MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous Institution – UGC, Govt. of India

Department of COMPUTATIONAL INTELLIGENCE
(CSE-AIML& B. Tech AIML)

B. TECH (R-22 Regulation)

(III YEAR – II SEM)

2024-25

WEB TECHNOLOGIES

(R22A6616)

LECTURE NOTES

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad–500100, Telangana State, India

Department of COMPUTATIONAL INTELLIGENCE

(CSE-AIML& B. Tech AIML)

WEB TECHNOLOGIES

(R22A6616)

LECTURE NOTES

Prepared

by

Dr.G.L.N.Jayaprada

Department of Computational Intelligence
CSE (Artificial Intelligence and Machine Learning)

Vision

To be a premier centre for academic excellence and research through innovative interdisciplinary collaborations and making significant contributions to the community, organizations, and society as a whole.

Mission

- To impart cutting-edge Artificial Intelligence technology in accordance with industry norms.
- To instill in students a desire to conduct research in order to tackle challenging technical problems for industry.
- To develop effective graduates who are responsible for their professional growth, leadership qualities and are committed to lifelong learning.

QUALITY POLICY

- To provide sophisticated technical infrastructure and to inspire students to reach their full potential.
- To provide students with a solid academic and research environment for a comprehensive learning experience.
- To provide research development, consulting, testing, and customized training to satisfy specific industrial demands, thereby encouraging self-employment and entrepreneurship among students.

For more information: www.mrcet.ac.in

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

B.Tech IIIYear IISem-CSE(AI&ML) **L/T/P/C**

3/0/0/3

(R22A6616) WEB TECHNOLOGIES (Java Stack-2)

Professional Elective-II

COURSE OBJECTIVES:

The objectives of this course are

- To gain the knowledge of Server-side programming languages and techniques associated with the World Wide Web.
- To make the students get acquainted the skill for developing web apps.
- Understand how to use web-based media-rich programming tools for creating dynamic web pages.

COURSE OUTCOMES:

After the successful completion of this course, the students will be able to:

CO1: Interpret Servlet Life Cycle and web servers.
CO2: Illustrate JSP Life cycle.
CO3: Apply Session Management for JSP applications.
CO4: Experiment with the usage of JDBC in JSP applications.
CO5: Develop Spring Boot applications and Spring JDBC applications.

UNIT – I

Web Servers and Servlets: Tomcat web server, introducing java Servlet, Introducing the servlet API, Lifecycle of a Servlet, Working with Initialization Parameters, Describing the HttpServlet Request & HttpServletResponse interfaces, Cookies, Session Tracking, Security Issues.

UNIT – II

Introduction to JSP: The Problem with Servlet. The Anatomy of a JSP Page, JSP Processing. JSP Application Design with MVC. JSP Application Development: Generating Dynamic Content: directive elements, what is template text, action elements. Using Scripting Elements, Implicit JSP Objects. Conditional Processing: Displaying Values Using an Expression to Set an Attribute, Declaring Variables and Methods.

UNIT – III

JSP: Error Handling and Debugging, Implicit JSP Objects, Sharing Data between JSP pages, Requests, and Users Passing Control and Data between Pages: passing the control between the pages, passing the data between JSP pages passing the data between JSP pages using session object, Memory Usage Considerations.

UNIT - IV

Spring Boot: Introduction to Spring Boot, how to create a Spring Boot Project, Annotations, Architecture, Actuator, Introduction to RESTful Web Services, basic application in Spring Boot, REST API in java Spring Boot.

UNIT - V

JDBC: Introduction, Database Programming Using JDBC: How JDBC Works, JDBC Architecture, JDBC Driver Types, Accessing Database From JSP page: Use of Prepared Statement, ResultSet, Spring JDBC.

TEXT BOOKS:

1. Web Technologies –A.A. Puntambekar, Technical Publications, Pune.
2. Web Technologies-Black Book, Kognent Learning Solutions Inc Sol. DreamtTech Press
3. Spring Boot in Action, Craig Walls.

REFERENCE BOOKS:

1. An Introduction to Web Design and Programming –Wang Thomson.
2. An Introduction to Web Design + Prgramming Wang, Katila CENGAGE.
3. Web Technologies A Developer's Perspective, N.P.Gopalan , Akhilandeswari, PHI

WEB SERVERS AND SERVLETS

TOMCAT WEB SERVER

A Web Server is a computer or virtual machine used to run Web applications. The main purpose of the Web Server is to provide web pages to clients. A client uses a Web browser to make an HTTP request for a specific resource. We can deploy and run Java Web Applications on Tomcat Server which is an open source Web server introduced by Apache. Before installing the Tomcat server we must ensure that java is also installed on the computer. By default, the Tomcat server uses port 8080 to run web applications. After installing Tomcat, you can start or stop the Tomcat server by using its Windows service installed on our computer.

INTRODUCING JAVA SERVLET

Java Servlet is one of the component APIs of Java Platform Enterprise Edition. It sets standards for creating dynamic Web applications in Java. Java Servlet application can run on any Web Server integrated with a Servlet Container. Java Servlet is usually designed to process Http request, such as GET and POST.

Figure below shows how a Servlet works at higher level.

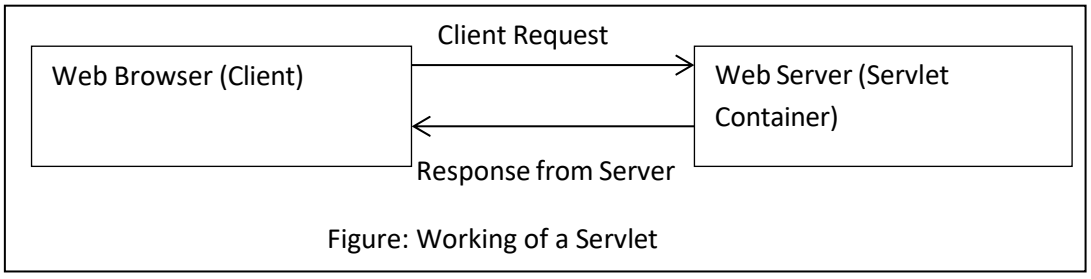


Figure: Working of a Servlet

The process can be summarized as follows.

- A client sends a request to a Servlet container.
- The Web server searches for the required Servlet and initiates it.
- The Servlet then processes the client request and sends the response back to the server which is then forwarded to the client.

INTRODUCING THE SERVLET API

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

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

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

Interfaces in javax.servlet package (*in place of javax in the latest versions we need to use Jakarta***)**

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

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

Classes in javax.servlet package (*in place of javax in the latest versions we need to use Jakarta***)**

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

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

Interfaces in javax.servlet.http package (*in place of javax in the latest versions we need to use Jakarta*)

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

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

Classes in javax.servlet.http package (*in place of javax in the latest versions we need to use Jakarta*)

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

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

Servlet Interface

Servlet interface provides common behaviour to all the servlets. Servlet interface defines methods that all servlets must implement.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

Methods of Servlet interface

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

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

Servlet Example by implementing Servlet interface

First.java

```
import java.io.*;
import javax.servlet.*;
public class First implements Servlet{
ServletConfig config=null;

public void init(ServletConfig config){
this.config=config;
System.out.println("servlet is initialized");
}

public void service(ServletRequest req,ServletResponse res)
throws IOException,ServletException{

res.setContentType("text/html");

PrintWriter out=res.getWriter();
out.print("<html><body>");
```

```
out.print("<b>hello simple servlet</b>");
out.print("</body></html>");
}
public void destroy()
{
System.out.println("servlet is destroyed");
}
public ServletConfig getServletConfig()
{
return config;
}
public String getServletInfo()
{
return "Servlet Info";
}
 }
```

GenericServlet class

GenericServlet class implements **Servlet**, **ServletConfig** and **Serializable** interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.

You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

Methods of GenericServlet class

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

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

9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
11. **public void log(String msg)** writes the given message in the servlet log file.
12. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

Servlet Example by inheriting the GenericServlet class

Example of servlet by inheriting the GenericServlet class.

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

public class First extends GenericServlet{
public void service(ServletRequest req,ServletResponse res)
throws IOException,ServletException{

res.setContentType("text/html");

PrintWriter out=res.getWriter();
out.print("<html><body>");
out.print("<b>hello generic servlet</b>");
out.print("</body></html>");
}
}
```

HttpServlet class

The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

Methods of HttpServlet class

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

1. **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
2. **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.

3. **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
4. **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
5. **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
6. **protected void doOptions(HttpServletRequest req, HttpServletResponse res)**handles the OPTIONS request. It is invoked by the web container.
7. **protected void doPut(HttpServletRequest req, HttpServletResponse res)** handles the PUT request. It is invoked by the web container.
8. **protected void doTrace(HttpServletRequest req, HttpServletResponse res)** handles the TRACE request. It is invoked by the web container.
9. **protected void doDelete(HttpServletRequest req, HttpServletResponse res)**handles the DELETE request. It is invoked by the web container.
10. **protected long getLastModified(HttpServletRequest req)** returns the time when HttpServletRequest was last modified since midnight January 1, 1970 GMT.

LIFE CYCLE OF A SERVLET (SERVLET LIFE CYCLE)

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

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

1) Servlet class is loaded

The servlet container loads the servlet class into memory. This will happen when the container starts (if the servlet is configured to load on startup), or it can be triggered by the first request made to the servlet (lazy loading).

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. This method is used for **initialization** tasks, such as opening database connections, loading configuration files, or performing other one-time setup

tasks. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

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

4) service method is invoked

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

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

5) destroy method is invoked

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

public void destroy()

The HttpServlet class is widely used to create the servlet because it provides methods to handle http requests such as doGet(), doPost, doHead() etc.

In this example we are going to create a servlet that extends the HttpServlet class. In this example, we are inheriting the HttpServlet class and providing the implementation of the doGet() method.

DemoServlet.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");//setting the content type
PrintWriter pw=res.getWriter();//get the stream to write the data
//writing html in the stream
pw.println("<html><body>");
pw.println("Welcome to servlet");
```

```
pw.println("</body></html>");
pw.close();//closing the stream
}
}
```

Deployment descriptor (web.xml file)

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

The web container uses the Parser to get the information from the web.xml file. There are many xml parsers such as SAX, DOM and Pull.

There are many elements in the web.xml file. Here is given some necessary elements to run the simple servlet program.

web.xml file

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

Description of the elements of web.xml file

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

<web-app> represents the whole application.

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

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

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

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

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

invoke the servlet.

welcome-file-list in web.xml

The **welcome-file-list** element of **web-app**, is used to define a list of welcome files. Its sub element is **welcome-file** that is used to define the welcome file.

A **welcome file** is the file that is invoked automatically by the server, if you don't specify any file name.

By default server looks for the welcome file in following order:

1. welcome-file-list in web.xml
2. index.html
3. index.htm
4. index.jsp

If none of these files are found, server renders 404 error.

If you have specified welcome-file in web.xml, and all the files index.html, index.htm and index.jsp exists, priority goes to welcome-file.

If welcome-file-list entry doesn't exist in web.xml file, priority goes to index.html file then index.htm and at last index.jsp file.

The following is the web.xml file that defines the welcome files.

```
<web-app>
 ....

  <welcome-file-list>
   <welcome-file>home.html</welcome-file>
   <welcome-file>default.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Now, home.html and default.html will be the welcome files.

ServletRequest Interface

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

Methods of ServletRequest interface

There are many methods defined in the ServletRequest interface. Some of them are as follows:

Method	Description
public String getParameter(String name)	is used to obtain the value of a parameter by name.
public String[] getParameterValues(String name)	returns an array of String containing all values of given parameter name. It is mainly used to obtain values of a Multi select list box.
java.util.Enumeration getParameterNames()	returns an enumeration of all of the request parameter names.
public int getContentLength()	Returns the size of the request entity data, or -1 if not known.
public String getCharacterEncoding()	Returns the character set encoding for the input of this request.
public String getContentType()	Returns the Internet Media Type of the request entity data, or null if not known.
public ServletInputStream getInputStream() throws IOException	Returns an input stream for reading binary data in the request body.
public abstract String getServerName()	Returns the host name of the server that received the request.
public int getServerPort()	Returns the port number on which this request was received.

Example of ServletRequest to display the name of the user

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

index.html

```
<form action="welcome" method="get">
Enter your name<input type="text" name="name"><br>
<input type="submit" value="login">
</form>
```

DemoServ.java

```
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
public class DemoServ extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();

String name=req.getParameter("name");//will return value
pw.println("Welcome "+name);

pw.close();
}}
```

DESCRIBING THE HTTPSERVLETREQUEST AND HTTPSERVLETRESPONSE INTERFACES

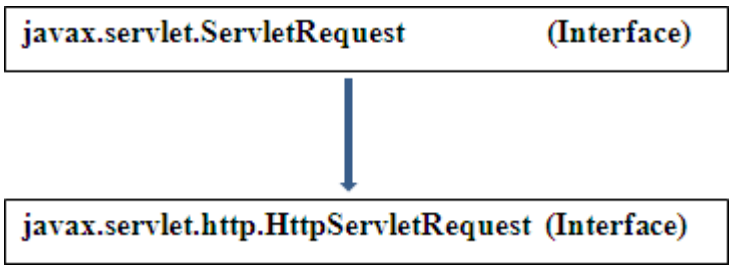
Both HttpServletRequest and HttpServletResponse are inevitable in each Servlet, the Programmer write. Both exist in Servlet code as parameters to service() method. Even a Novice should be acquainted of these two interfaces.

List of Differences HttpServletRequest vs HttpServletResponse

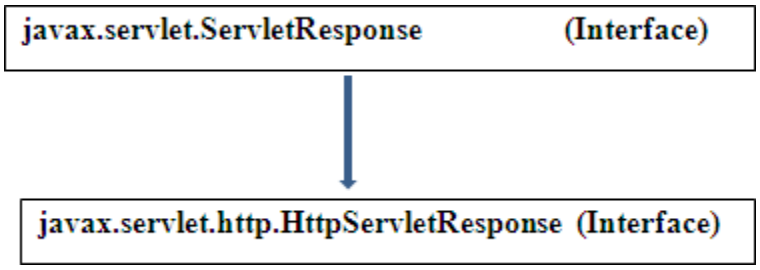
1. Both are interfaces from **javax.servlet.http** package.
2. They are derived from ServletRequest and ServletResponse interfaces.
3. Both objects are created and implicitly passed to the service() method of Servlet by container.

Figures below shows the hierarchy of interfaces.

A) Hierarchy of HttpServletRequest



B) Hierarchy of HttpServletResponse



- The job of **HttpServletRequest** is to **receive data** sent by Web client, like user name and password.
- Also comes with many **getXXX()** methods to retrieve other client information like client's IP address, what protocol client used etc. and also include methods to know about the client's browser like what is the name of the browser, its version etc.
- The job of **HttpServletResponse** is to send data to Web client.
- Also comes with many **setXXX()** methods to set properties to a Servlet.
- These being interfaces, the implementation of the abstract methods is to be developed by the Web server (like **Tomcat**) developers. Even the servlet container is to be developed by the Web server developers.

RequestDispatcher in Servlet

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

There are two methods defined in the RequestDispatcher interface.

Methods of RequestDispatcher interface

The RequestDispatcher interface provides two methods. They are:

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

How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher. Syntax:

Syntax of getRequestDispatcher method

public RequestDispatcher getRequestDispatcher(String resource);

Example:

```
RequestDispatcher rd=request.getRequestDispatcher("servlet2");
rd.forward(request, response);//method may be include or forward
```

index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
Password:<input type="password" name="userPass"/><br/>
<input type="submit" value="login"/>
</form>
```

Login.java

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

public class Login extends HttpServlet {

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

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

         if(p.equals("servlet")){
        RequestDispatcher rd=request.getRequestDispatcher("servlet2");
        rd.forward(request, response);
        }
        else{
        out.print("Sorry UserName or Password Error!");
        RequestDispatcher rd=request.getRequestDispatcher("/index.html");
```

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

            }
           }
           }
```

WelcomeServlet.java

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

public class WelcomeServlet extends HttpServlet {

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

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   String n=request.getParameter("userName");
   out.print("Welcome "+n);
   }
}
```

Web.xml

```
<web-app>
 <servlet>
   <servlet-name>Login</servlet-name>
   <servlet-class>Login</servlet-class>
  </servlet>
  <servlet>
   <servlet-name>WelcomeServlet</servlet-name>
   <servlet-class>WelcomeServlet</servlet-class>
  </servlet>

  <servlet-mapping>
   <servlet-name>Login</servlet-name>
   <url-pattern>/servlet1</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
   <servlet-name>WelcomeServlet</servlet-name>
   <url-pattern>/servlet2</url-pattern>
  </servlet-mapping>
```

```
  <welcome-file-list>
   <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

SendRedirect in servlet

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

It accepts relative as well as absolute URL.

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

Syntax of sendRedirect() method
public void sendRedirect(String URL)**throws** IOException;

Example of sendRedirect() method
response.sendRedirect("http://www.abc.com");

Example of sendRedirect method in servlet

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

public class DemoServlet extends HttpServlet{
public void doGet(HttpServletRequest req,HttpServletResponse res)
throws ServletException,IOException
{
res.setContentType("text/html");
PrintWriter pw=res.getWriter();

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

pw.close();
}}
```

WORKING WITH INITIALIZATION PARAMETERS:

ServletConfig Interface

An object of ServletConfig is created by the web container for each servlet. This object can be used to get configuration information from web.xml file.

If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.

Advantage of ServletConfig

The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

Methods of ServletConfig interface

1. **public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.
2. **public Enumeration getInitParameterNames():**Returns an enumeration of all the initialization parameter names.
3. **public String getServletName():**Returns the name of the servlet.
4. **public ServletContext getServletContext():**Returns an object of ServletContext.

How to get the object of ServletConfig

getServletConfig() method of Servlet interface returns the object of ServletConfig.

Syntax to provide the initialization parameter for a servlet

The init-param sub-element of servlet is used to specify the initialization parameter for a servlet.

```
<web-app>
  <servlet>
    ......

    <init-param>
      <param-name>parametername</param-name>
      <param-value>parametervalue</param-value>
    </init-param>
    ......
  </servlet>
</web-app>
```

Example of ServletConfig to get initialization parameter

In this example, we are getting the one initialization parameter from the web.xml file and printing this information in the servlet.

DemoServlet.java

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

public class DemoServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   ServletConfig config=getServletConfig();
   String driver=config.getInitParameter("driver");
   out.print("Driver is: "+driver);
   out.close();
   }
  }
```

web.xml

```
<web-app>

<servlet>
<servlet-name>DemoServlet</servlet-name>
<servlet-class>DemoServlet</servlet-class>

<init-param>
<param-name>driver</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>

</servlet>

<servlet-mapping>
<servlet-name>DemoServlet</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

</web-app>
```

Session Tracking in Servlets

Session means a particular interval of time.

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

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

HTTP is stateless that means each request is considered as the new request.

Why use Session Tracking?

It is used to recognize the particular user.

Session Tracking Techniques

There are four techniques used in Session tracking:

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

Cookies in Servlet

A **cookie** is a small piece of information that is persisted among the multiple client requests.

A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

How Cookie works

By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.

Advantage of Cookies

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

Disadvantage of Cookies

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

Cookie class

javax.servlet.http.Cookie class provides the functionality of using cookies. It provides a lot of useful methods for cookies.

Constructor of Cookie class

Constructor	Description
Cookie()	constructs a cookie.
Cookie(String name, String value)	constructs a cookie with a specified name and value.

Useful Methods of Cookie class

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):** method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[] getCookies():** method of HttpServletRequest interface is used to

return all the cookies from the browser.

Simple example of Servlet Cookies

index.html

```
<form action="servlet1" method="post">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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


public class FirstServlet extends HttpServlet {

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

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

    Cookie ck=new Cookie("uname",n);//creating cookie object
    response.addCookie(ck);//adding cookie in the response

    //creating submit button
    out.print("<form action='servlet2'>");
    out.print("<input type='submit' value='go'>");
    out.print("</form>");

    out.close();

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

```
    }
```

SecondServlet.java

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

public class SecondServlet extends HttpServlet {

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

   response.setContentType("text/html");
   PrintWriter out = response.getWriter();

   Cookie ck[]=request.getCookies();
   out.print("Hello "+ck[0].getValue());

   out.close();

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

web.xml

```
<web-app>
<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
```

```
</servlet-mapping>
</web-app>
```

HttpSession interface

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

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

How to get the HttpSession object ?

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

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

Commonly used methods of HttpSession interface

1. **public String getId():**Returns a string containing the unique identifier value.
2. **public long getCreationTime():**Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
3. **public long getLastAccessedTime():**Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
4. **public void invalidate():**Invalidates this session then unbinds any objects bound to it.

Example of using HttpSession

index.html

```
<form action="servlet1">
Name:<input type="text" name="userName"/><br/>
<input type="submit" value="go"/>
</form>
```

FirstServlet.java

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

```
public class FirstServlet extends HttpServlet {

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

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

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

      HttpSession session=request.getSession();
      session.setAttribute("uname",n);

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

      out.close();

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

}
```

SecondServlet.java

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

public class SecondServlet extends HttpServlet {

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

      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      HttpSession session=request.getSession(false);
      String n=(String)session.getAttribute("uname");
      out.print("Hello "+n);

      out.close();

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

```
}
```

web.xml

```
<web-app>

<servlet>
<servlet-name>s1</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s1</servlet-name>
<url-pattern>/servlet1</url-pattern>
</servlet-mapping>

<servlet>
<servlet-name>s2</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>s2</servlet-name>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>

