GFG** **extends** HttpServlet {

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

*// Retrieve the HttpSession object*

HttpSession session = request.getSession();

*// Use HttpSession methods*

session.setAttribute("username", "GFG");

String value = (String) session.getAttribute("username");

}

}

In the above example, the getSession() method is called on the HttpServletRequest object to retrieve the HttpSession. Once you have the HttpSession object, you can use its methods to set and retrieve session attributes.

## Session Tracking Mechanisms in Servlets

To maintain the session between a web server and a web client, following methods can be used.

### 1. Cookies

To address the client subsequent request, the web server assigns a unique session ID as a cookie to the web client so that for the next oncoming requests, this cookie can be passed and the server can identify its validity and can process the request.

This is not considered an efficient approach as it is browser dependent and some browser at times doesn't support a cookie. The general syntax follows.

Cookie cookie = new Cookie("sessionId", "123456");  
response.addCookie(cookie);

### 2. Hidden Form Fields

A web server can also pass a hidden HTML form field as an input element along with a unique session ID as follows.

<input type = "hidden" name = "sessionId" value = "12345">

This simply adds the name and value on each and every request which the client sends to the server and the response is received, the session\_id value can be used to track different browser actions.

Since clicking on a regular ankle tag (<a href="">) hypertext link doesn't results into form submission, so this can's support general session tracking and so can't be an efficient way to handle the transmission of information.

### 3. URL Rewriting

Simply appending the URL with some data or token as the session id which can be decoded at the server end, which can associate that data with its stored session values and can identify the legitimate request can be done by URL rewriting.

This appending can be done in the form of passing query parameters with a sessionid value. This can be accessed at the server side to identify the client request.

Though this can help to maintain the information flow between the server and the client and would be independent of the browser, but this is generated dynamically. Thus, this will not be considered a choice for static pages.

String urlWithSessionId = response.encodeURL("/api");

## Session Lifecycle

The stages which a user session goes through right from its creation to its eventual expiration completes the session lifecycle. In the context of web applications, the lifecycle involves the management of user-specific data across multiple session requests. The key stages involved in the session lifecycle are described below.

### 1. Session Creation

* **Trigger:** The session is created when the user logs into the application for the initial time.
* **Action:**The servlet container creates a new session for the user using HttpSession attribute when the user logs in the platform for the first time. As no existing session is associated with the user, it creates a new one and generates a session ID which is stored as a cookie on the client's browser based on the system requirements.

### 2. Attribute Setting

* **Trigger:** Once the session is created, the required attributes are set on the session using the servlets or the JSP page.
* **Action:**The required attribute are named and stored using the setAttribute method of the HttpSession object which makes sure that this can be use across multiple locations.

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

### 3. Client Interaction

* **Trigger:** The user interacts with the web application, making additional requests.
* **Action:** The session ID is now associated with each successive request, allowing the servlet container to map the request with the correct session. To retrieve the attributes associated to the particular session, use getAttribute method.

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

### 4. Session Invalidation

* **Trigger:** The session can be invalidated explicitly by the application or automatically based on certain criteria as per the system requirements.
* **Action:** We can invoke invalidate method to remove all the session attributes associated with that particular session. Further we can use a timeout concept which will invalidate the sessions automatically after a specified period of inactivity.

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

### 5. Session Expiration

* **Trigger:** We can set the session timeout, which is a defined maximum time of existence.
* **Action:**Once the browsing is completed, we can remove the data and the user's preference based on the maximum inactive time interval.

HttpSession session = request.getSession();  
// Session will be invalidated after 30 minutes of inactivity  
session.setMaxInactiveInterval(1800);

## Example of Session Management

Below example demonstrates creating a session, setting and retrieving attributes, and finally, invalidating the session.

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

@WebServlet("/api")

**public** **class** **GFG** **extends** HttpServlet {

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

**throws** ServletException, IOException {

*// Get the HttpSession object. If it doesn't exist, a new one will be created.*

HttpSession session = request.getSession();

*// Set a session attribute*

session.setAttribute("username", "GFG");

*// Retrieve the session attribute*

String username = (String) session.getAttribute("username");

*// Display the attribute value in the response*

response.getWriter().println("Username from Session: " + username);

*// Invalidate the session after displaying the attribute*

session.invalidate();

}

}

#### Explaination of the above Example:

* A servlet named /api is created.
* When a user accesses the servlet, the doGet method is called which retrieves the existing HttpSession associated with the request or creates a new one if it doesn't exist using the request.getSession() method.
* The servlet sets a session attribute named "username" to the value "GFG" using session.setAttribute.
* It retrieves the "username" attribute from the session using session.getAttribute and prints it in the HTTP response.
* Finally, the session.invalidate() method is called to invalidate the session.

## Conclusion

The concept of session management becomes crucial when dealing with dynamic and large web applications. A session provides the details about the user's interactions with a site which can be managed using HttpSession interface in Java Servlets