</web-app>
```

SECURITY ISSUES

There are mainly two security issues in servlets. They are

1. It is not secure to send and receive data by using cookies. Since everyone can view and modify the data in cookies.
2. When we send the information to server by using the doGet() method, the user information is being visible in browser. So, it is not secure to use the doGet() method to send the information to server.

Unit – I (Part-2)

Web Servers and Servlets

Apache Tomcat Server & Its Installation steps:

Apache Tomcat is an open-source web server and servlet container for Java code. Tomcat executes programs written in the Java programming language, and it implements many Java EE specifications, including Jakarta Servlet, Jakarta Server Pages, and others.

Prerequisites:

- Java JRE installed and configured
- Administrator privileges

How to Install Tomcat on Windows

In this section, we will see installation of Apache Tomcat Server:

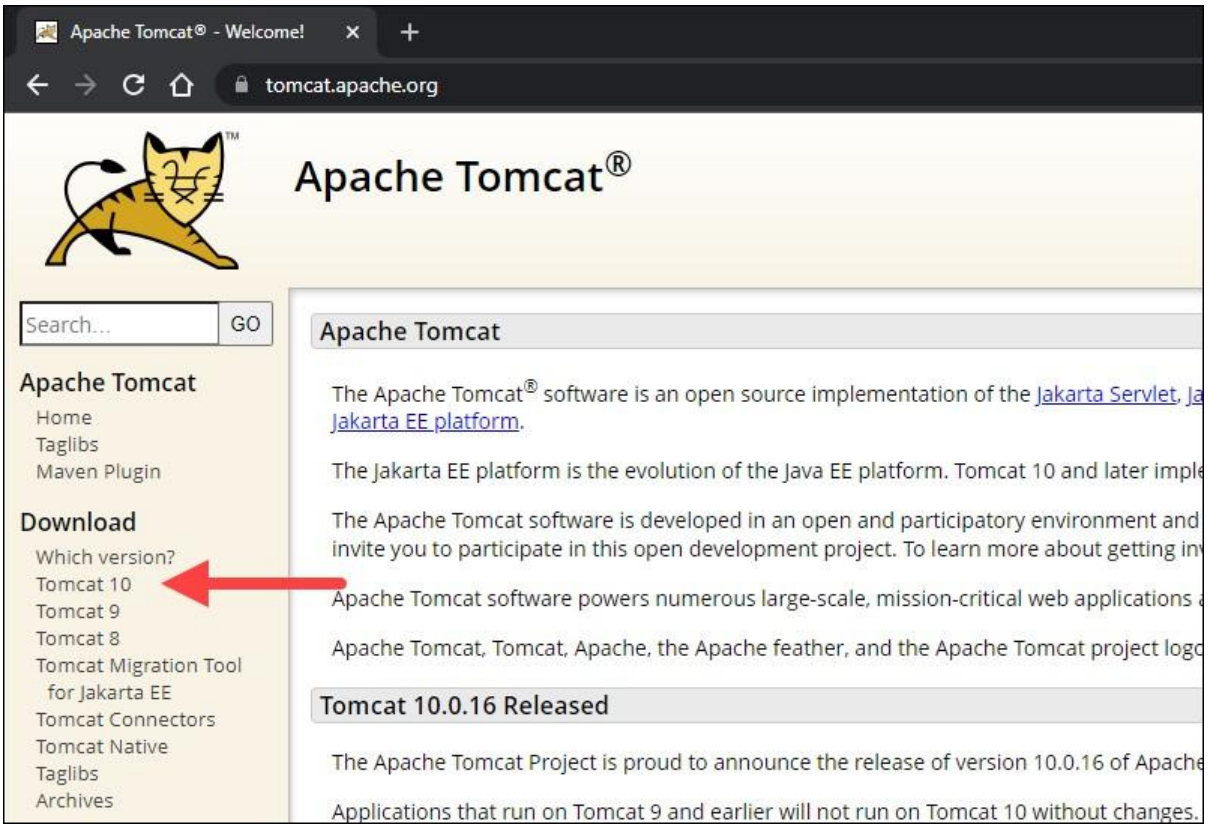
- Via Windows Service Installer.

Follow the steps below to download and install Tomcat.

Step 1: Download Tomcat for Windows

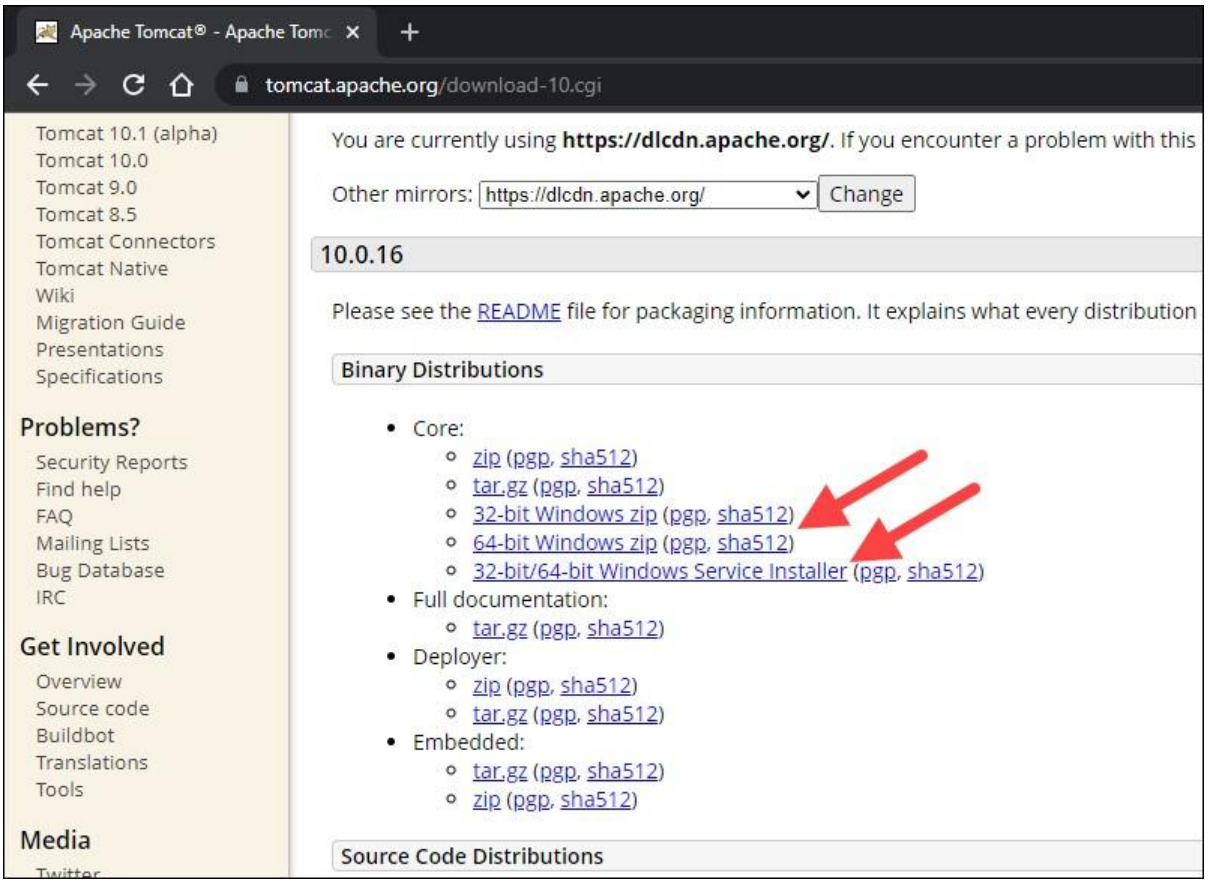
To download the Tomcat installation file, follow the steps below:

1. Browse to the official Apache Tomcat website. Locate the *Download* section and click the **latest Tomcat version** available.



2. On the *Download* page, scroll down and locate the *Binary Distributions* area.

In the *Core* list, depending on the installation type you prefer, click the download link for the **Windows Service Installer** or the **32bit/64bit Windows zip file**.



Step 2: Install Tomcat

Install Tomcat via the **Windows Service Installer** for an automated and wizard-guided experience. The service installer installs the Tomcat service and runs it automatically when the system boots.

For a portable experience, install Tomcat using the **zip file** and avoid installing the service. Easily uninstall Tomcat when it is no longer needed by deleting the Tomcat directory, or move it around when necessary.

Install Tomcat Using the Windows Service Installer

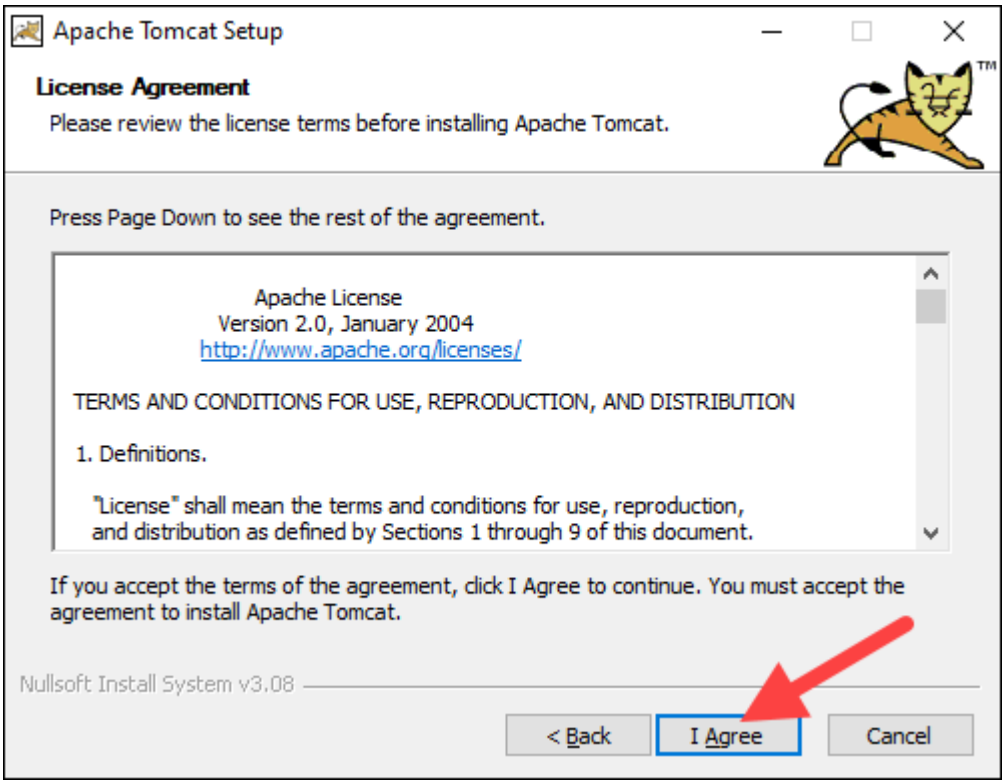
Follow the steps below to install Tomcat using the Windows Service Installer.

1. Open the downloaded **Windows Service Installer** file to start the installation process.

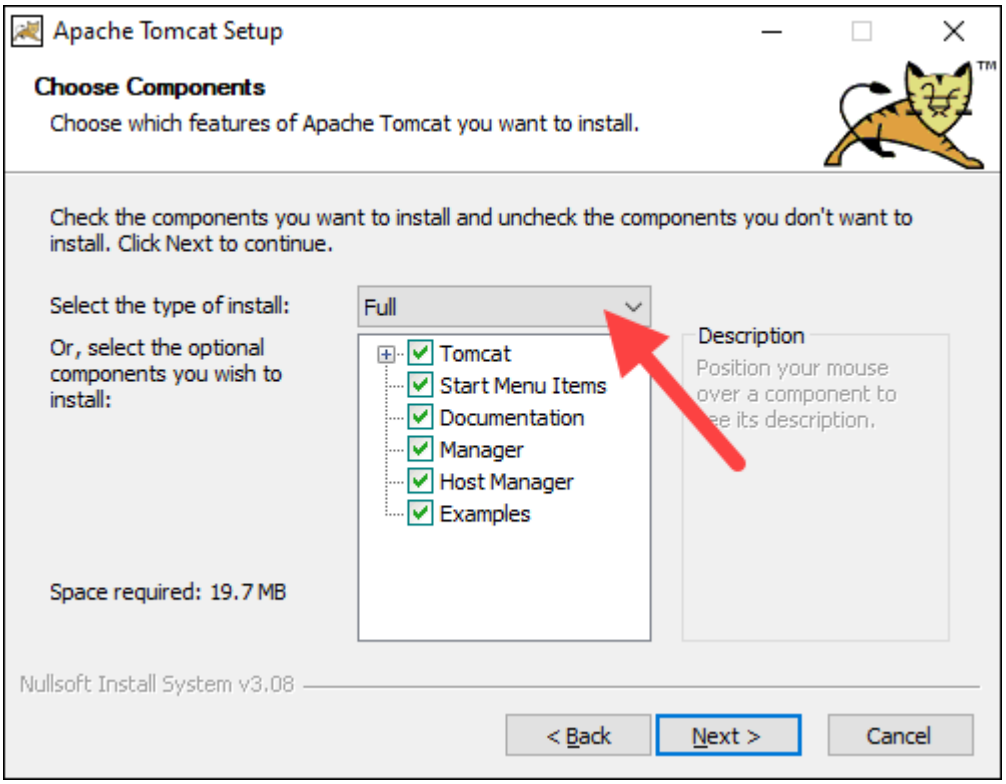
2. In the Tomcat Setup welcome screen, click **Next** to proceed.



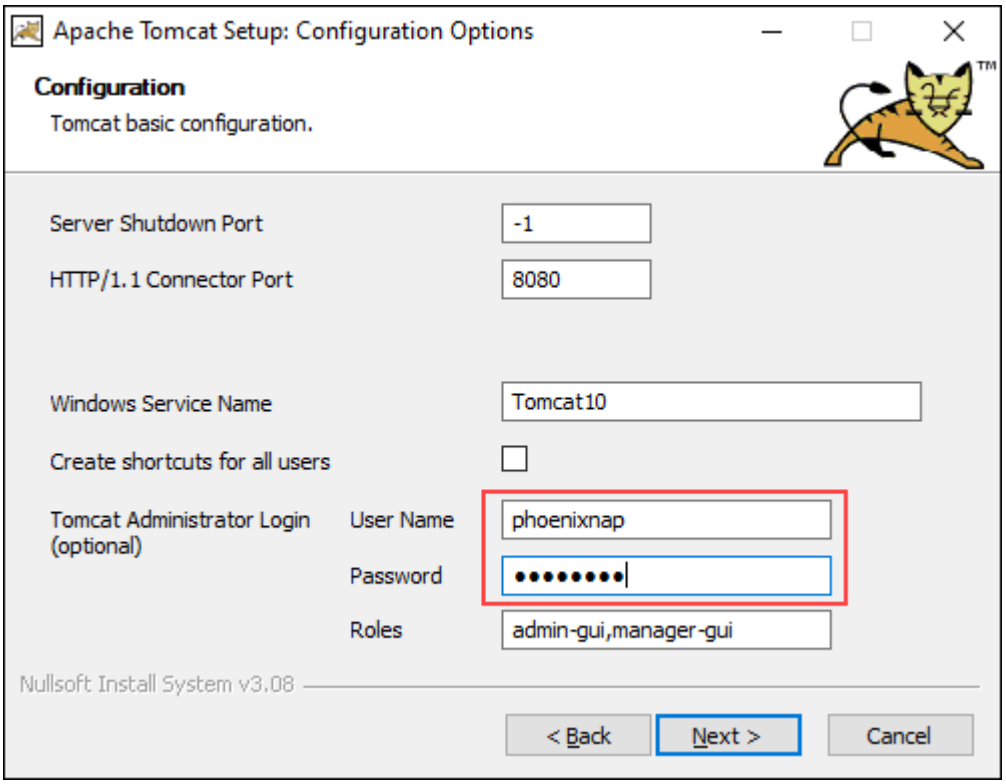
3. Read the License Agreement and if you agree to the terms, click **I Agree** to proceed to the next step.



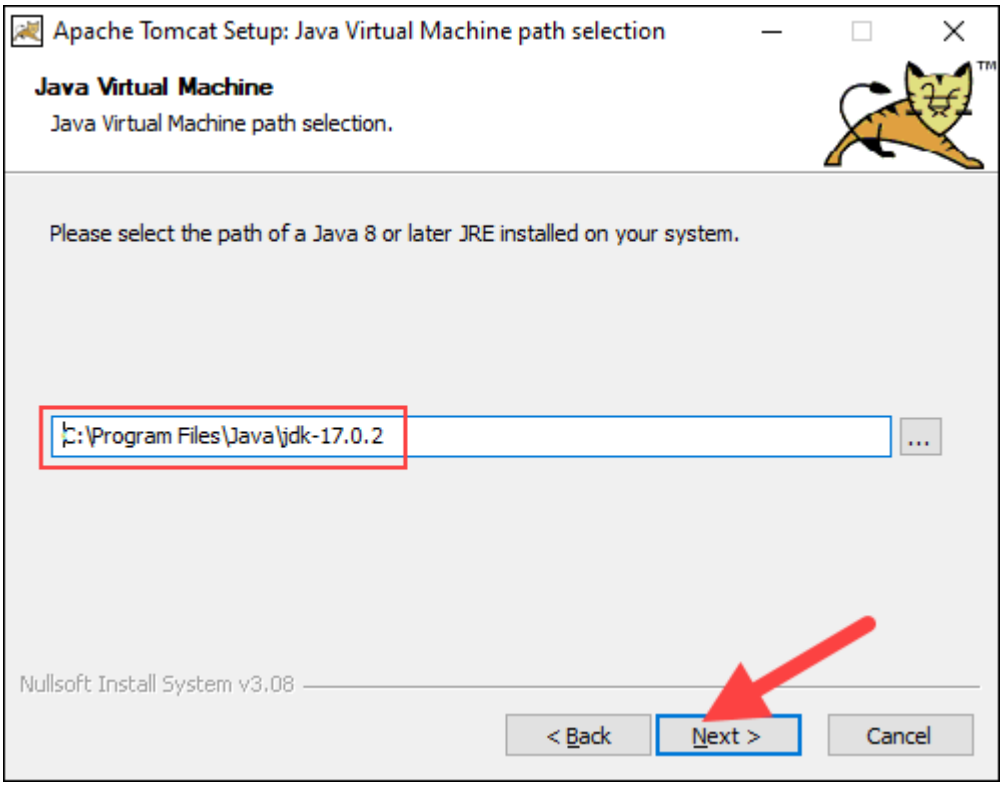
4. In the Tomcat component selection screen, choose **Full** in the dropdown menu to ensure the wizard installs the Tomcat Host Manager and Servlet and JSP examples web applications. Alternatively, keep the default **Normal** installation type and click **Next**.



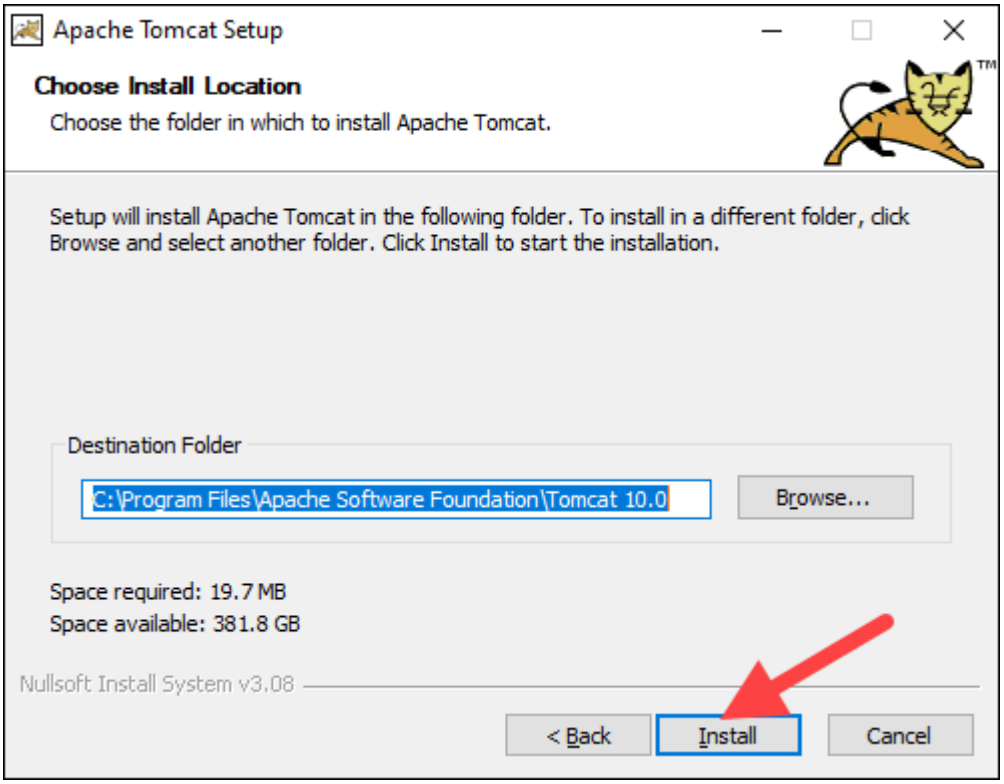
5. The next step configures the Tomcat server. For instance, enter the **Administrator login credentials** or choose a different **connection port**. When finished, click **Next** to proceed to the next step.



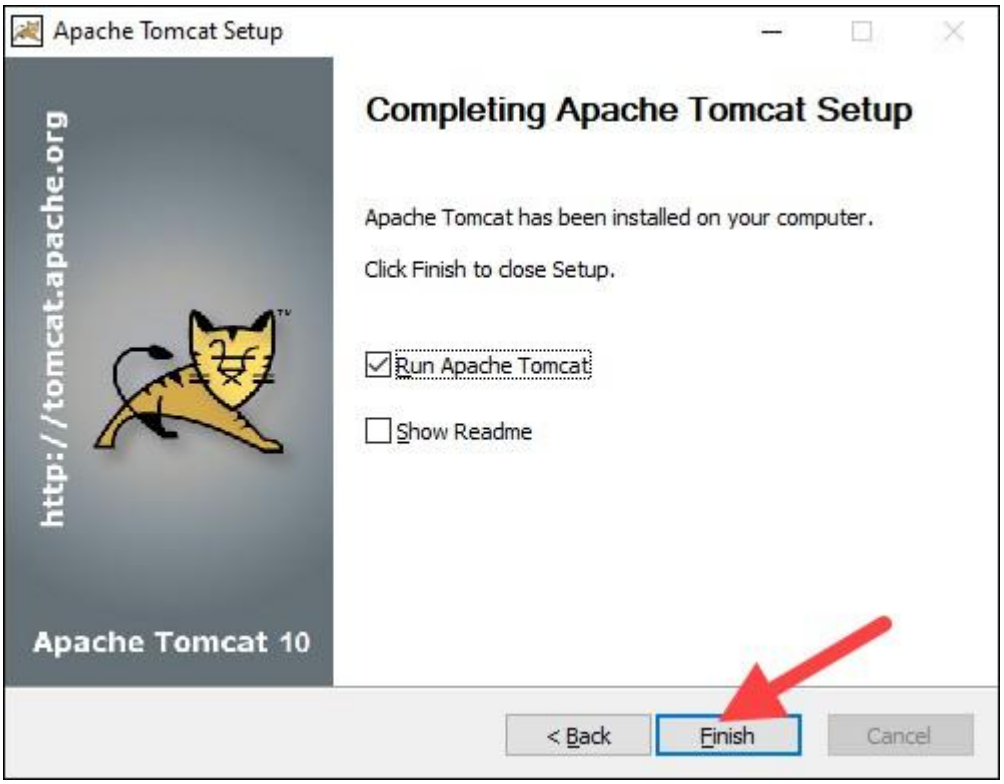
6. The next step requires you to enter the full path to the JRE directory on your system. The wizard auto-completes this if you have previously set up the Java environment variables. Click **Next** to proceed to the next step.



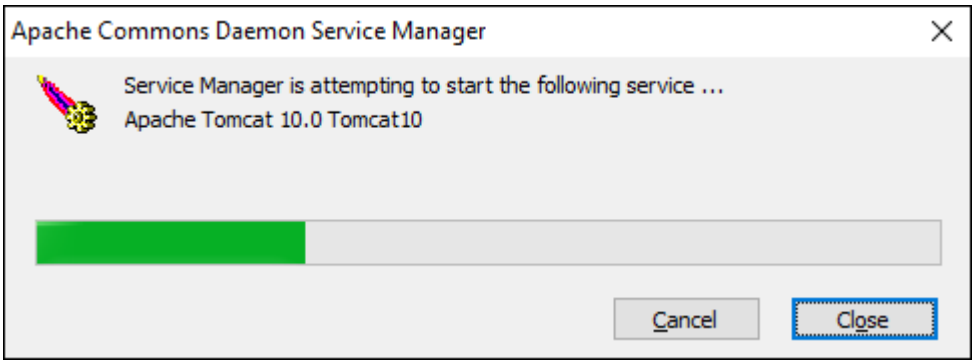
7. Choose the Tomcat server install location or keep the default one and click **Install**.



8. Check the **Run Apache Tomcat** box to start the service after the installation finishes. Optionally, check the **Show Readme** box to see the Readme file. To complete the installation, click **Finish**.

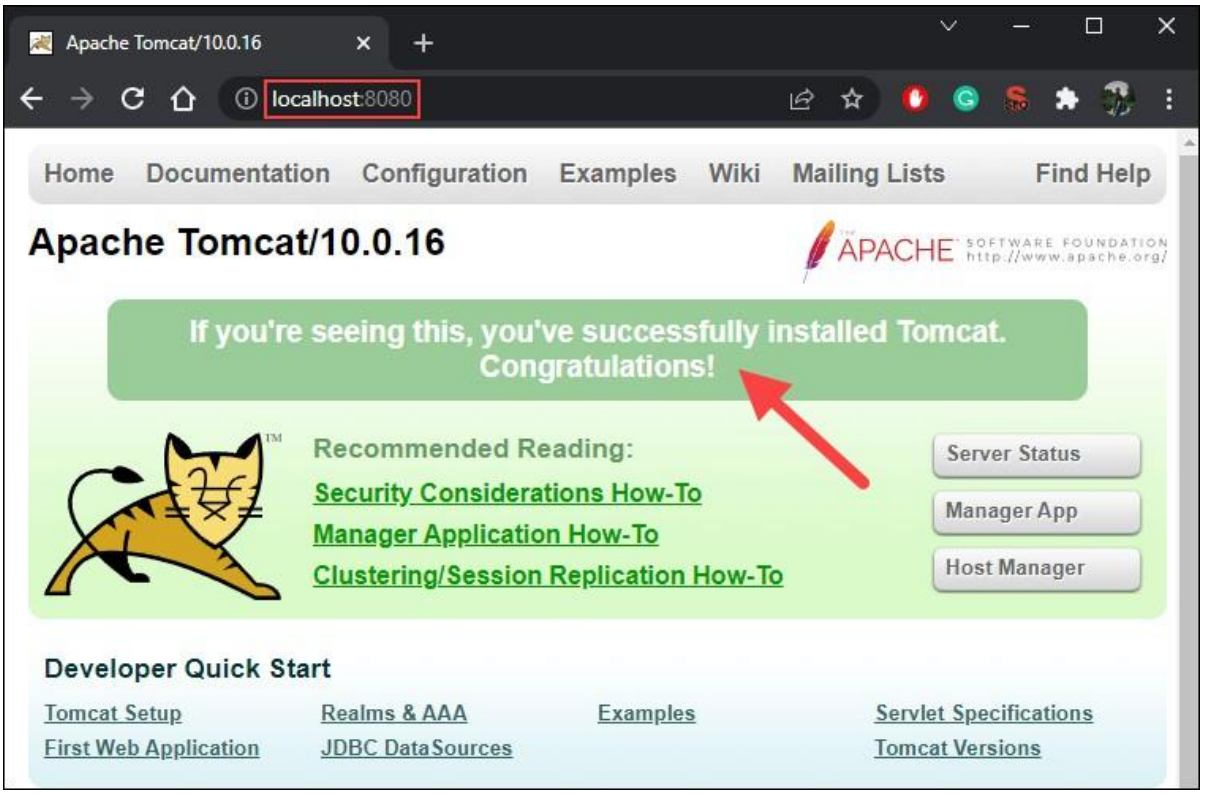


9. A popup window appears that starts the Tomcat service. After the process completes, the window closes automatically. The Apache Tomcat web server is now successfully installed .



Access the server using a browser as an HTTP client. Browse to http://localhost:8080 and access the Tomcat welcome page to ensure the server works.

In addition, use the *Developer Quick Start* links to see more information about the server and start using and configuring the server.



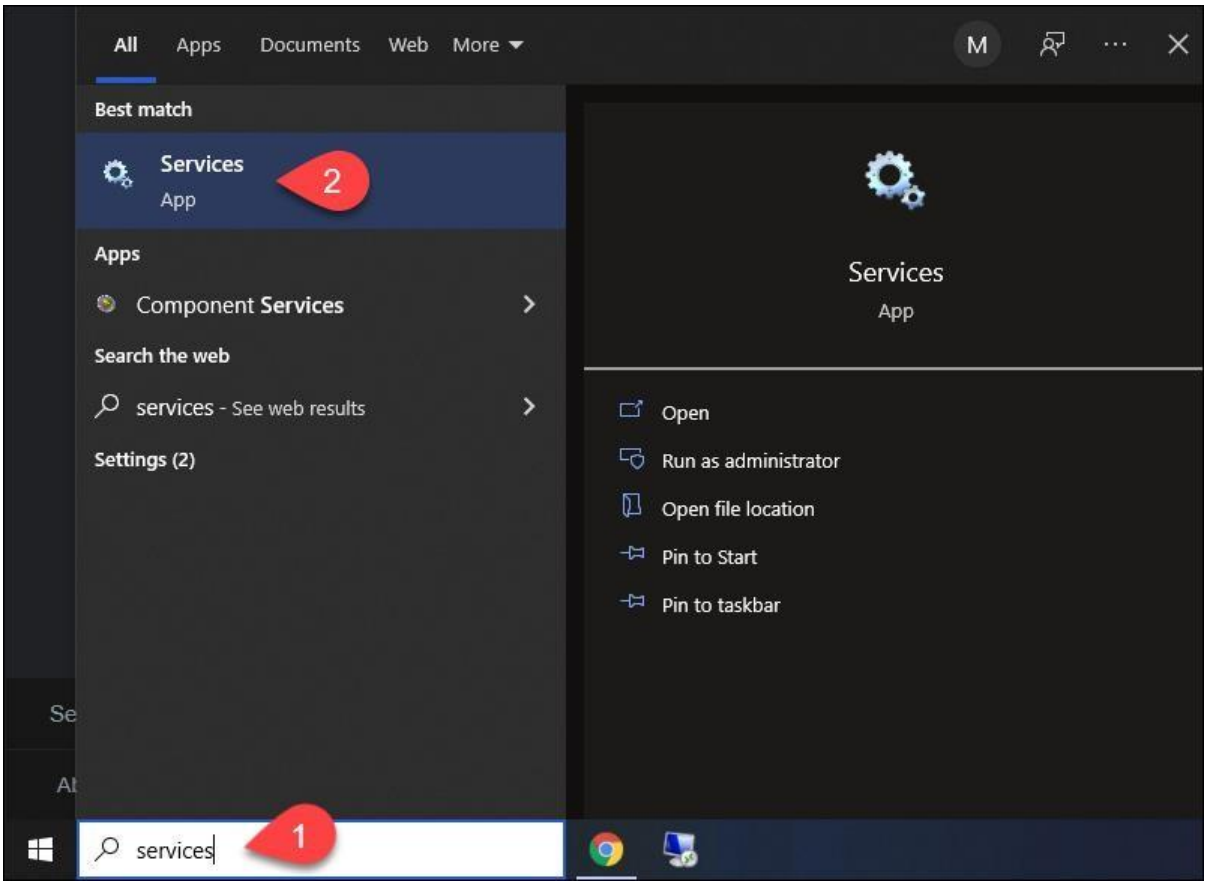
11. Shut down the Tomcat server by pressing **Ctrl+C** on the Tomcat console.

Step 3: Check if Apache Tomcat Service Is Running

Installing Tomcat using the Windows Service Installer installs Tomcat as a Windows service that automatically runs on boot. Follow the steps below to ensure that Tomcat is started as a Windows service.

1. Open the **Start** menu and search for *Services*.

2. Select the **Services** result.

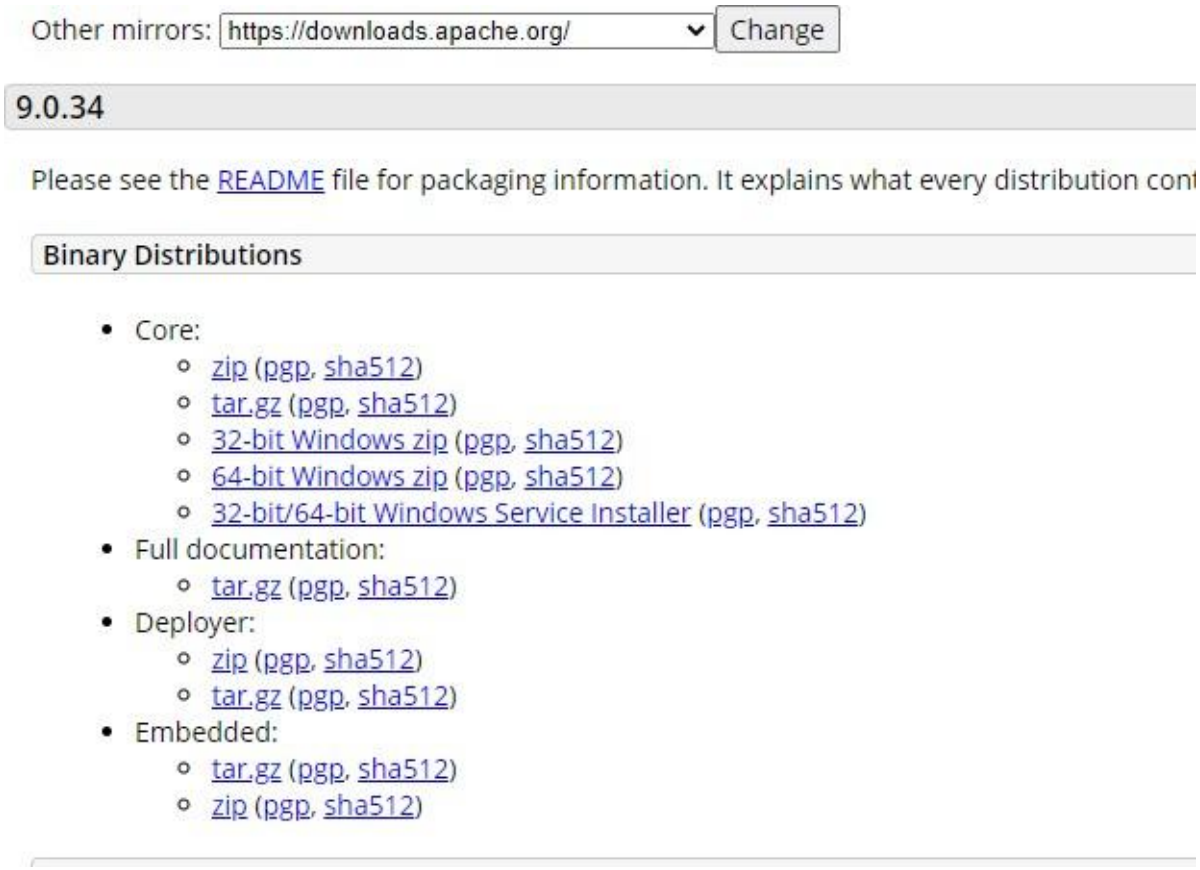


3. In the *Services* window, locate the *Apache Tomcat* service. The *Status* column indicates whether the service is running or not. **Start** or **Stop** the service using the buttons in the toolbar or by pressing **Stop** or **Restart** on the left side of the service list.

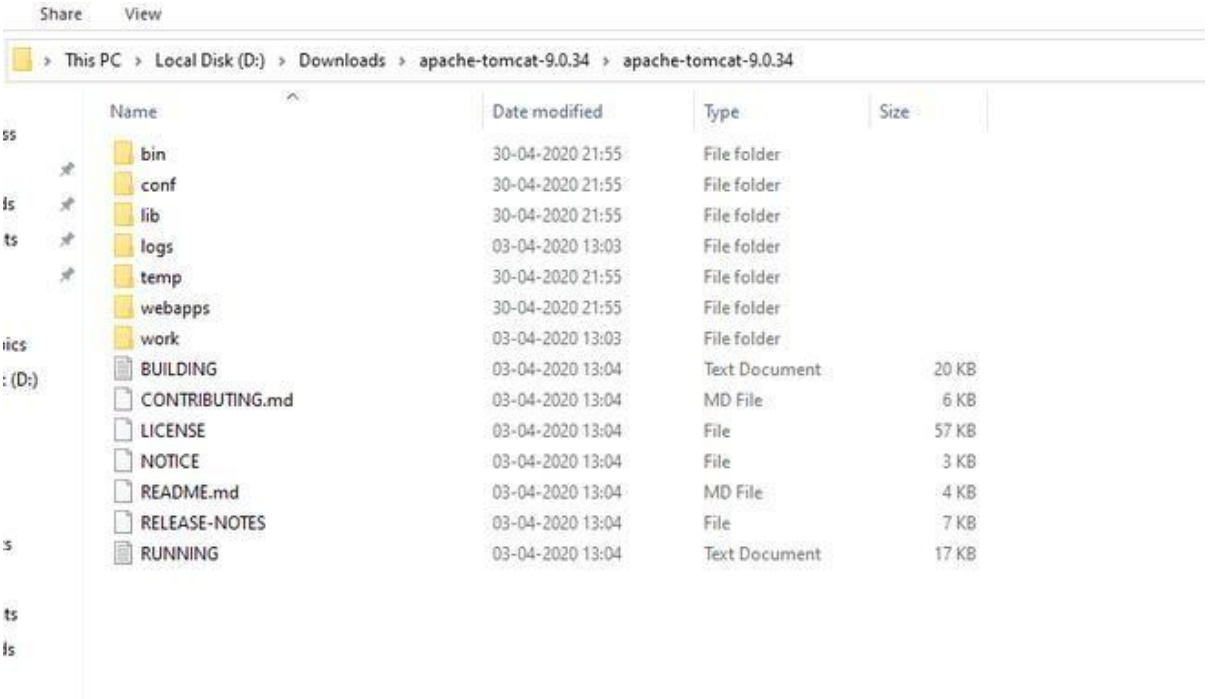
Configuration of Apache Tomcat Server with Eclipse IDE

In this section we will discuss a step by step guide to setup Apache Tomcat server in Eclipse IDE.

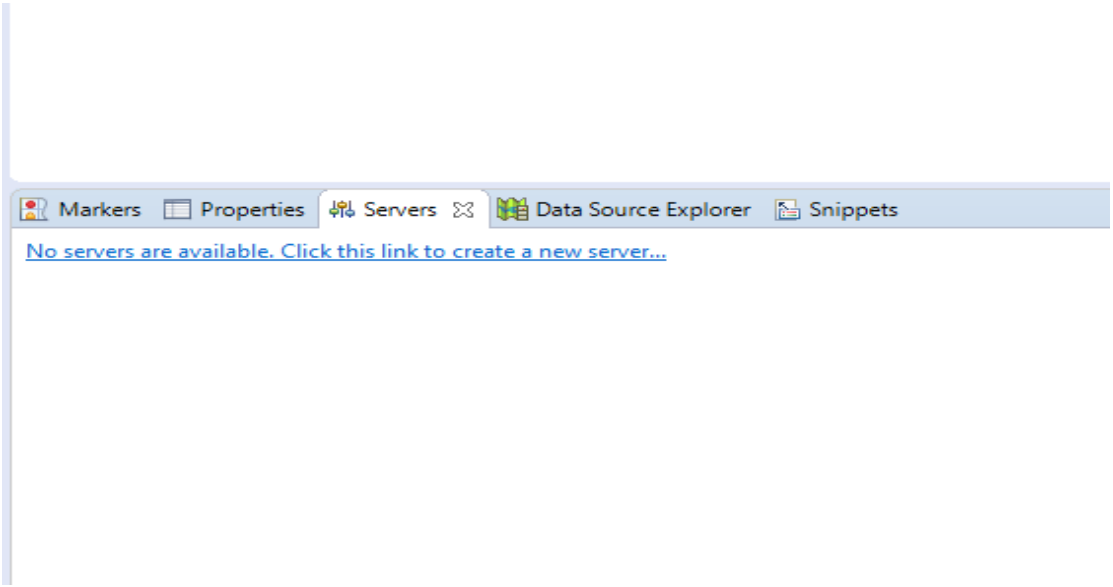
Step 1: Download the latest version (9.0.34) of Apache tomcat server according to your platform.



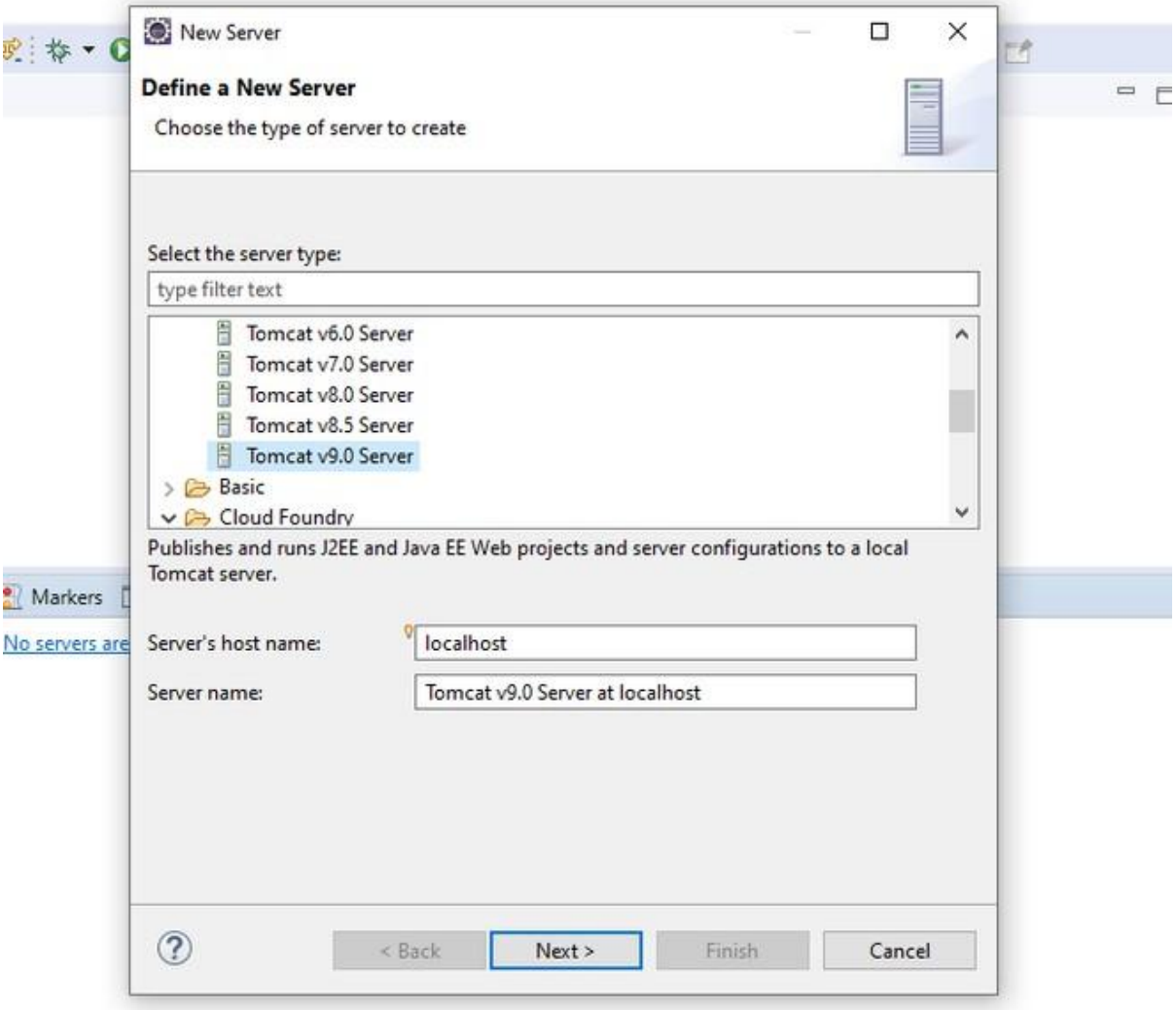
Step 2: Extract the files from zip folder. The structure of the folder should look like this:



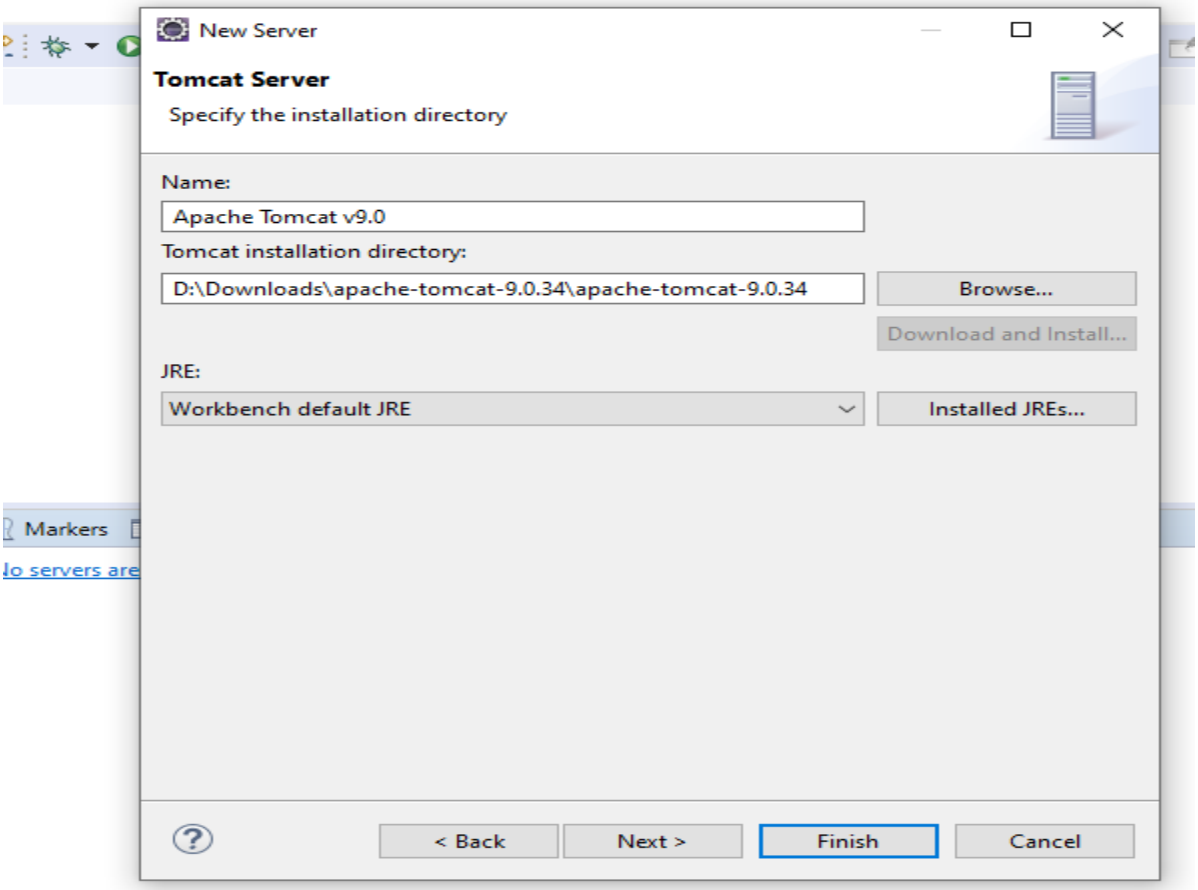
Step 3: Open Eclipse Java EE (Enterprise edition) environment. Click on *Servers* tab at bottom. Click on *No servers are available. Click this to create server.*



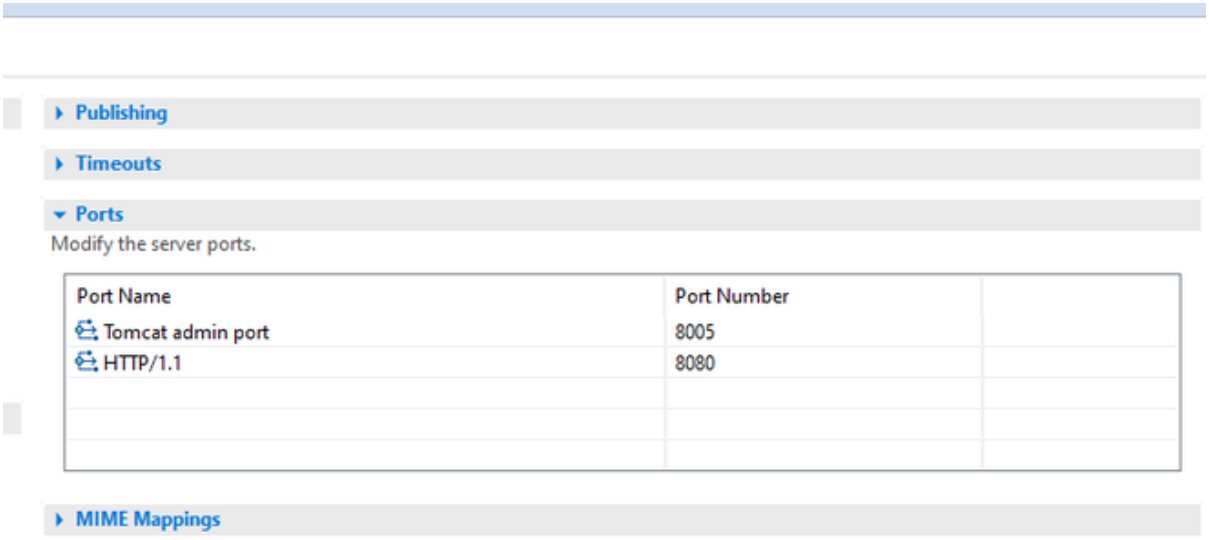
Step 4: A dialog box will appear. Select *Tomcat 9.0 server* folder. Click *Next.*



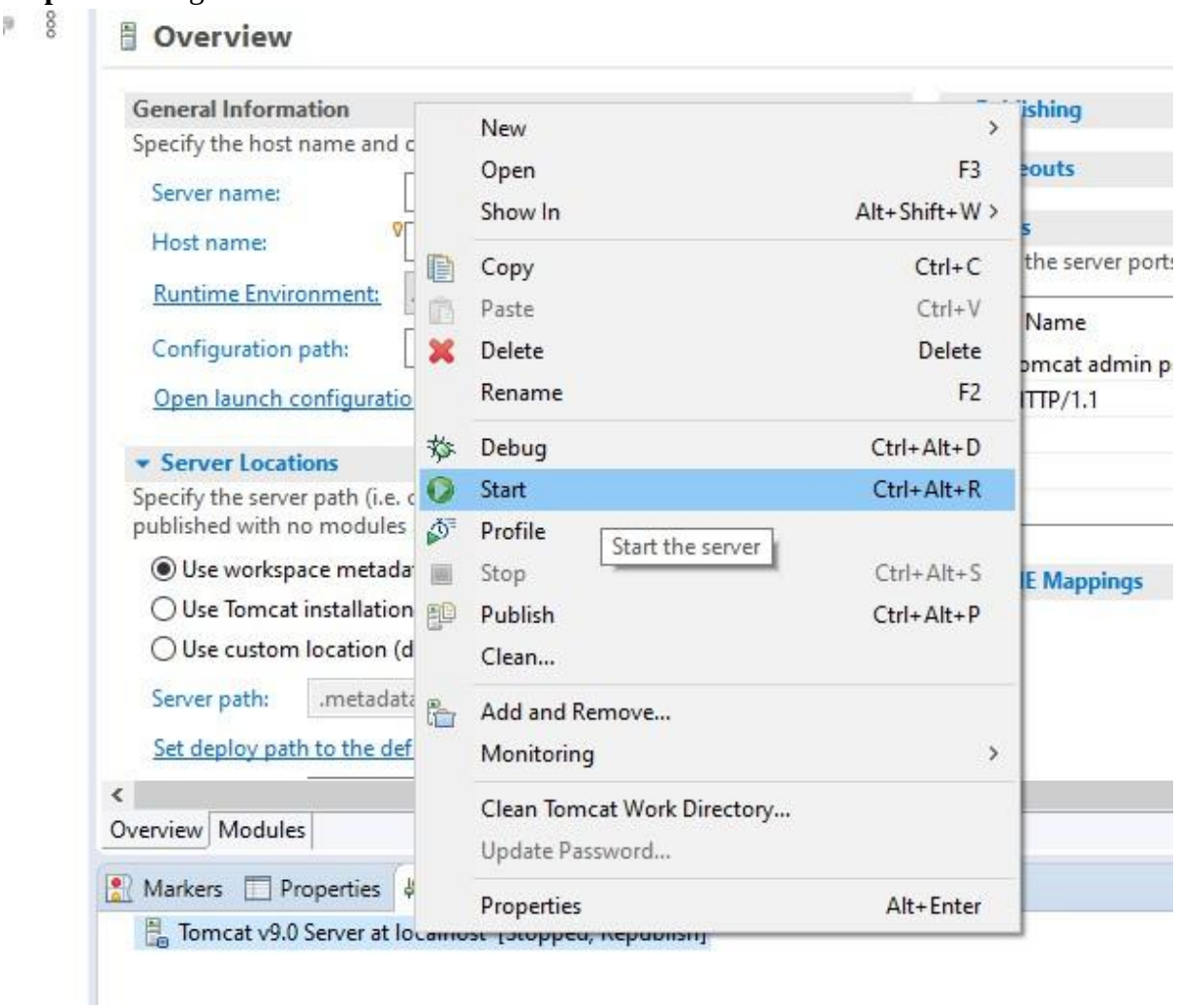
Step 5: Browse to *Apache Tomcat 9.0* folder select it. Click *Finish.*



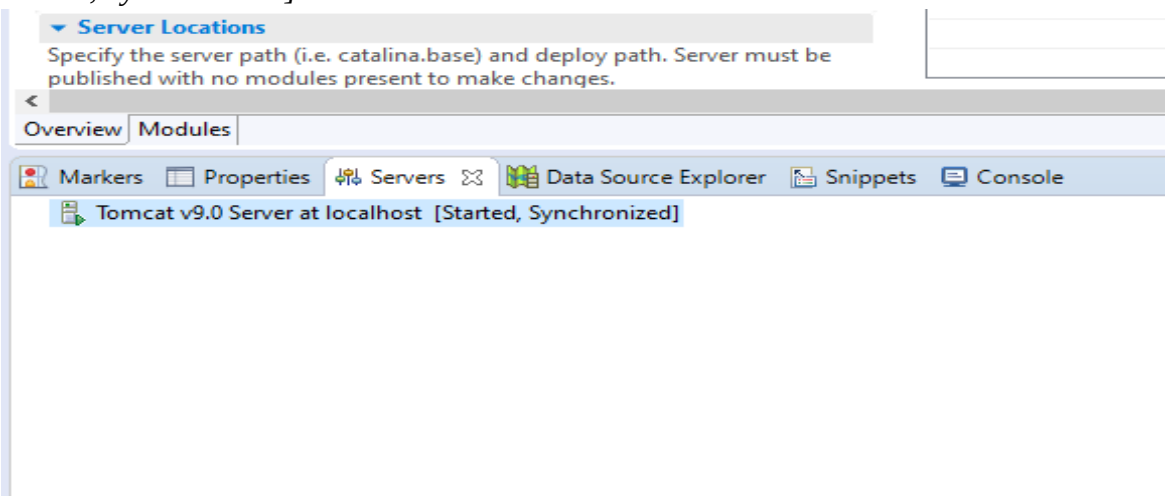
Step 6: You should see *Tomcat v9.0 Server at localhost [Stopped, Republish]* under *Servers* tab. Double click on it and verify the HTTP ports. By default HTTP/1.1 port is 8080. If there is any application running on default port 8080 then change it to any other port.



Step 7: Now right click on Server and click *Start.*



Step 8: The server will be started and shows the following *Tomcat v9.0 Server at localhost [Started, Synchronized]*.



The following server is running at port 8080 on localhost and we can visit the webpage using the URL *http://localhost:8080/*

Creating Servlet Example in Eclipse

Eclipse is an open-source ide for developing JavaSE and JavaEE (J2EE) applications. You can download the eclipse ide from the eclipse website http://www.eclipse.org/downloads/.

You need to download the eclipse ide for JavaEE developers.

Creating **servlet example in eclipse ide**, saves a lot of work to be done. It is easy and simple to create a servlet example. Let's see the steps, you need to follow to create the first servlet example.

- Create a Dynamic web project
- create a servlet
- add servlet-api.jar file
- Run the servlet

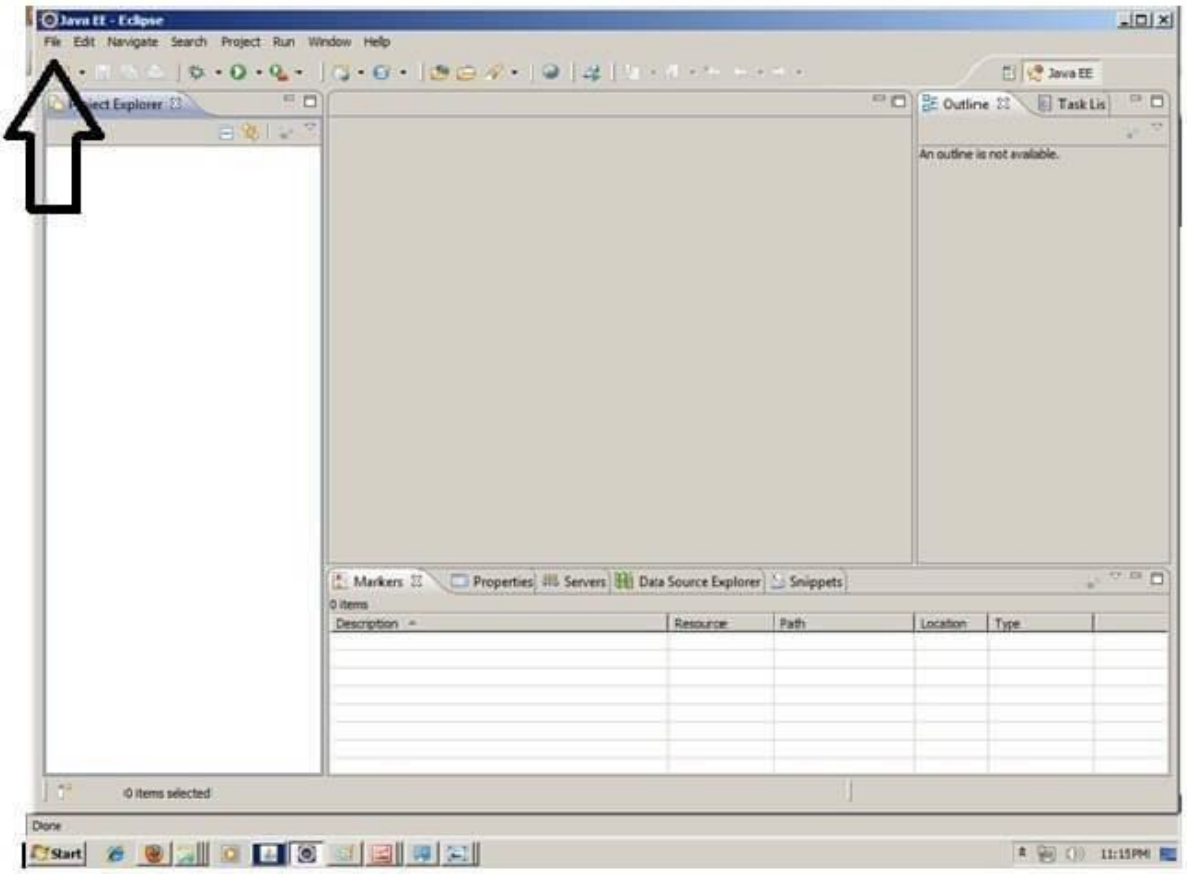
---

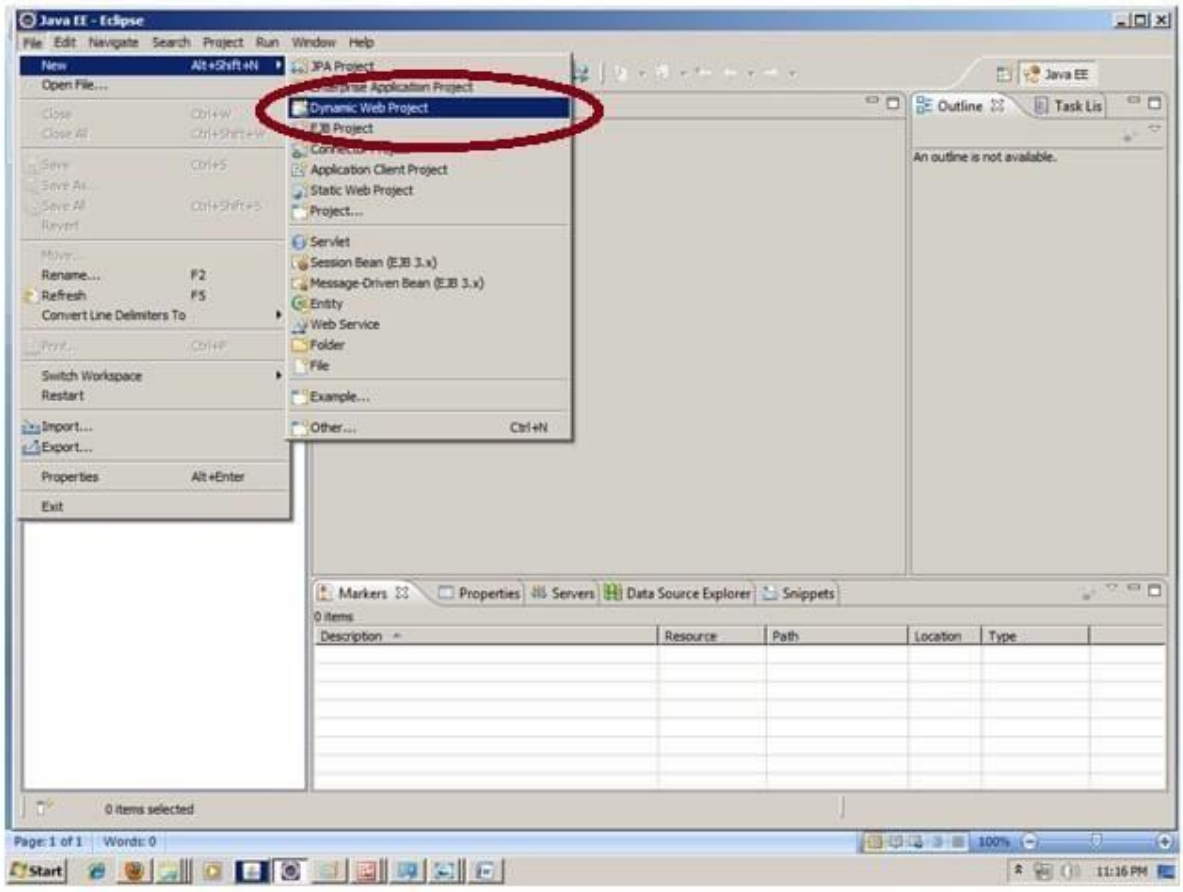
download this example (developed in eclipse)

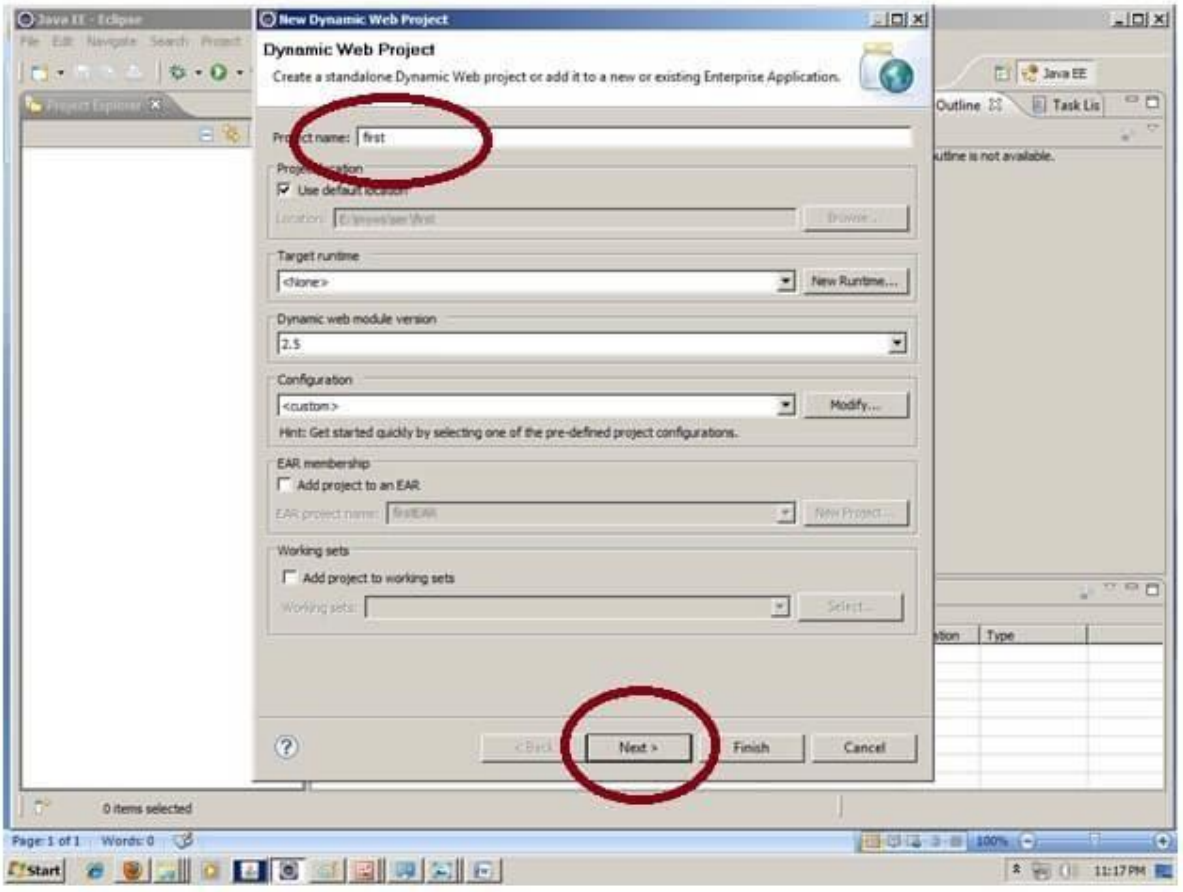
---

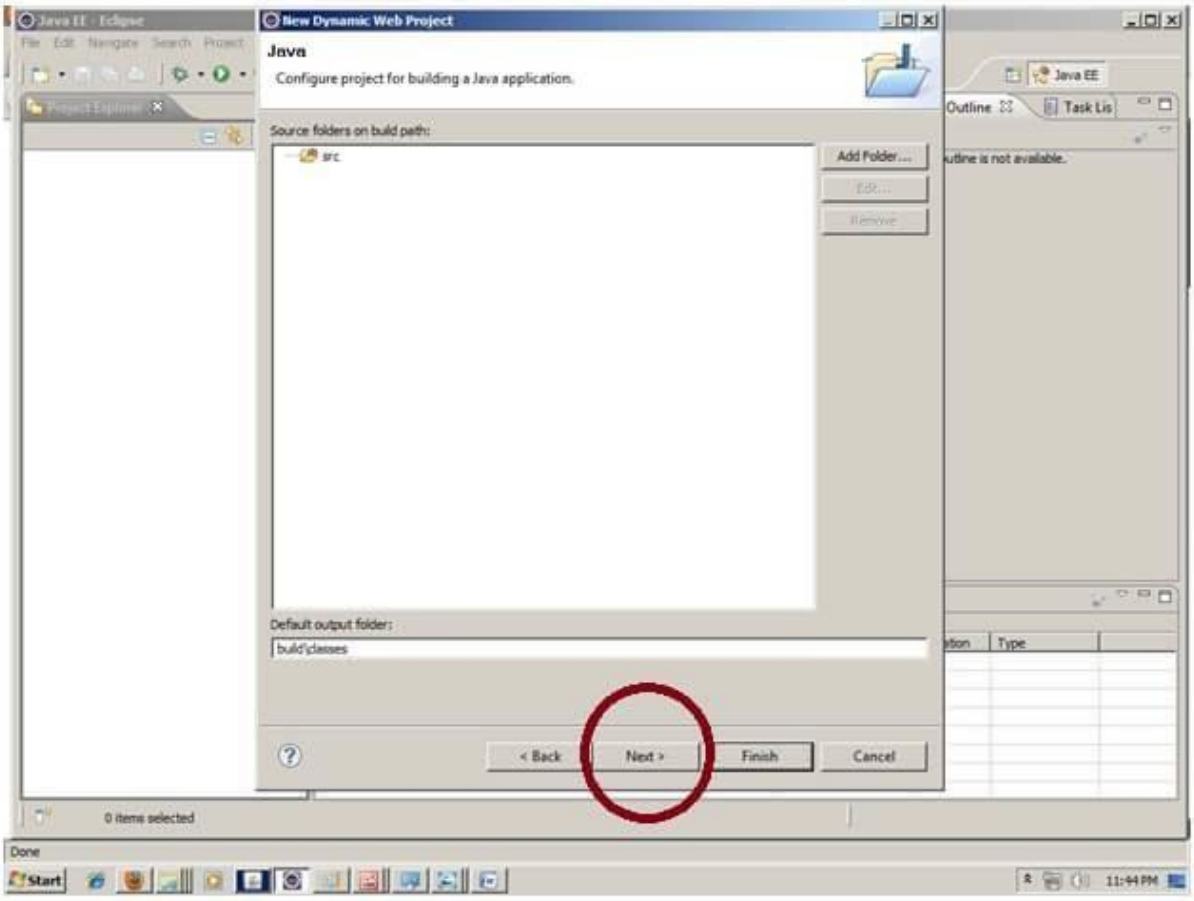
1) Create the dynamic web project:

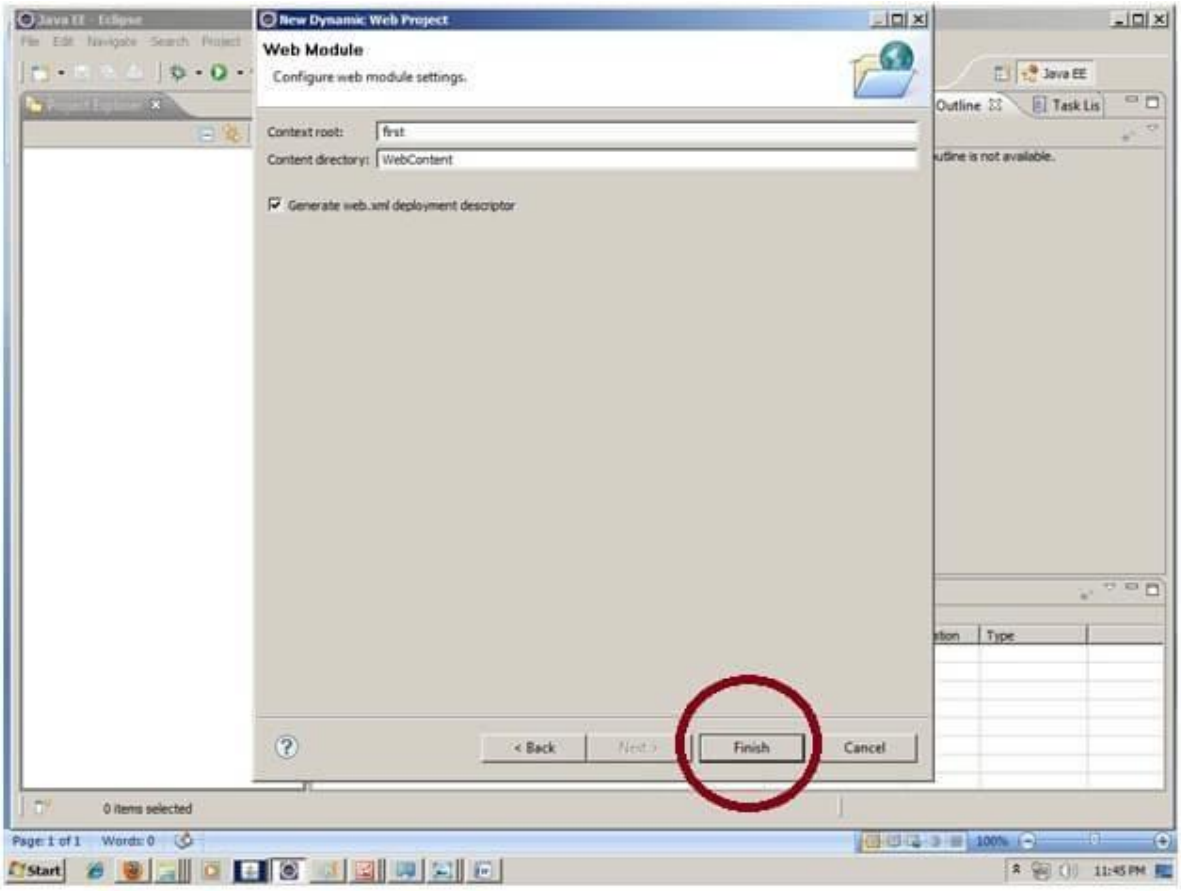
For creating a dynamic web project **click on File Menu -> New -> Project..-> Web -> dynamic web project -> write your project name e.g. first -> Finish**.

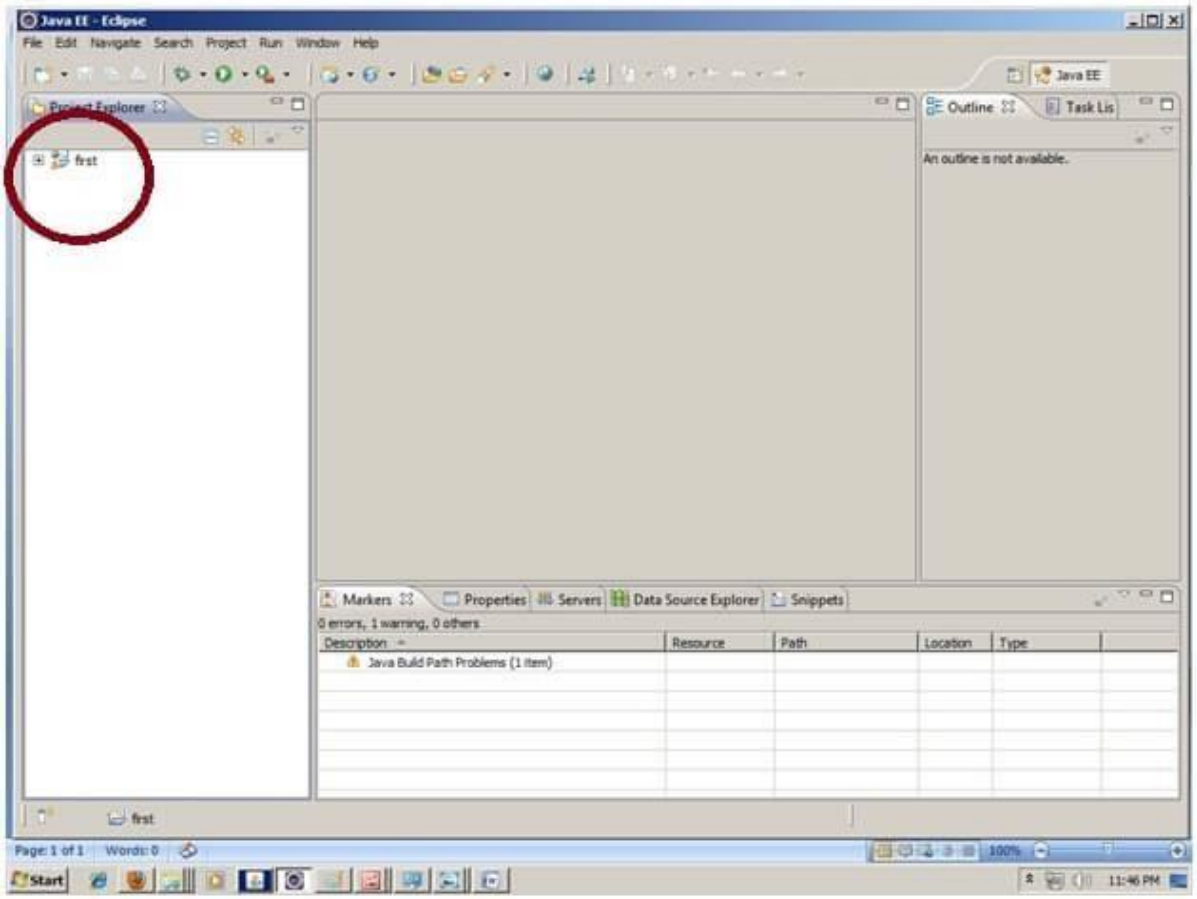



Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Close Ctrl+W
Close All Ctrl+Shift+W
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Revert
Move...
Rename... F2
Refresh F5
Convert Line Delimiters To
Print... Ctrl+P
Switch Workspace
Restart
Import...
Export...
Properties Alt+Enter
Exit
JPA Project
Enterprise Application Project
Dynamic Web Project
EJB Project
Connector Project
Application Client Project
Static Web Project
Project...
Servlet
Session Bean (EJB 3.x)
Message-Driven Bean (EJB 3.x)
Entity
Web Service
Folder
File
Example...
Other... Ctrl+N
Java EE
Outline
Task List
An outline is not available.
Markers
Properties
Servers
Data Source Explorer
Snippets
0 items
Description
Resource
Path
Location
Type
0 items selected
Page: 1 of 1
Words: 0
100%
Start
11:16 PM


New Dynamic Web Project
Dynamic Web Project
Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.
Project name: first
Use default location
Location:
Browse...
Target runtime
<none>
New Runtime...
Dynamic web module version
2.5
Configuration
<custom>
Modify...
Hint: Get started quickly by selecting one of the pre-defined project configurations.
EAR membership
Add project to an EAR
EAR project name:
New Project...
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Finish
Cancel
Java EE
Outline
Task List
0 items selected
Page: 1 of 1
Words: 0
100%
Start
11:17 PM


New Dynamic Web Project
Java
Configure project for building a Java application.
Source folders on build path:
src
Add Folder...
Edit...
Remove
Default output folder:
build\classes
< Back
Next >
Finish
Cancel
Java EE
Outline
Task Lis
utline is not available.
Type
0 items selected
Done
Start
11:44 PM


New Dynamic Web Project
Web Module
Configure web module settings.
Context root:
first
Content directory:
WebContent
Generate web.xml deployment descriptor
< Back
Next >
Finish
Cancel
Java EE
Outline
Task Lis
utline is not available.
Type
0 items selected
Page: 1 of 1
Words: 0
100%
Start
11:45 PM

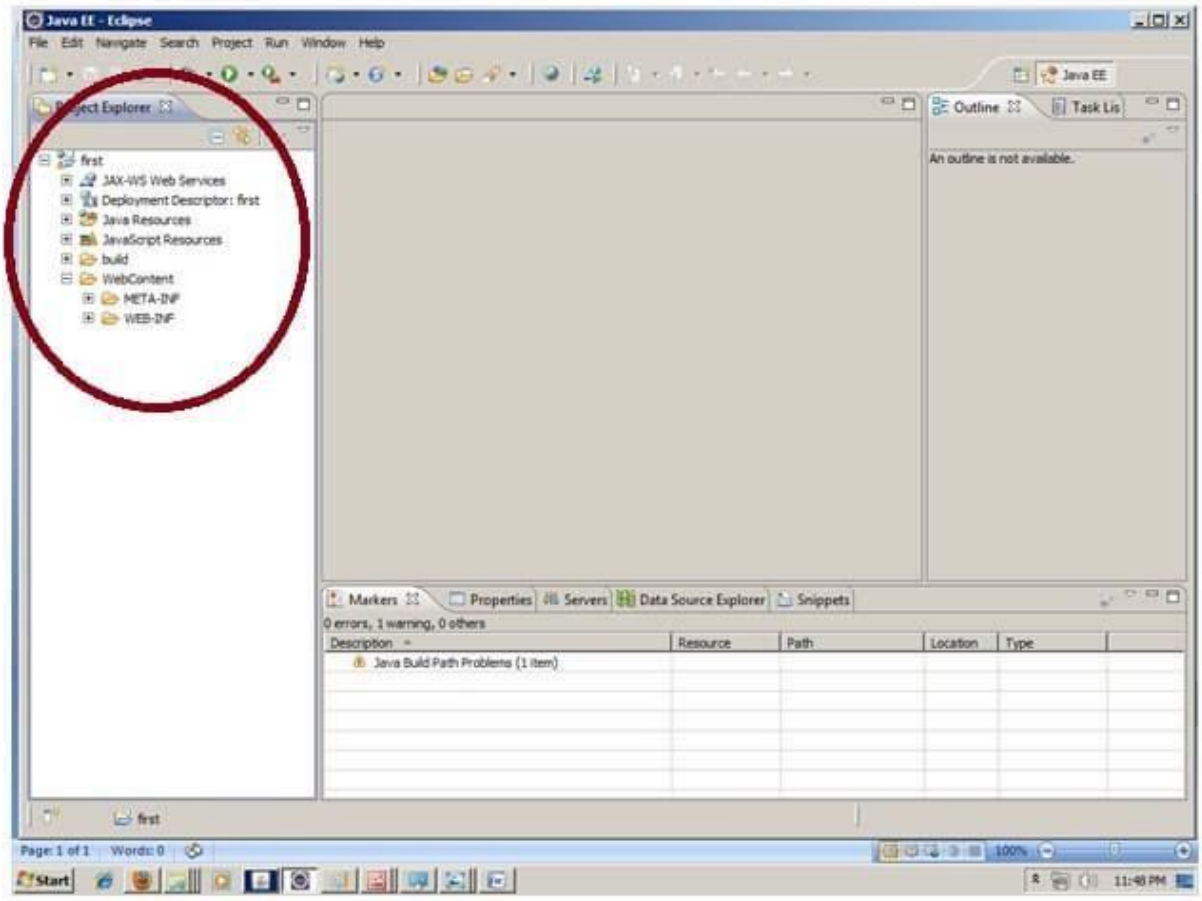


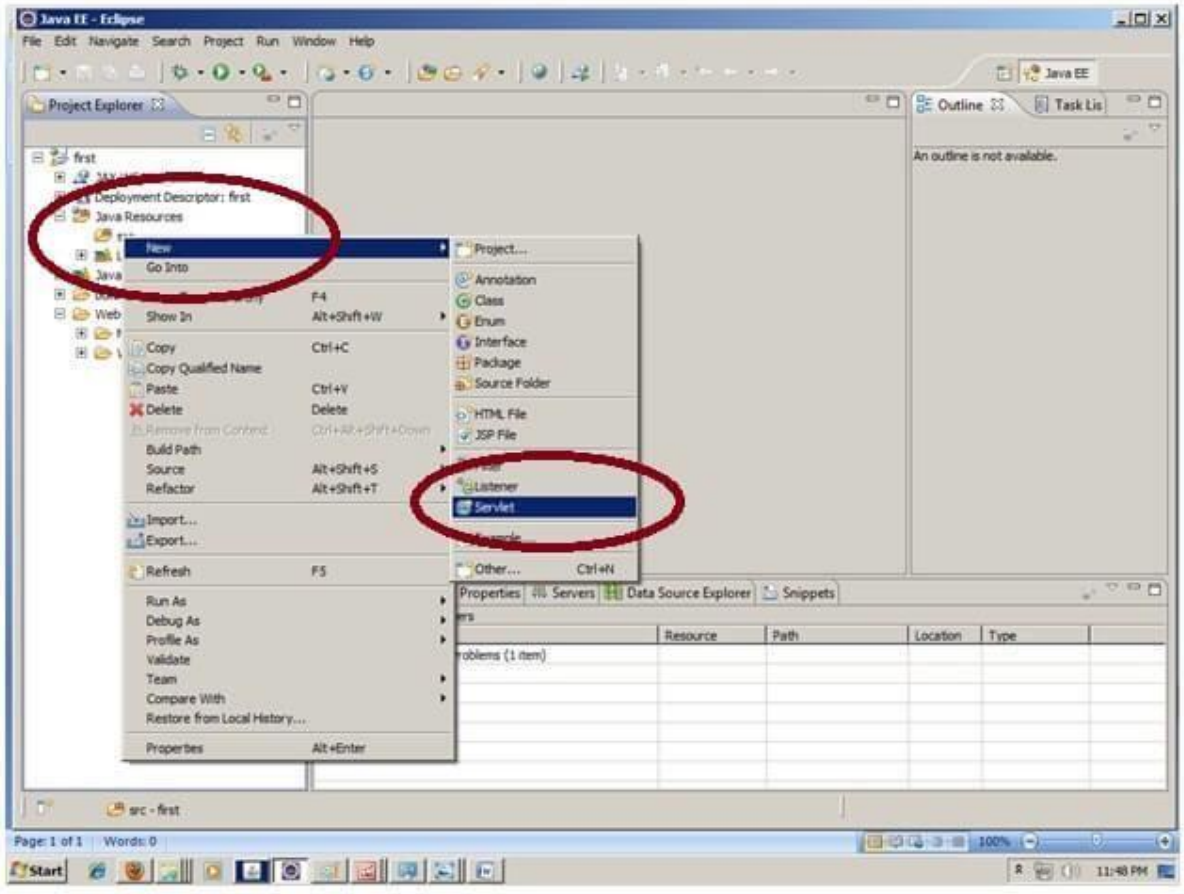
Advertisement

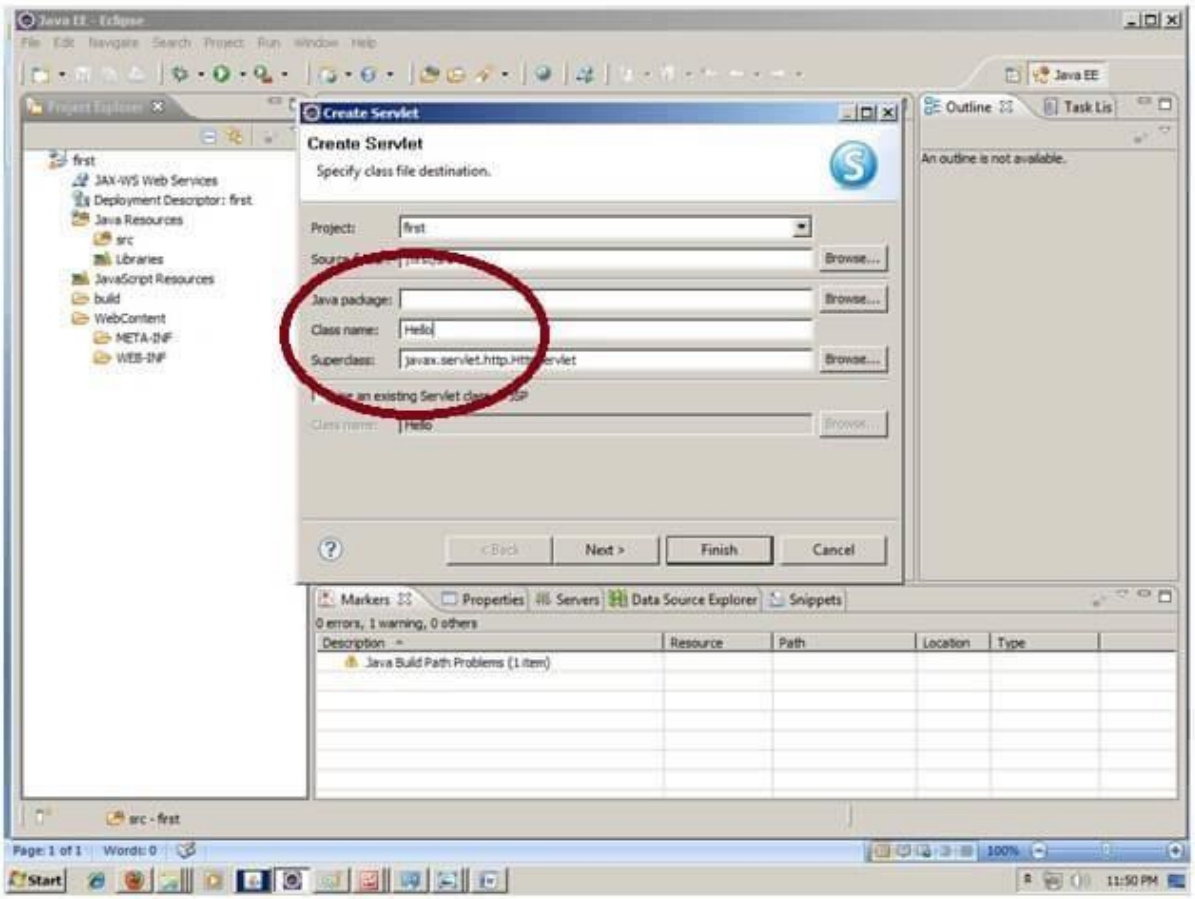
---

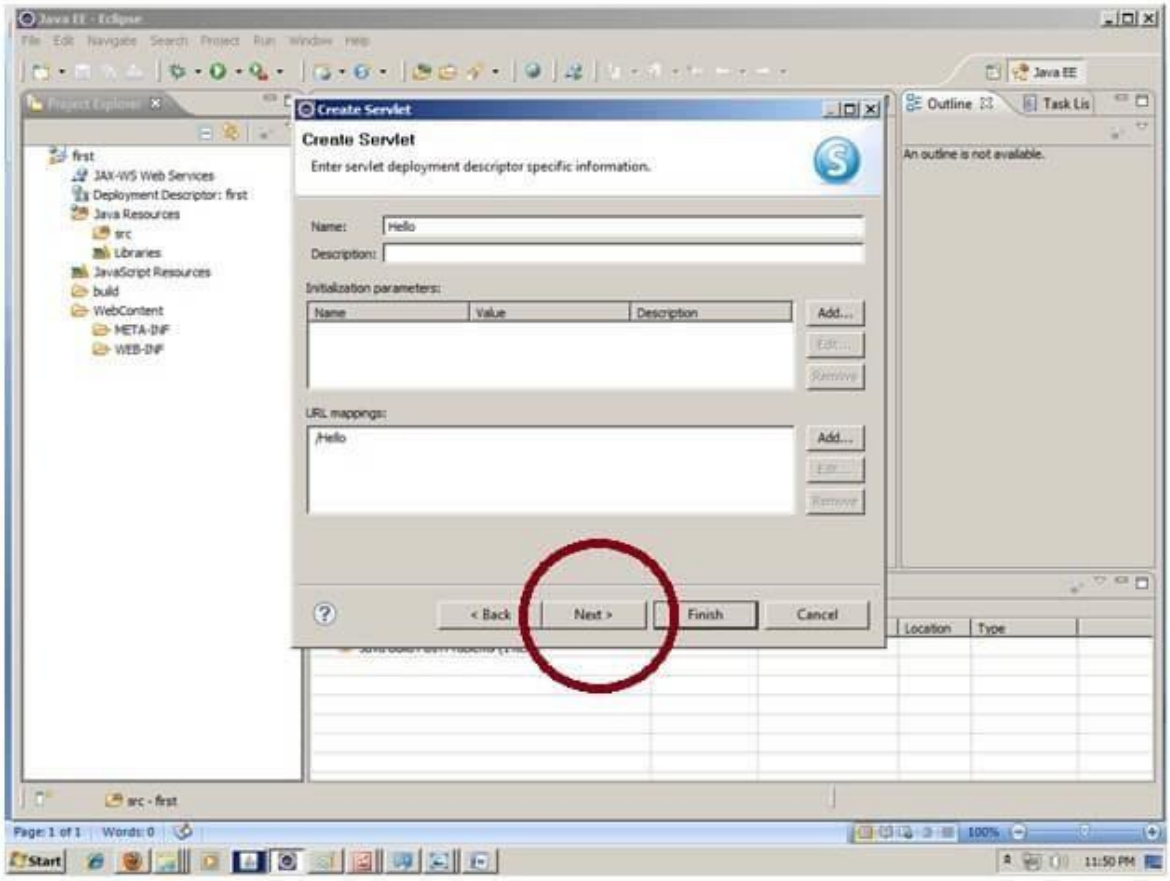
2) Create the servlet in eclipse IDE:

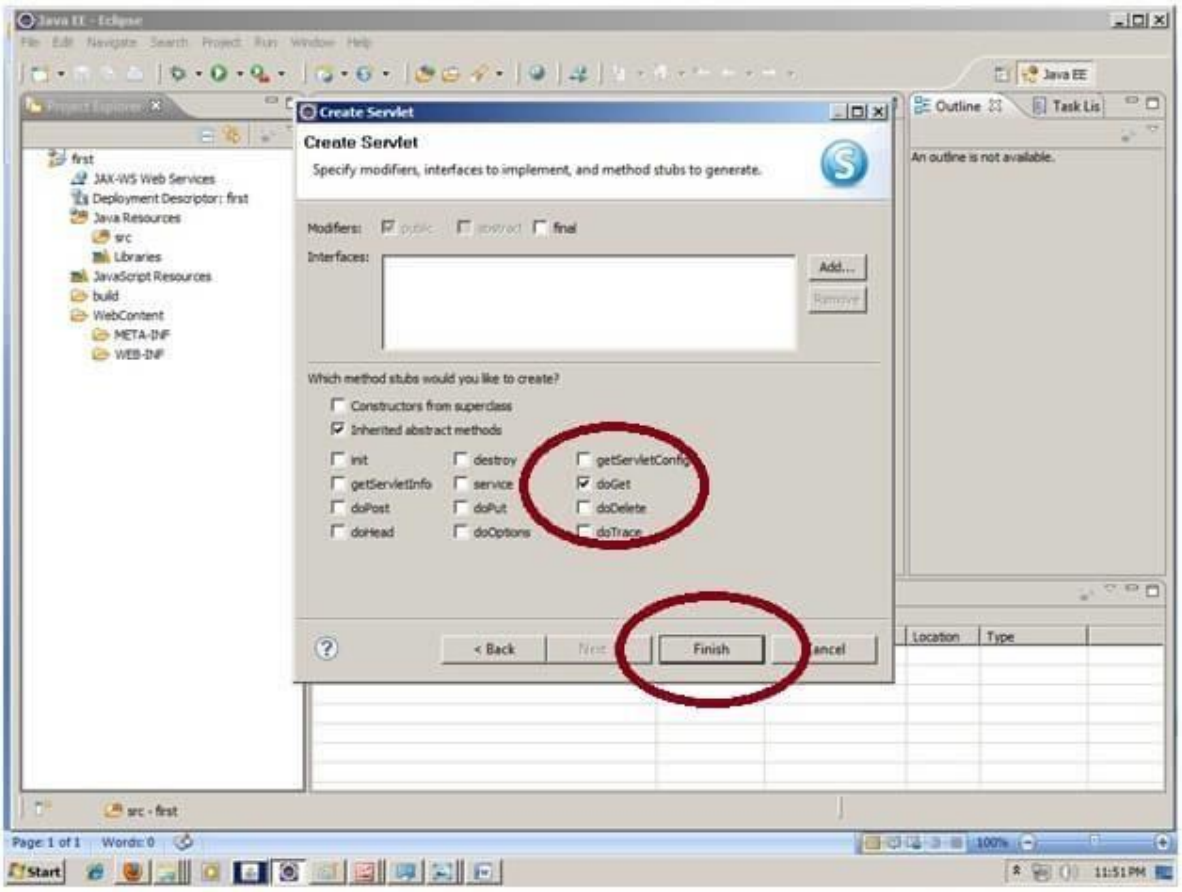
For creating a servlet, **explore the project by clicking the + icon -> explore the Java Resources -> right click on src -> New -> servlet -> write your servlet name e.g. Hello -> uncheck all the checkboxes except doGet() -> next -> Finish**.

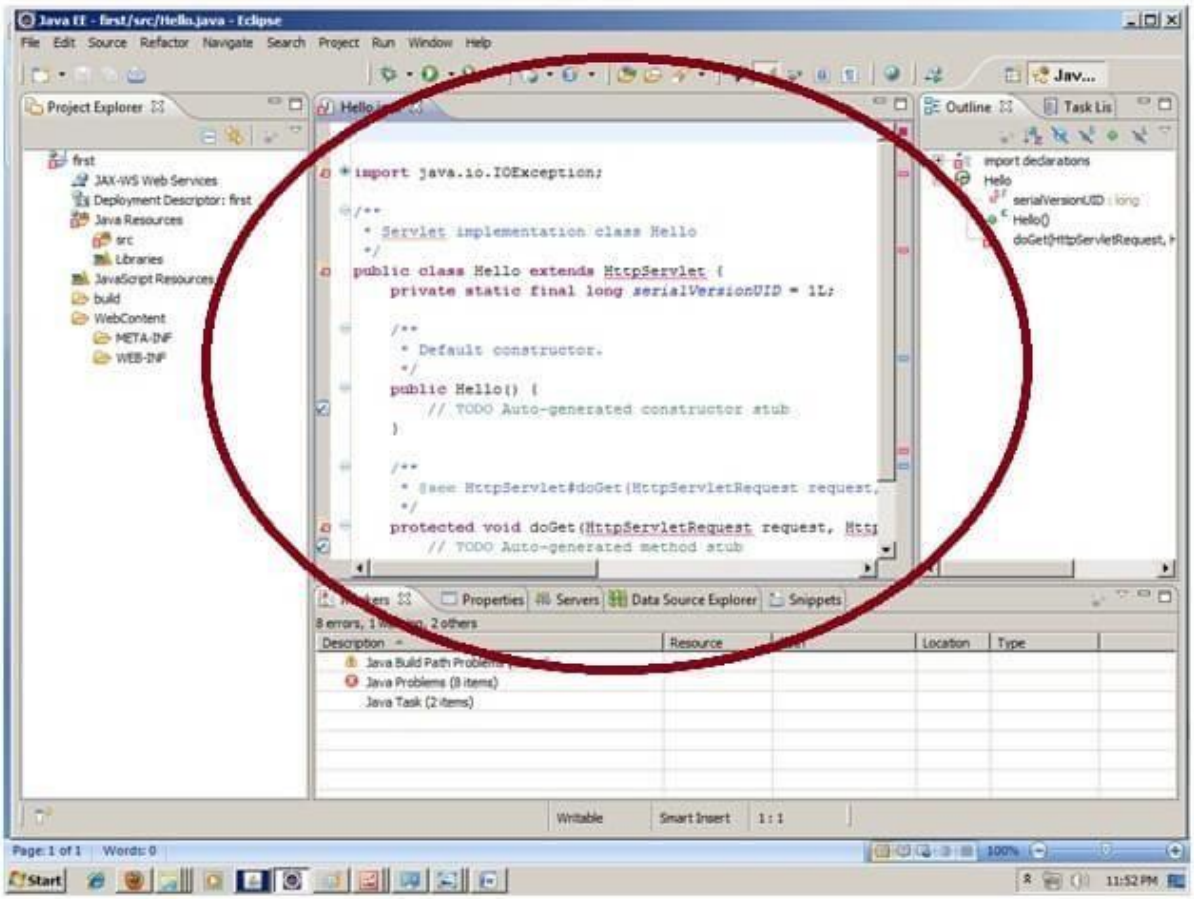

Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
Java EE
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
JavaScript Resources
build
WebContent
META-INF
WEB-INF
Outline
Task Lis
An outline is not available.
Markers
Properties
Servers
Data Source Explorer
Snippets
0 errors, 1 warning, 0 others
Description
Resource
Path
Location
Type
Java Build Path Problems (1 item)
first
Page: 1 of 1
Words: 0
100%
Start
11:48 PM


Java EE - Eclipse
File Edit Navigate Search Project Run Window Help
Java EE
Project Explorer
Outline
Task Lis
An outline is not available.
Deployment Descriptor: first
Java Resources
New
Go Into
Show In
Alt+Shift+W
Copy
Ctrl+C
Copy Qualified Name
Paste
Ctrl+V
Delete
Delete
Build Path
Source
Alt+Shift+S
Refactor
Alt+Shift+T
Import...
Export...
Refresh
F5
Run As
Debug As
Profile As
Validate
Team
Compare With
Restore from Local History...
Properties
Alt+Enter
Project...
Annotation
Class
Enum
Interface
Package
Source Folder
HTML File
JSP File
Listener
Servlet
Other...
Ctrl+N
Properties
Servers
Data Source Explorer
Snippets
Resource
Path
Location
Type
src - first
Page: 1 of 1
Words: 0
100%
Start
11:48 PM


Create Servlet
Create Servlet
Specify class file destination.
Project: first
Java package:
Class name: Hello
Superclass:
Browse...
< Back
Next >
Finish
Cancel
Outline
An outline is not available.
Markers
Properties
Servers
Data Source Explorer
Snippets
0 errors, 1 warning, 0 others
Description
Resource
Path
Location
Type
Java Build Path Problems (1 item)
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF


Create Servlet
Create Servlet
Enter servlet deployment descriptor specific information.
Name: Hello
Description:
Initialization parameters:
Name
Value
Description
Add...
URL mappings:
/Hello
< Back
Next >
Finish
Cancel
Outline
An outline is not available.
Location
Type
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF

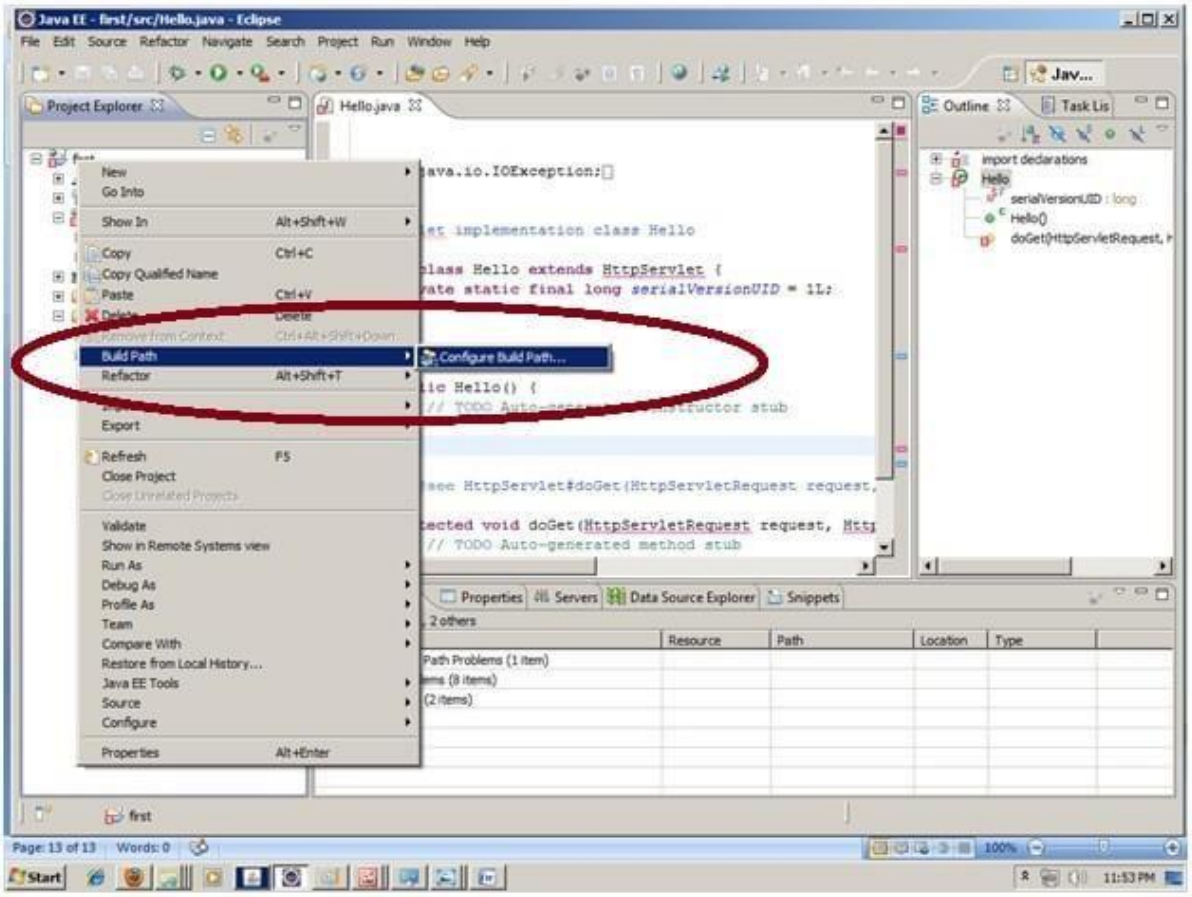

Create Servlet
Create Servlet
Specify modifiers, interfaces to implement, and method stubs to generate.
Modifiers:
Interfaces:
Add...
Remove
Which method stubs would you like to create?
Constructors from superclass
Inherited abstract methods
init
destroy
getServletConfig
getServletInfo
service
doGet
doPost
doPut
doDelete
doHead
doOptions
doTrace
< Back
Next
Finish
Cancel
Outline
Task Lis
An outline is not available.
Location
Type
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF

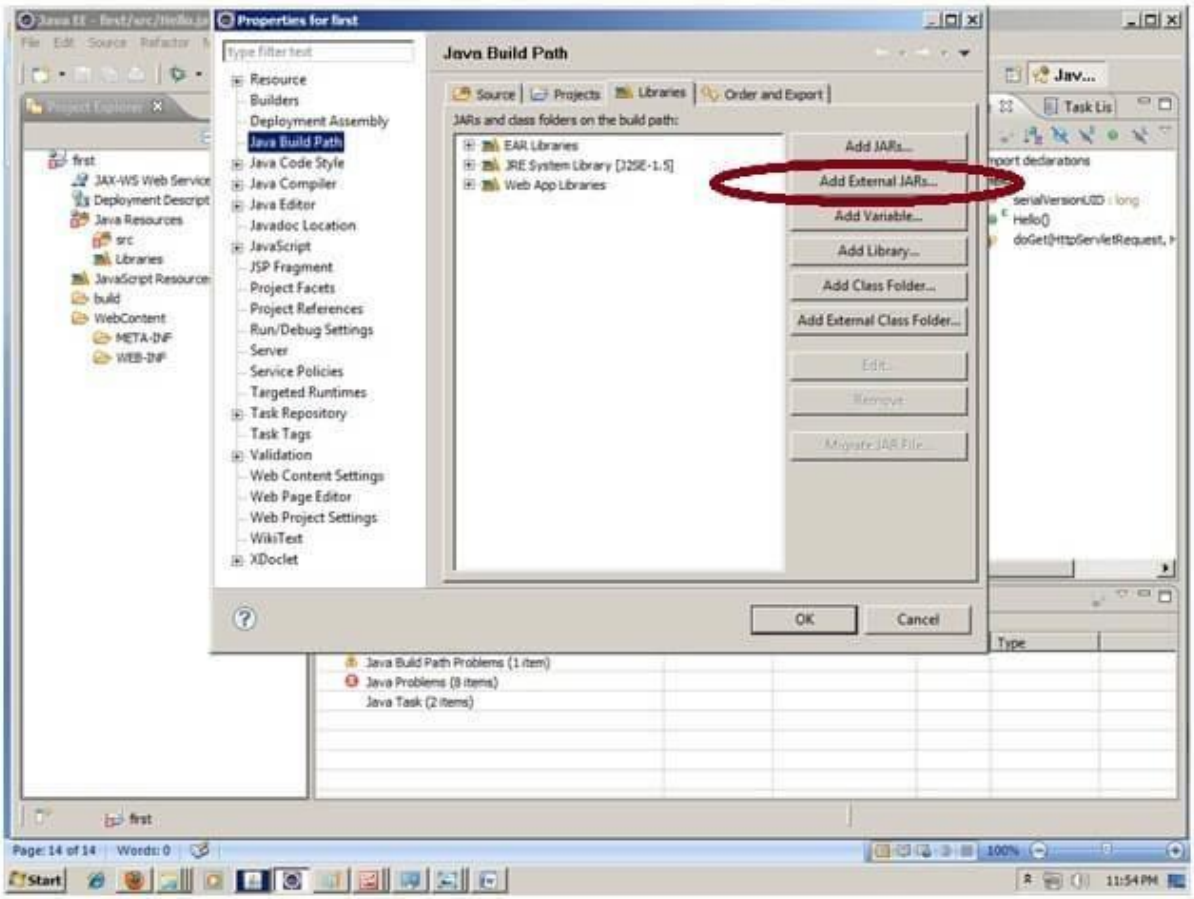

Java EE - first/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
Outline
Task Lis
import java.io.IOException;
/**
* Servlet implementation class Hello
*/
public class Hello extends HttpServlet {
private static final long serialVersionUID = 1L;
/**
* Default constructor.
*/
public Hello() {
// TODO Auto-generated constructor stub
}
/**
* @see HttpServlet#doGet(HttpServletRequest request,
*/
protected void doGet(HttpServletRequest request, Htt
// TODO Auto-generated method stub
import declarations
Hello
serialVersionUID : long
Hello()
doGet(HttpServletRequest,
Properties
Servers
Data Source Explorer
Snippets
Description
Resource
Location
Type
Java Problems (8 items)
Java Task (2 items)
Writable
Smart Insert
1 : 1

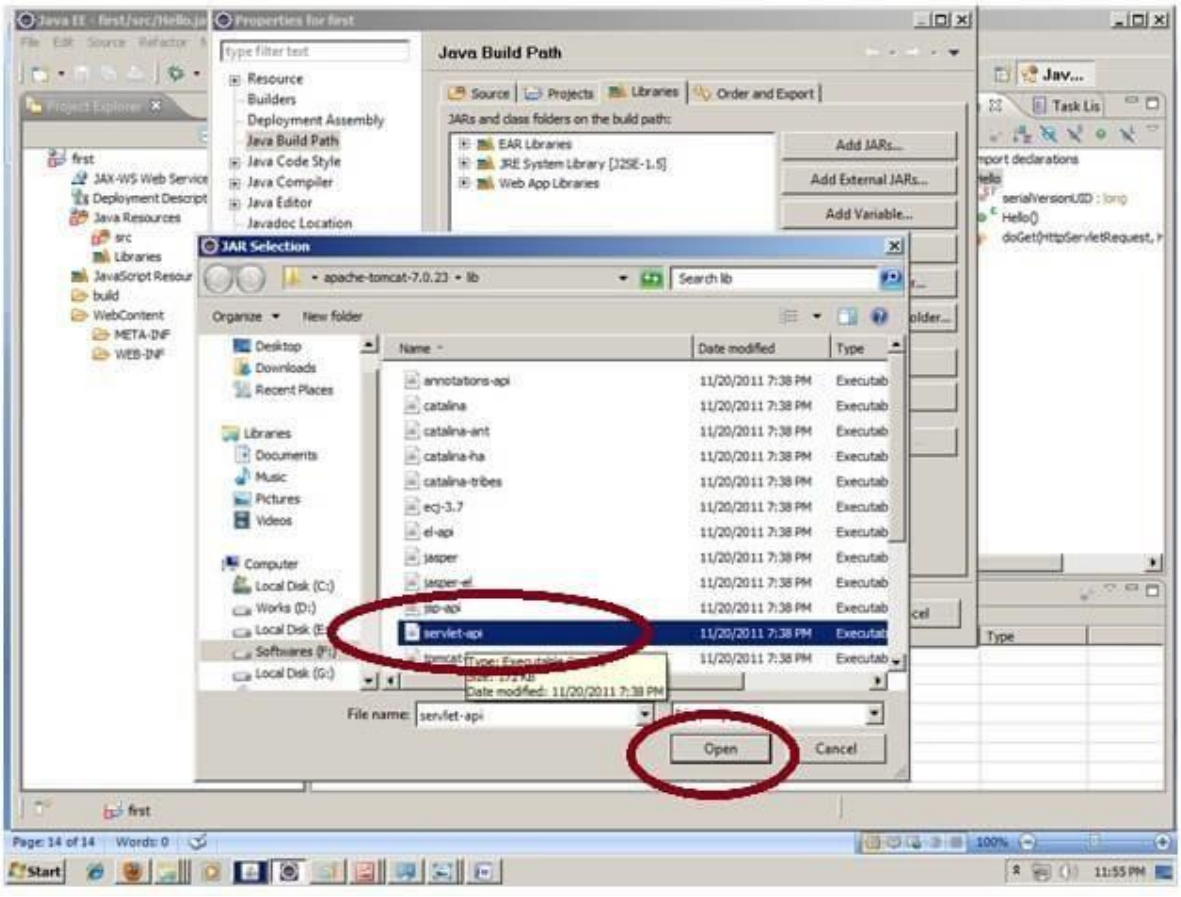
Advertisement

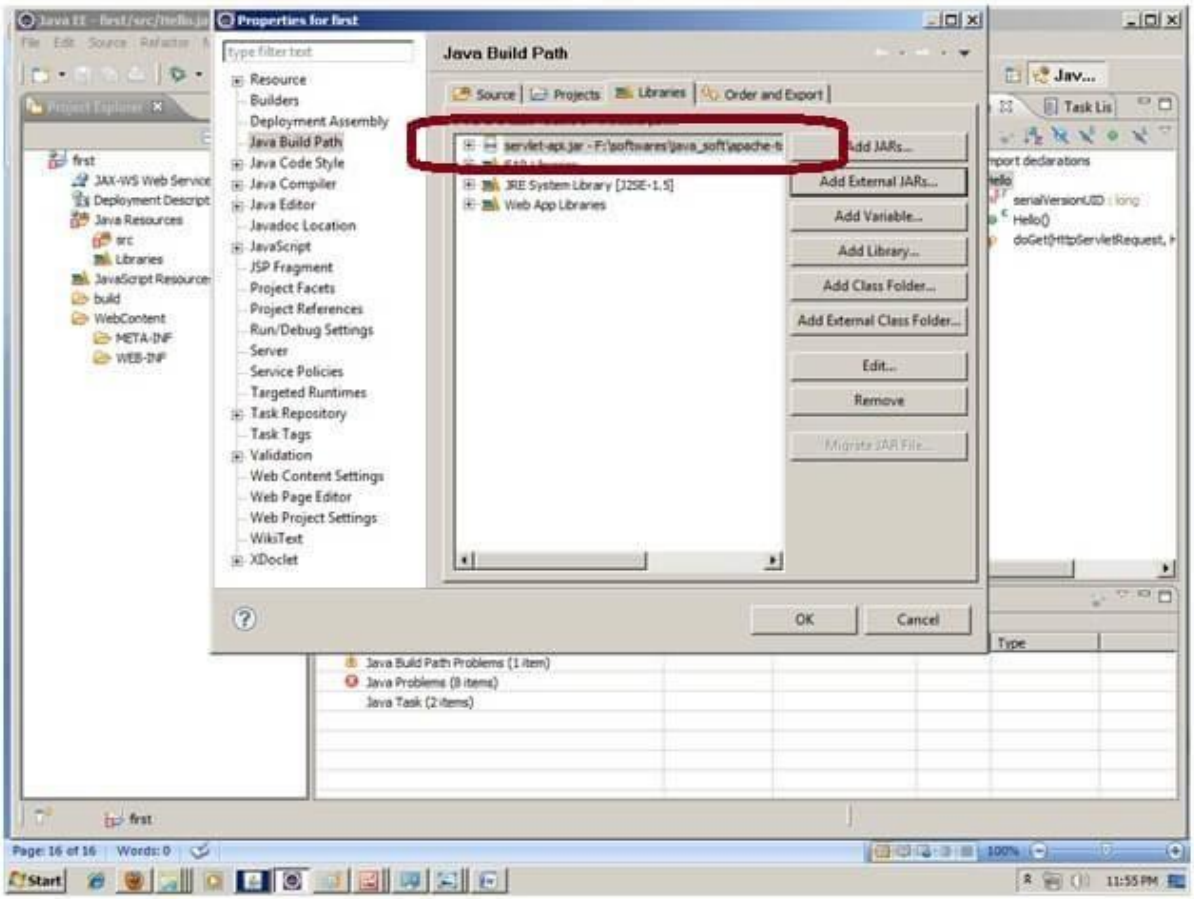
3) add jar file in eclipse IDE:

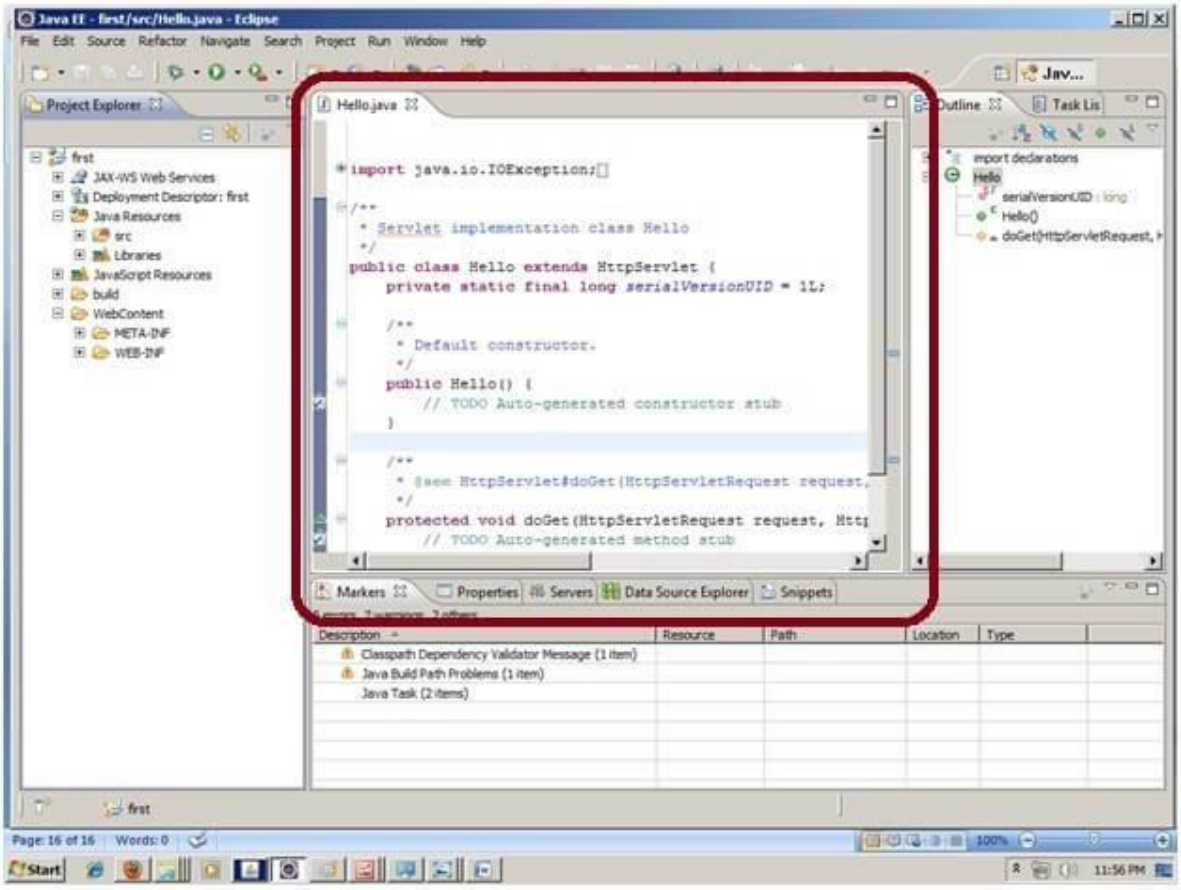
For adding a jar file, **right click on your project -> Build Path -> Configure Build Path -> click on Libra JARs button -> select the servlet-api.jar file under tomcat/lib -> ok.**



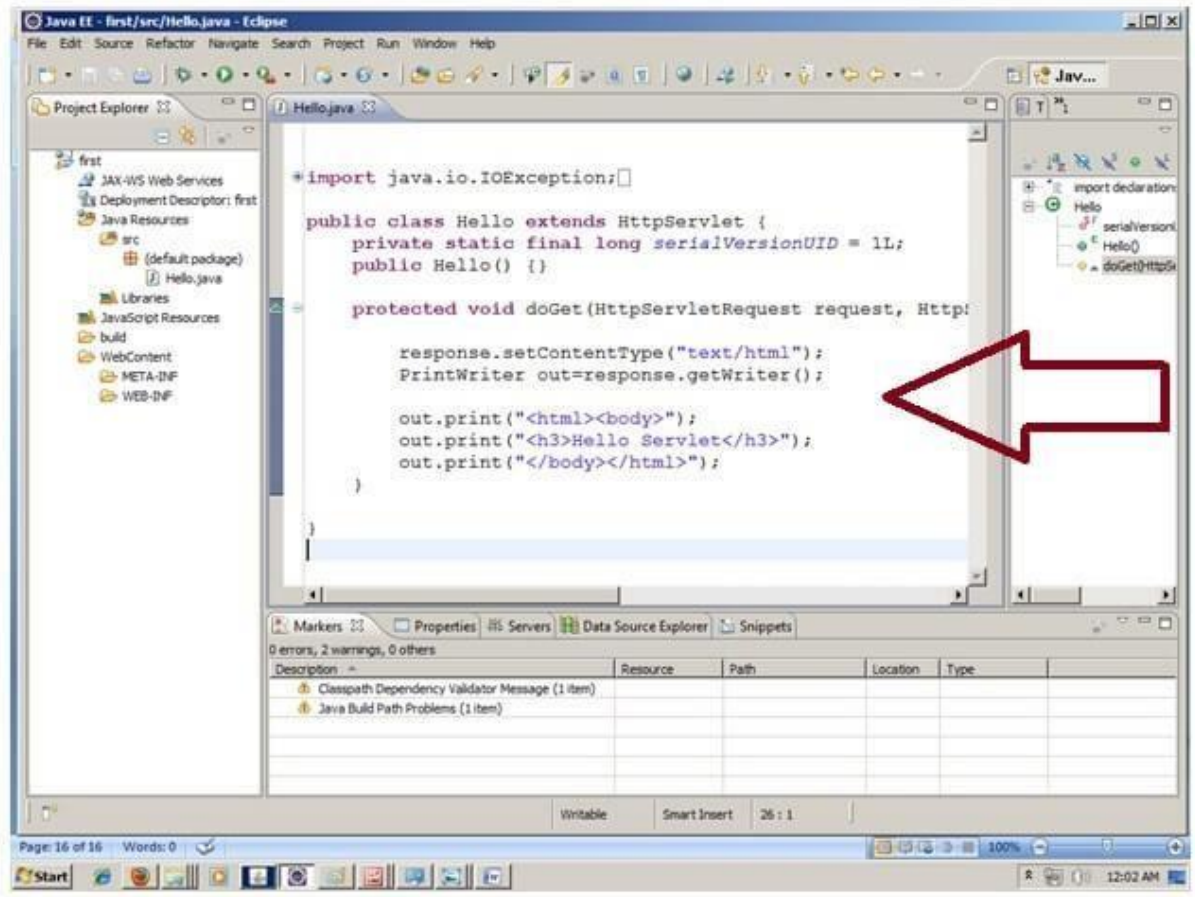

Properties for first
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
EAR Libraries
JRE System Library [J2SE-1.5]
Web App Libraries
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
OK
Cancel
Resource
Builders
Deployment Assembly
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
JavaScript
JSP Fragment
Project Facets
Project References
Run/Debug Settings
Server
Service Policies
Targeted Runtimes
Task Repository
Task Tags
Validation
Web Content Settings
Web Page Editor
Web Project Settings
WikiText
XDoclet
Java Build Path Problems (1 item)
Java Problems (8 items)
Java Task (2 items)


JAR Selection
apache-tomcat-7.0.23 ▸ lib
Search lib
Organize
New folder
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
Works (D:)
Local Disk (E:)
Softwares (F:)
Local Disk (G:)
Name
Date modified
Type
annotations-api 11/20/2011 7:38 PM Executab
catalina 11/20/2011 7:38 PM Executab
catalina-ant 11/20/2011 7:38 PM Executab
catalina-ha 11/20/2011 7:38 PM Executab
catalina-tribes 11/20/2011 7:38 PM Executab
ecj-3.7 11/20/2011 7:38 PM Executab
el-api 11/20/2011 7:38 PM Executab
jasper 11/20/2011 7:38 PM Executab
jasper-el 11/20/2011 7:38 PM Executab
jsp-api 11/20/2011 7:38 PM Executab
servlet-api 11/20/2011 7:38 PM Executab
Date modified: 11/20/2011 7:38 PM
File name: servlet-api
Open
Cancel


Properties for first
Java Build Path
Source
Projects
Libraries
Order and Export
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
OK
Cancel


Java EE - first/src/Hello.java - Eclipse
Hello.java
Project Explorer
Markers
Properties
Servers
Data Source Explorer
Snippets

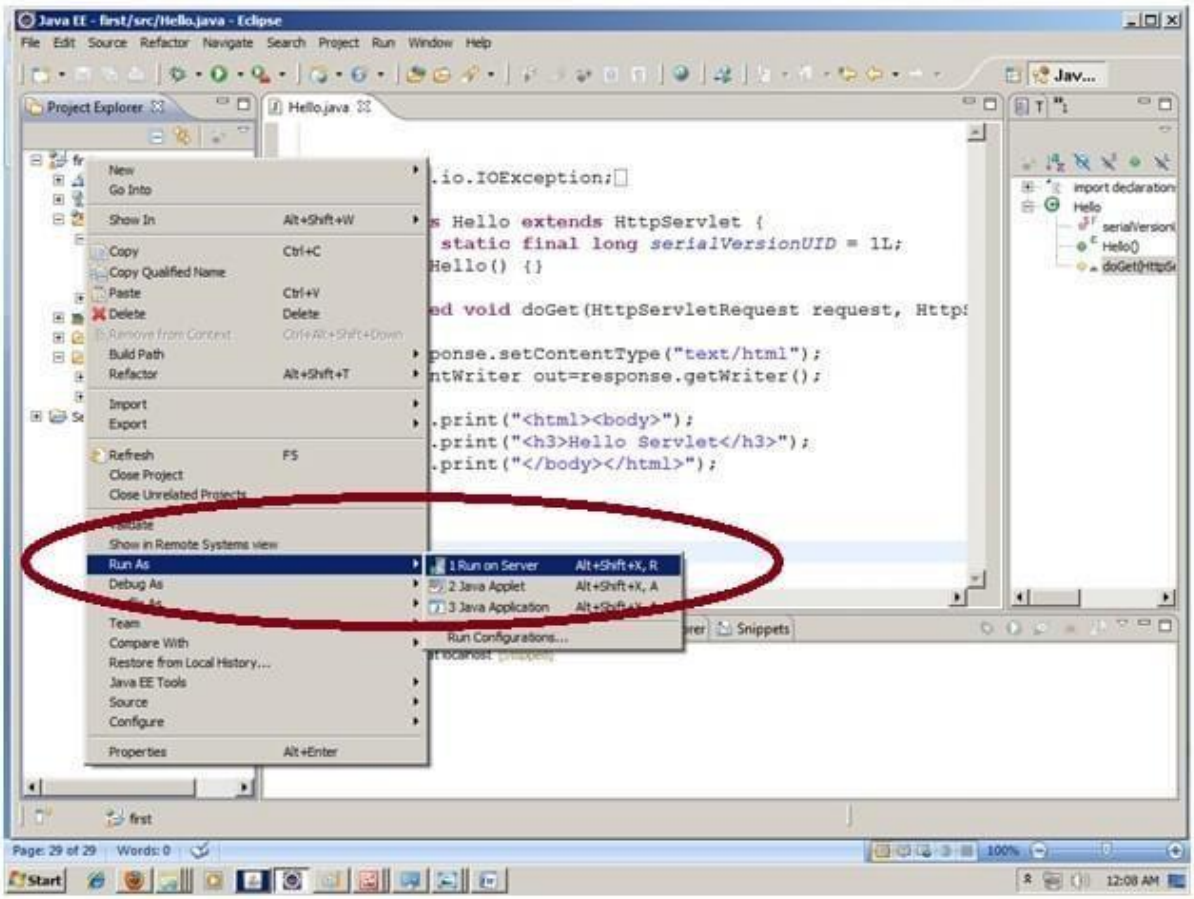
Now servlet has been created, Let's write the first servlet code.

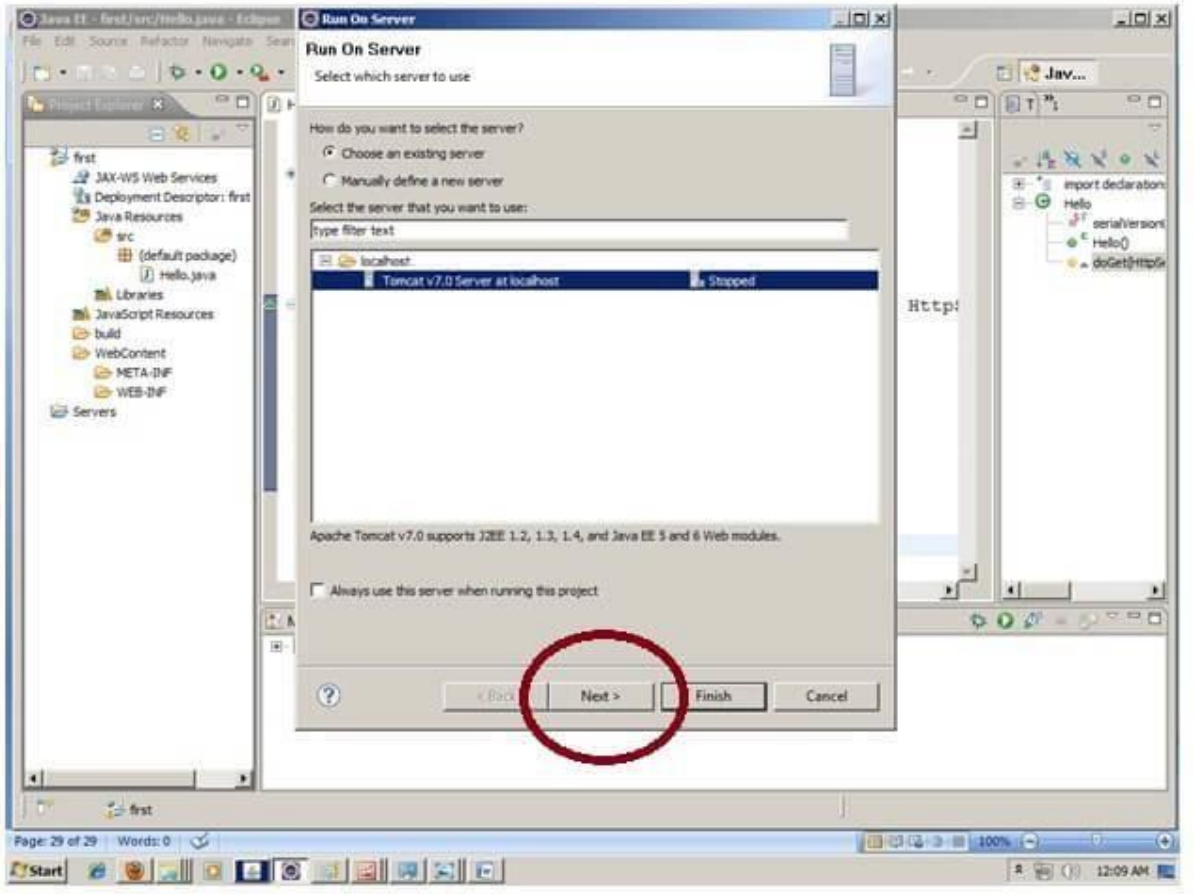


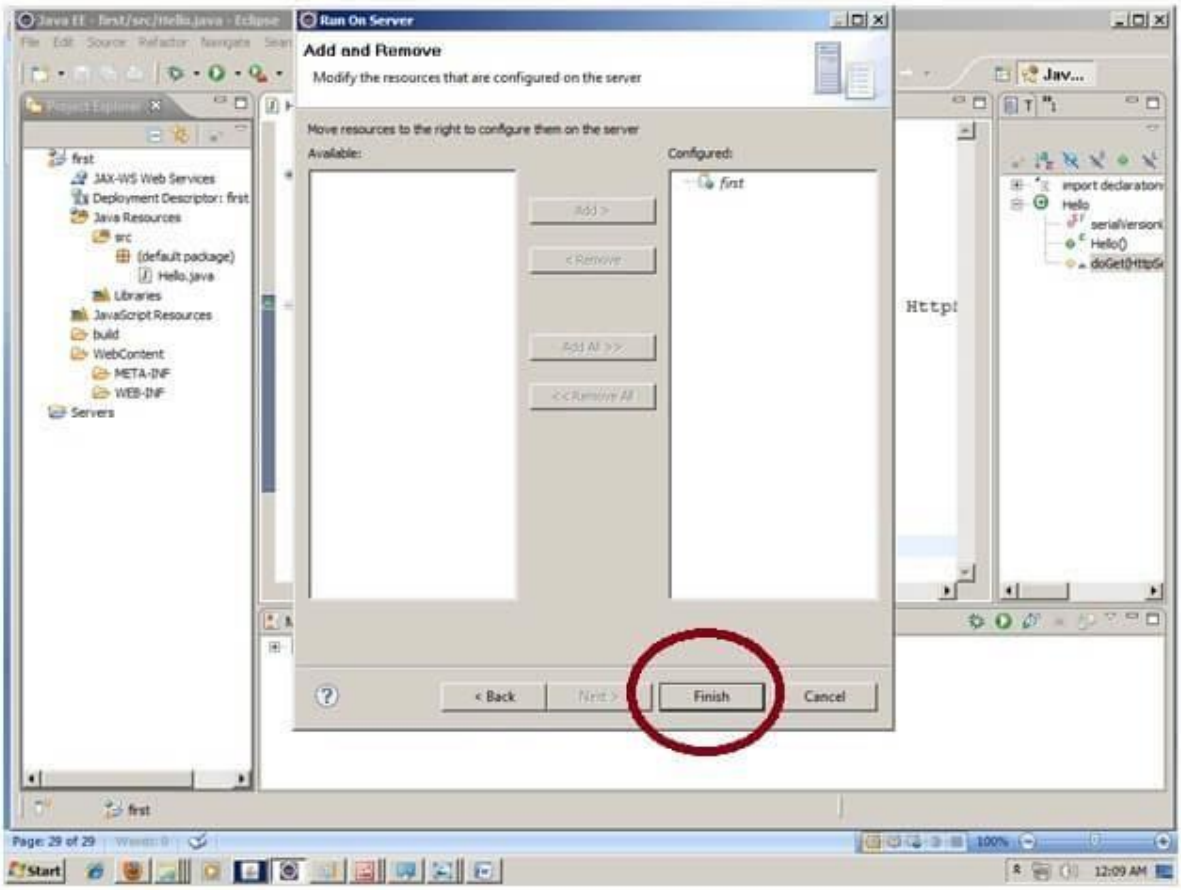
---

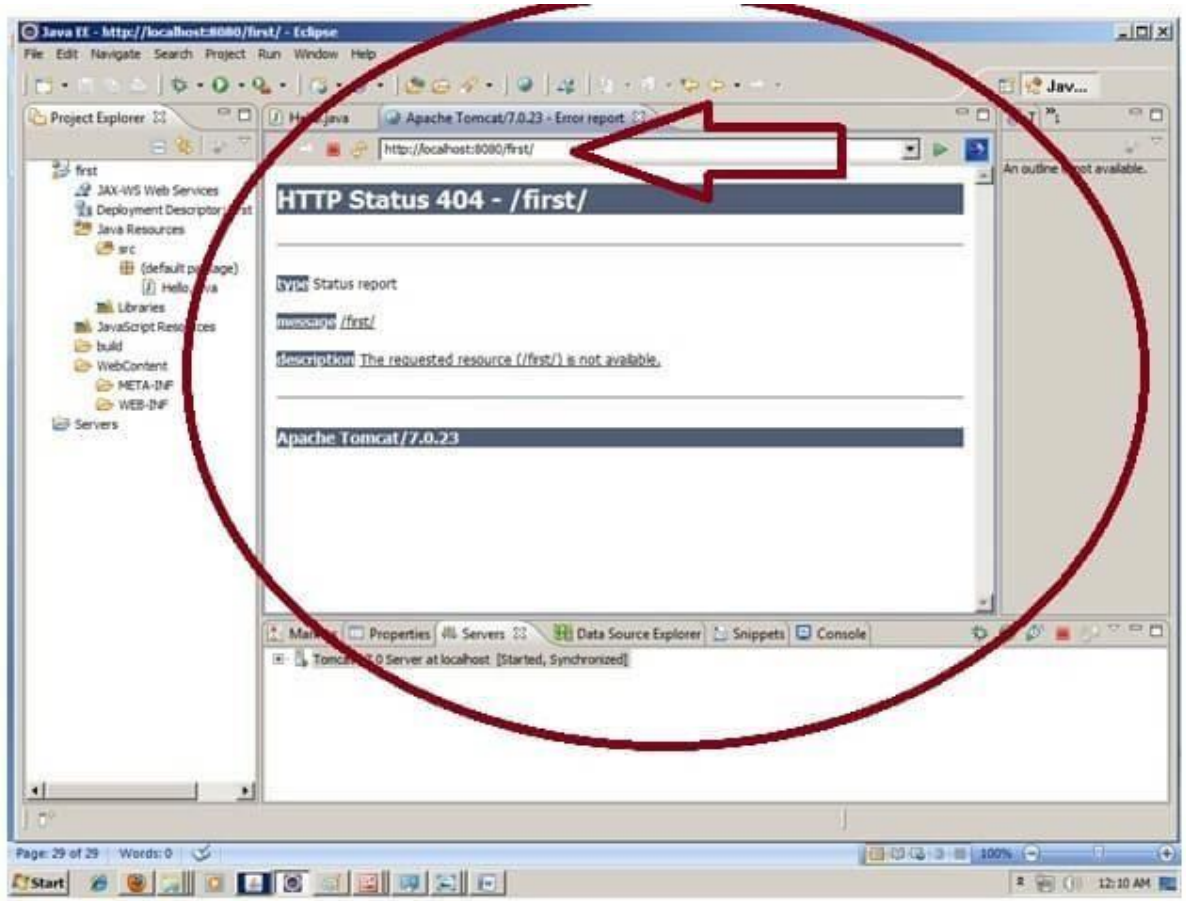
4) Start the server and deploy the project:

For starting the server and deploying the project in one step, **Right click on your project -> Run As -> Run on Server -> choose tomcat server -> next -> addAll -> finish.**

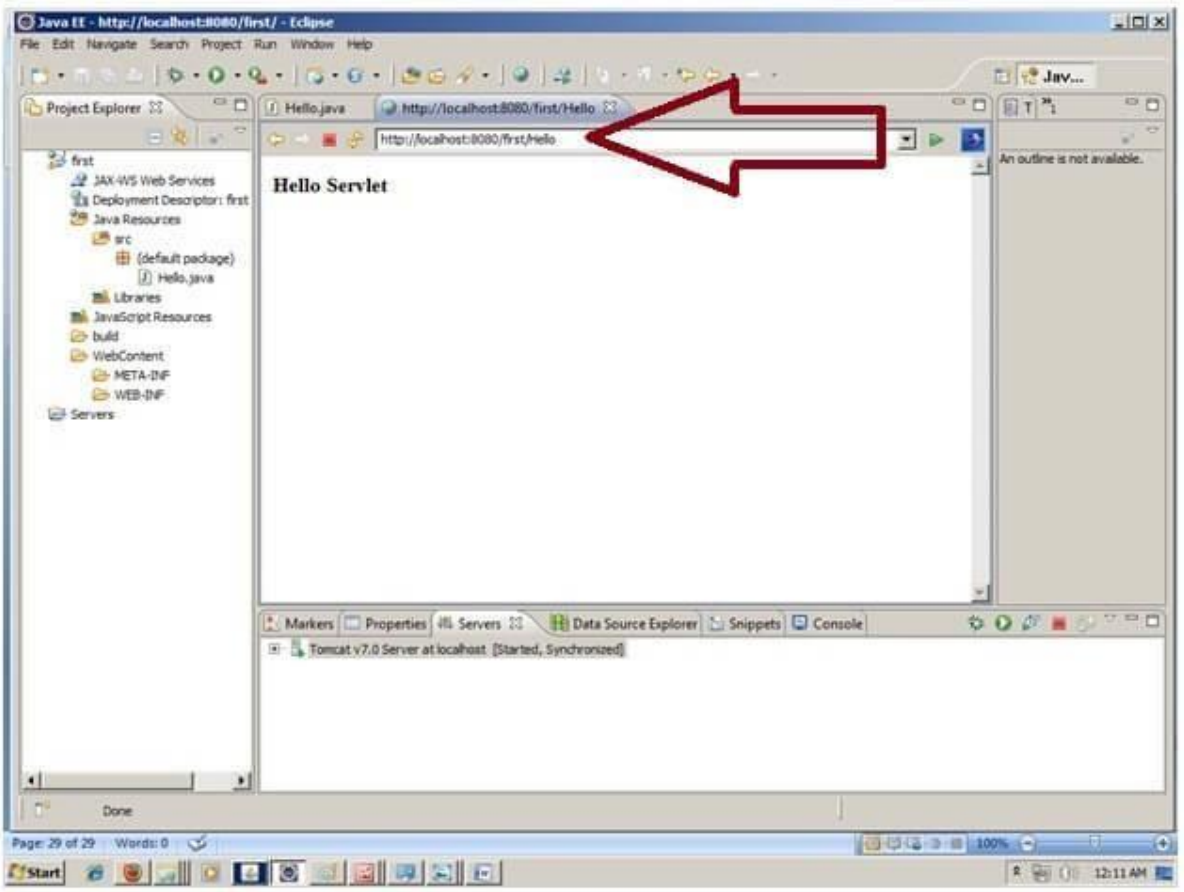

Java EE - first/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
Hello.java
New
Go Into
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Build Path
Refactor Alt+Shift+T
Import
Export
Refresh F5
Close Project
Close Unrelated Projects
Show in Remote Systems view
Run As
Debug As
Team
Compare With
Restore from Local History...
Java EE Tools
Source
Configure
Properties Alt+Enter
1 Run on Server Alt+Shift+X, R
2 Java Applet Alt+Shift+X, A
3 Java Application
Run Configurations...
Snippets
first


Run On Server
Run On Server
Select which server to use
How do you want to select the server?
Choose an existing server
Manually define a new server
Select the server that you want to use:
type filter text
localhost
Tomcat v7.0 Server at localhost
Stopped
Apache Tomcat v7.0 supports J2EE 1.2, 1.3, 1.4, and Java EE 5 and 6 Web modules.
Always use this server when running this project
Next >
Finish
Cancel
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
Servers


Run On Server
Add and Remove
Modify the resources that are configured on the server
Move resources to the right to configure them on the server
Available:
Configured:
first
Add >
< Remove
Add All >>
<< Remove All
< Back
Next >
Finish
Cancel


Java EE - http://localhost:8080/first/ - Eclipse
Apache Tomcat/7.0.23 - Error report
http://localhost:8080/first/
HTTP Status 404 - /first/
type Status report
message /first/
description The requested resource (/first/) is not available.
Apache Tomcat/7.0.23
Properties
Servers
Data Source Explorer
Snippets
Console

Now tomcat server has been started and project is deployed. To access the servlet write the url pattern name in the URL bar of the browser. In this case Hello then enter.



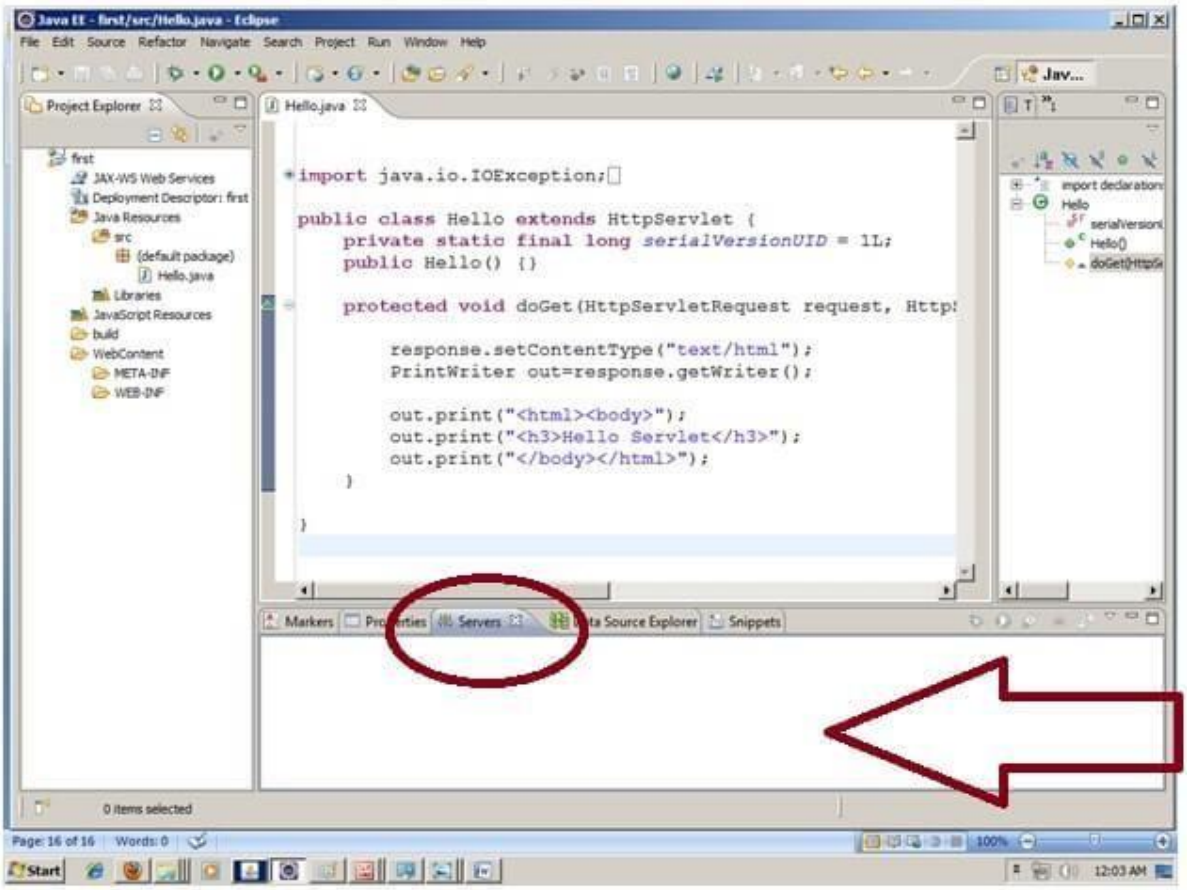
---

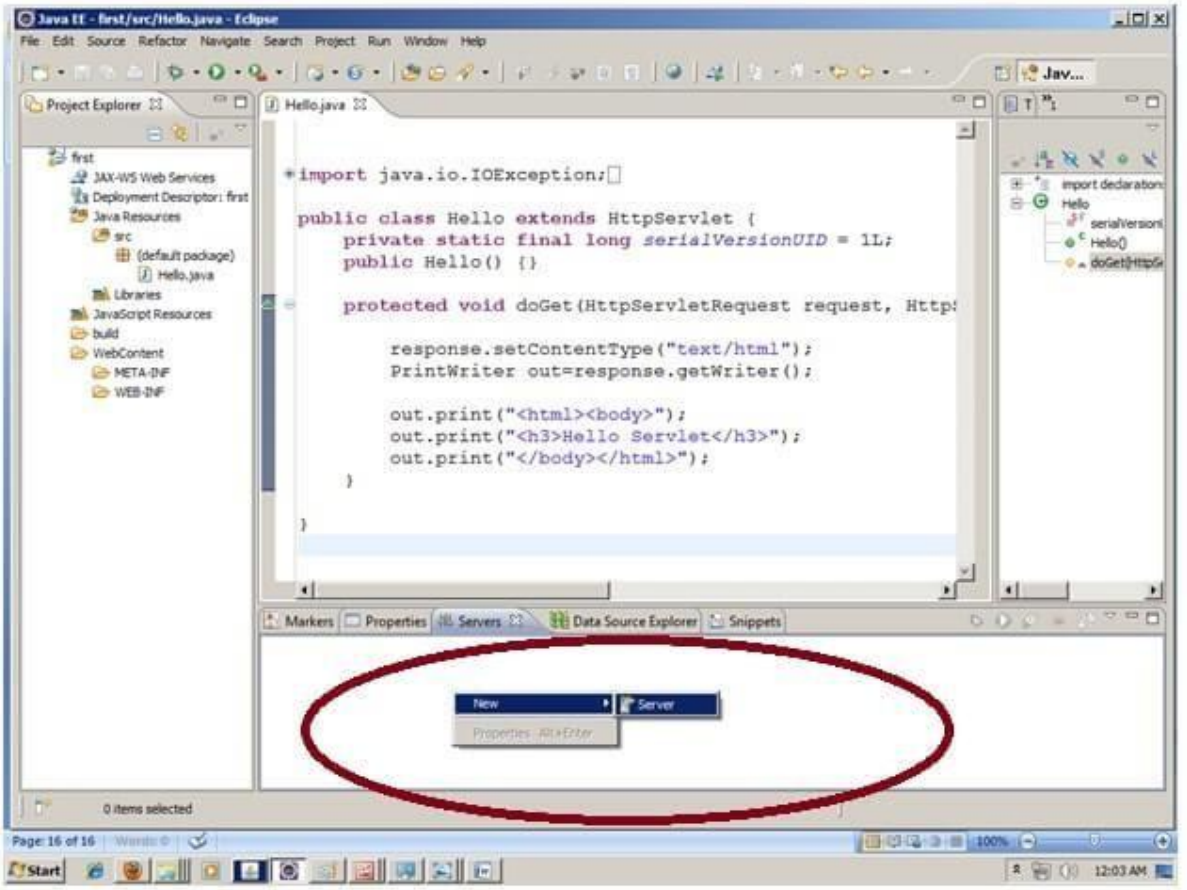
download this example

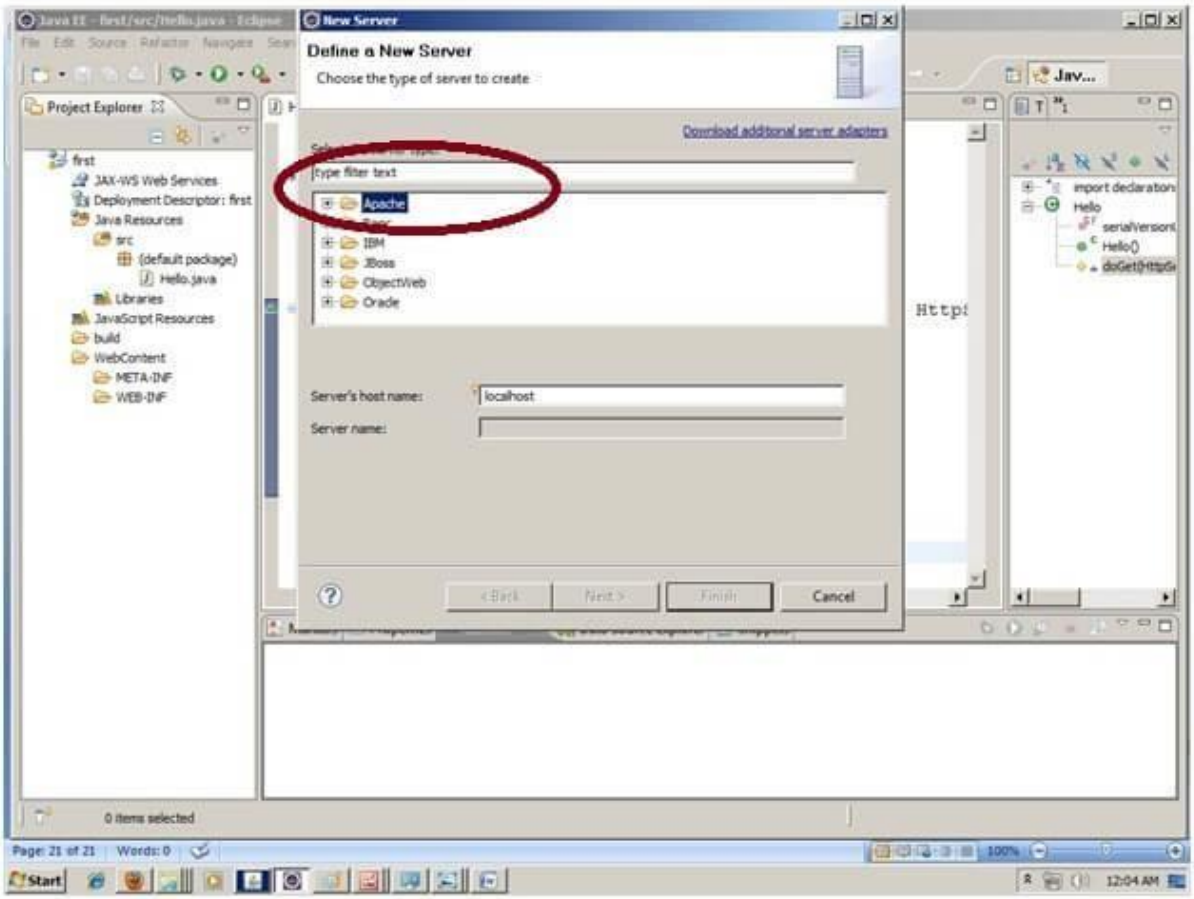
How to configure tomcat server in Eclipse ? (One time Requirement)

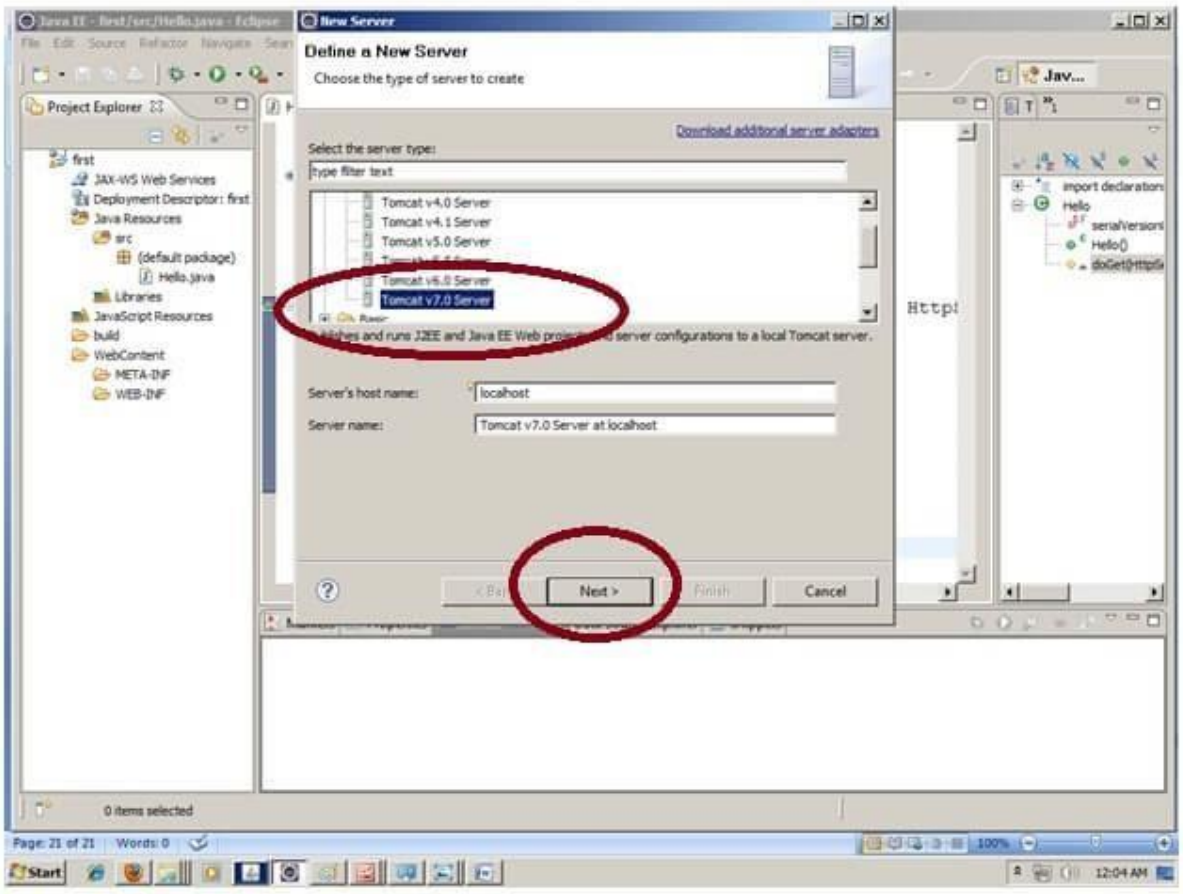
If you are using **Eclipse IDE first time**, you need to configure the tomcat server First.

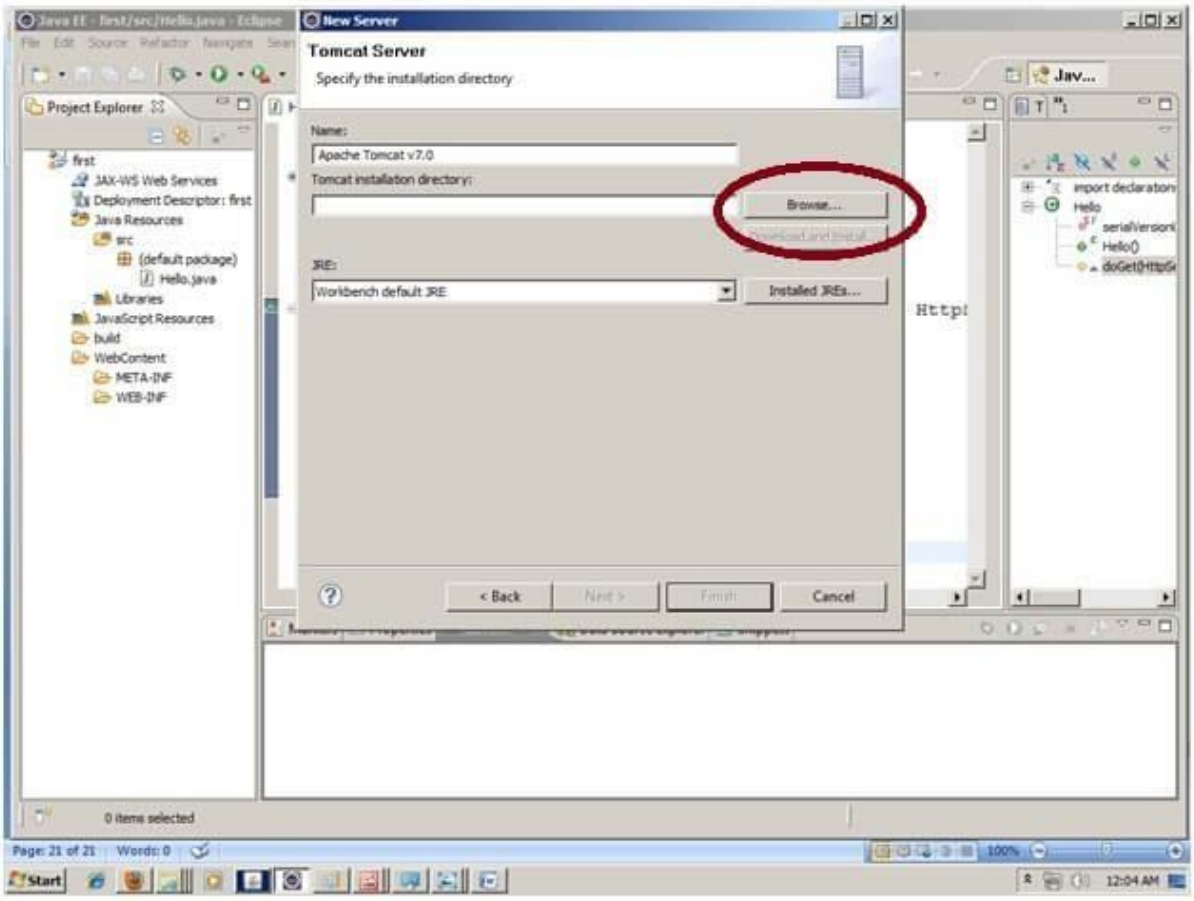
For configuring the tomcat server in eclipse IDE, **click on servers tab at the bottom side of the IDE -> right click on blank area -> New -> Servers -> choose tomcat then its version -> next -> click on Browse button -> select the apache tomcat root folder previous to bin -> next -> addAll -> Finish.**

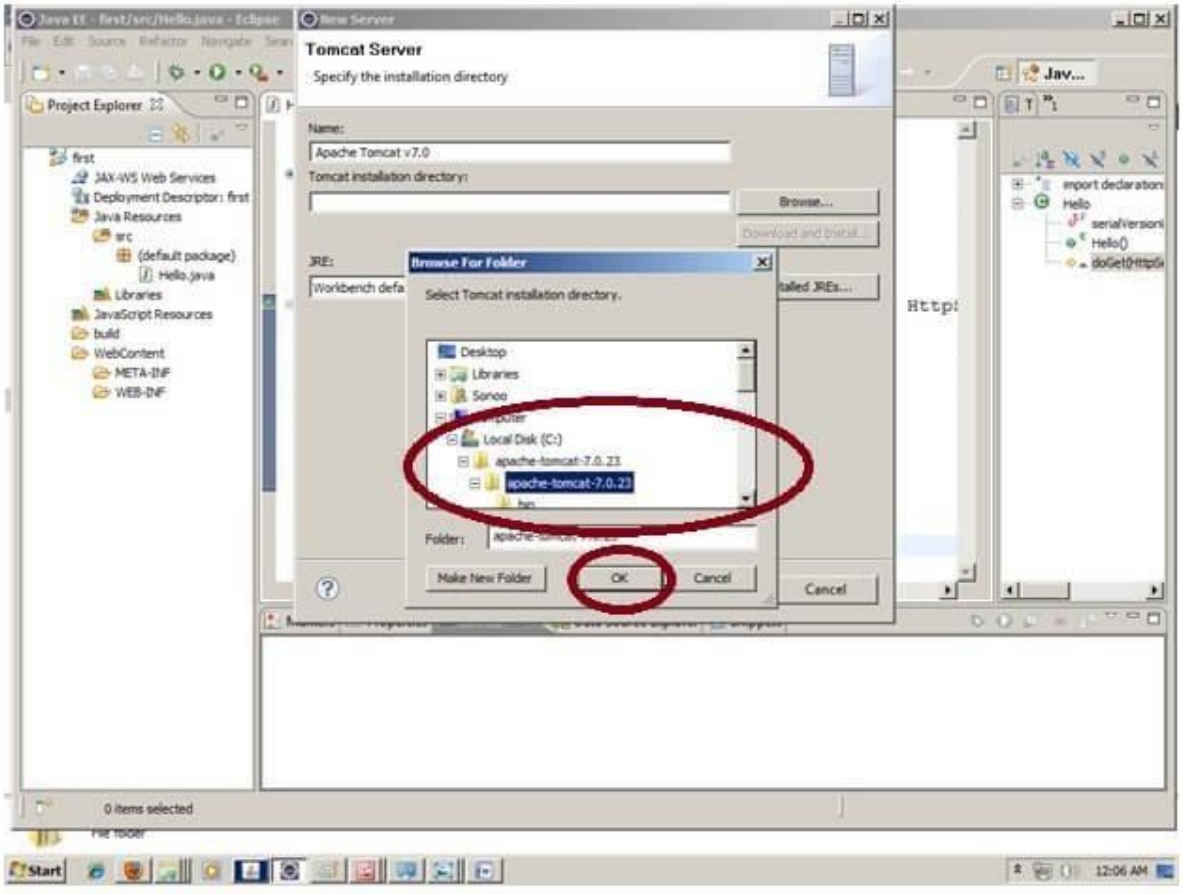

Java EE - first/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
Hello.java
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
import java.io.IOException;
public class Hello extends HttpServlet {
private static final long serialVersionUID = 1L;
public Hello() {}
protected void doGet(HttpServletRequest request, Http
response.setContentType("text/html");
PrintWriter out=response.getWriter();
out.print("<html><body>");
out.print("<h3>Hello Servlet</h3>");
out.print("</body></html>");
}
}
Markers
Properties
Servers
Data Source Explorer
Snippets
0 items selected
Page: 16 of 16
Words: 0
100%
Start
12:03 AM

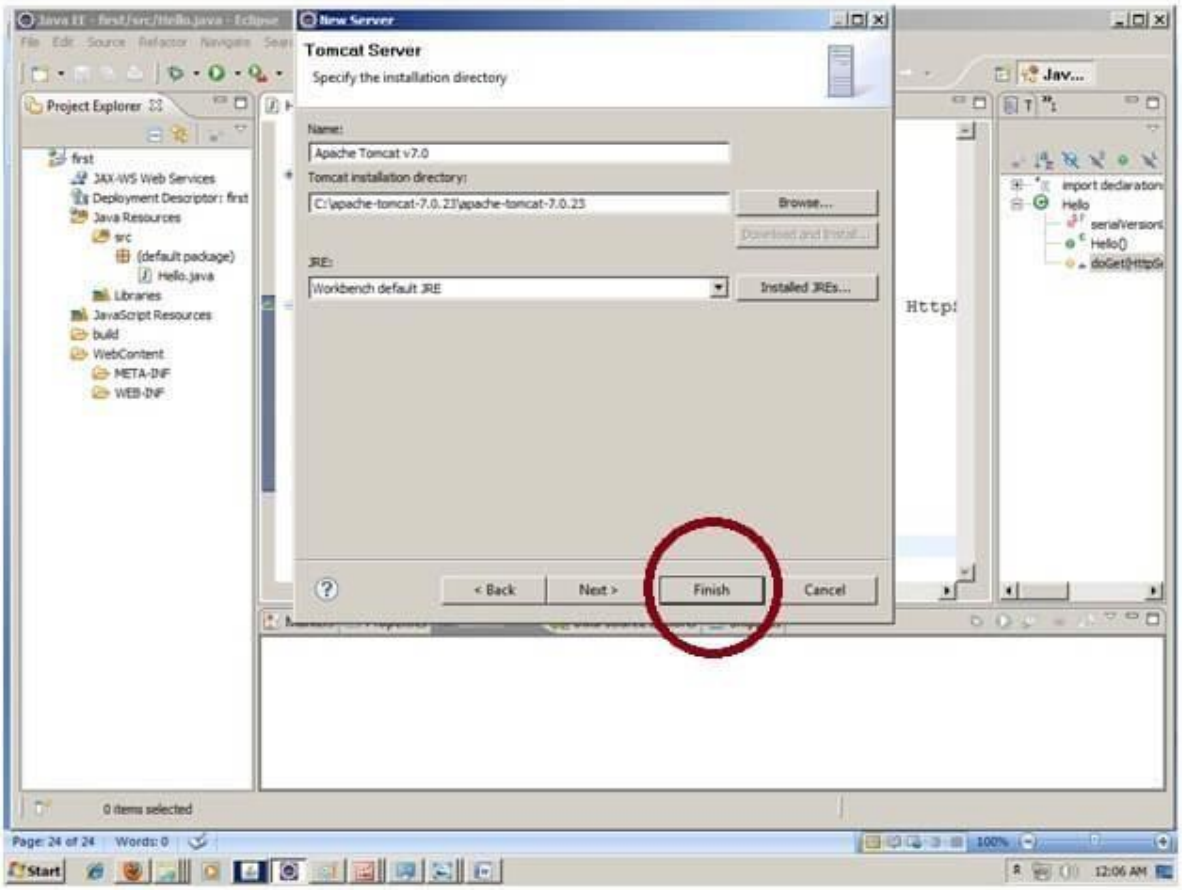

Java EE - first/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
Hello.java
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
import java.io.IOException;
public class Hello extends HttpServlet {
private static final long serialVersionUID = 1L;
public Hello() {}
protected void doGet(HttpServletRequest request, Http
response.setContentType("text/html");
PrintWriter out=response.getWriter();
out.print("<html><body>");
out.print("<h3>Hello Servlet</h3>");
out.print("</body></html>");
}
}
Markers
Properties
Servers
Data Source Explorer
Snippets
New
Server
Properties Alt+Enter
0 items selected
Page: 16 of 16
100%
Start
12:03 AM


New Server
Define a New Server
Choose the type of server to create
Download additional server adapters
type filter text
Apache
IBM
JBoss
ObjectWeb
Oracle
Server's host name:
localhost
Server name:
< Back
Next >
Finish
Cancel
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
0 items selected

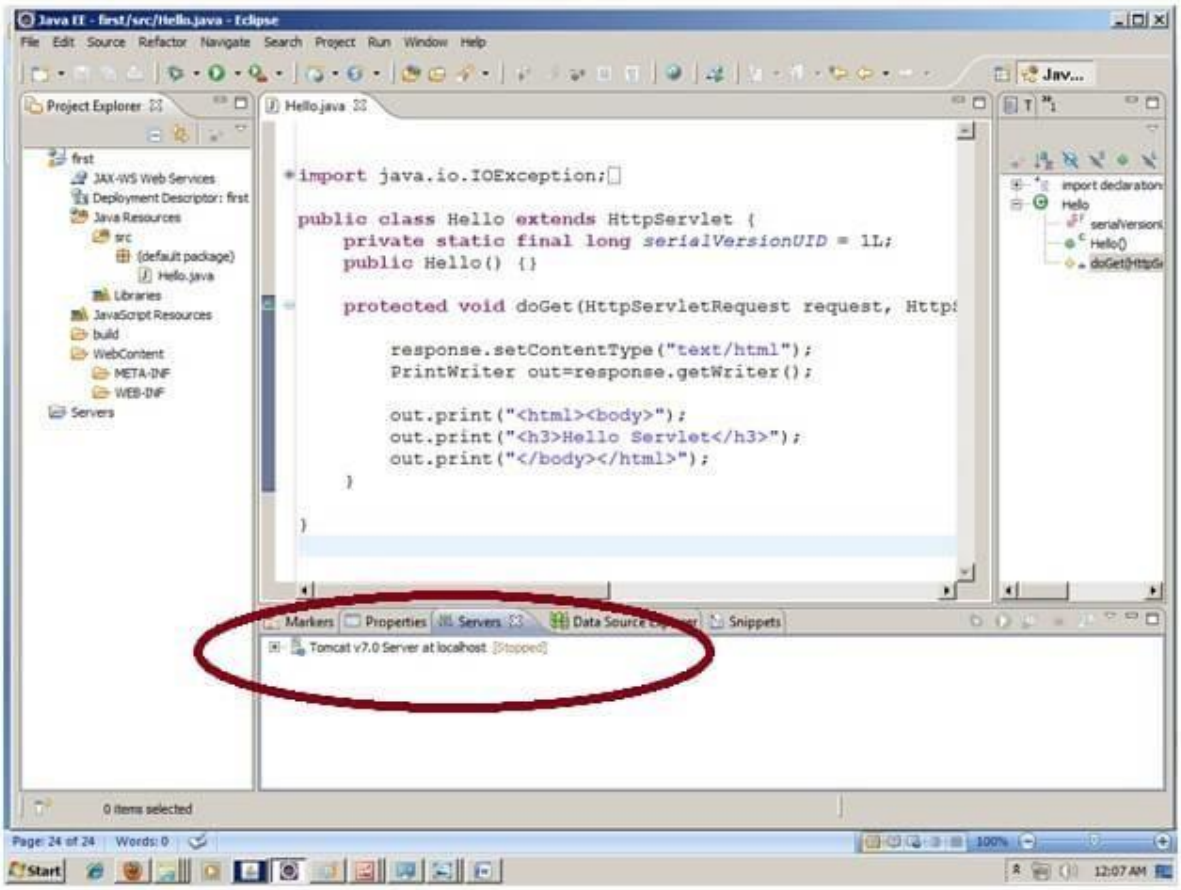

New Server
Define a New Server
Choose the type of server to create
Download additional server adapters
Select the server type:
type filter text
Tomcat v4.0 Server
Tomcat v4.1 Server
Tomcat v5.0 Server
Tomcat v6.0 Server
Tomcat v7.0 Server
Server's host name:
localhost
Server name:
Tomcat v7.0 Server at localhost
Next >
Finish
Cancel
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
0 items selected


New Server
Tomcat Server
Specify the installation directory
Name:
Apache Tomcat v7.0
Tomcat installation directory:
Browse...
JRE:
Workbench default JRE
Installed JREs...
< Back
Next >
Finish
Cancel
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
0 items selected


New Server
Tomcat Server
Specify the installation directory
Name:
Apache Tomcat v7.0
Tomcat installation directory:
Browse...
JRE:
Browse For Folder
Select Tomcat installation directory.
Desktop
Libraries
Local Disk (C:)
apache-tomcat-7.0.23
apache-tomcat-7.0.23
bin
Folder:
Make New Folder
OK
Cancel
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
0 items selected



Now tomcat7 server has been configured in eclipse IDE.


Java EE - first/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Project Explorer
first
JAX-WS Web Services
Deployment Descriptor: first
Java Resources
src
(default package)
Hello.java
Libraries
JavaScript Resources
build
WebContent
META-INF
WEB-INF
Servers
Hello.java
import java.io.IOException;
public class Hello extends HttpServlet {
private static final long serialVersionUID = 1L;
public Hello() {}
protected void doGet(HttpServletRequest request, Https
response.setContentType("text/html");
PrintWriter out=response.getWriter();
out.print("<html><body>");
out.print("<h3>Hello Servlet</h3>");
out.print("</body></html>");
}
}
Markers
Properties
Servers
Snippets
Tomcat v7.0 Server at localhost [Stopped]
0 items selected
Page: 24 of 24
Words: 0
100%
Start
12:07 AM

JAVA SERVER PAGES

Introduction to JSP:

JavaServer Pages (JSP) is a technology for developing Webpages that supports dynamic content. This helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>.

A JavaServer Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands.

Using JSP, you can collect input from users through Webpage forms, present records from a database or another source, and create Webpages dynamically.

JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages, and sharing information between requests, pages etc.

Advantages of JSP:

Following are the some advantages of JSP

1. JSP is useful for server side programming.
2. JSP can be used along with servlets. Hence business logic for any application can be devoloped using JSP
3. Dynamic contents can be handled using JSP because JSP allows scripting and element based programming
4. JSP allows creating and using our own custom tag libraries.Hence any application specific requirements can be satisfied using custom tag libraries.This helps the developer to devolop any kind of application.
5. JSP is a specification not product.HenceDevolopers can devolop variety of applications and add upto performance and quality of software.
6. JSP is an essential component of J2EE. Hence using JSP it is possisble to devolop simple as well as complex applications.

Problem with Servlet:

- In one and only one class ,the servlet alone has to do various tasks such as
    - Acceptance of request
    - Processing of request
    - Handling of business logic
    - Generation of response

Here there are some problems associated with servlets

1. For developing servlet based application,knowledge of java as well as HTML code is necessary
2. While developing any web based application if a look and feel pf web based application needs to be changed then entire code needs to be changed and recompiled.
3. There are some **web page tools** available using which the developer can develop web based applications. But the servlets do not support such tools. Even if such tools are used for servlet, we need to change embedded HTML code manually, which is time consuming, error prone and complicated process.

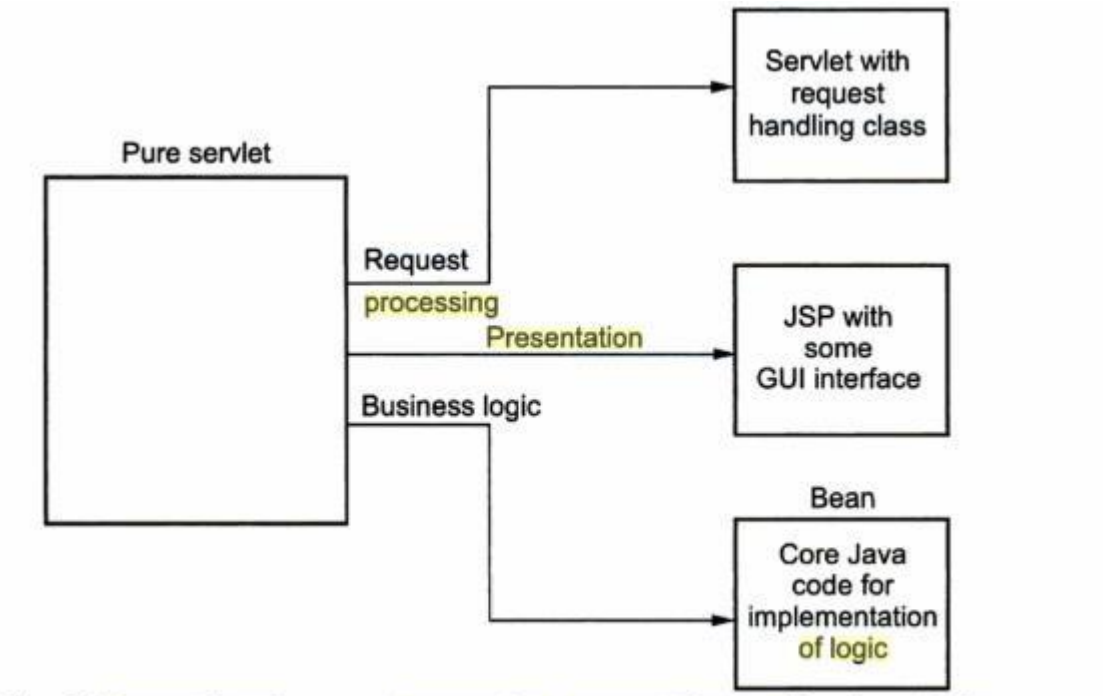


Fig. 8.1 Separation of request processing, presentation and business logic

- If some modifications are required to perform for JSP based application then required changes can be performed independently. For instance business logic can be changed with out changing the user interface.

The Anatomy of JSP page

JSP page s a simple web page which contains **the JSP elements** and **Template text .** The template text can be scripting code such as HTML,XML ,WML or a simple plain text.Various JSP elements are responsible for generating dynamic contents.

JSP code:

```
<%@page language="java" contentType="text/html" %>
<html>
<head>
<title>Demo for Template Text</title>
</head>
<body bgcolor="gray">
<h1>abcdefgh</h1>
<h2>123456789</h2>
<li>qwerty</li>
<p>
<%out.print("JSP equal to HTML and JAVA"); %>
</p>
<h3>Today Date is:
<%= new Date.toString() %>
</h3>
</body>
</html>
```

←------ **JSP ELEMENT**

←--------- **Template text**

←----------**JSP ELEMENT**

← -------**Template text**

←--------- **JSP ELEMENT**

←---------- **Template Text**

JSP PROCESSSING:

JSP pages can be processed using JSP container only. Following are the steps that are need to be followed while processing the request for JSP page.

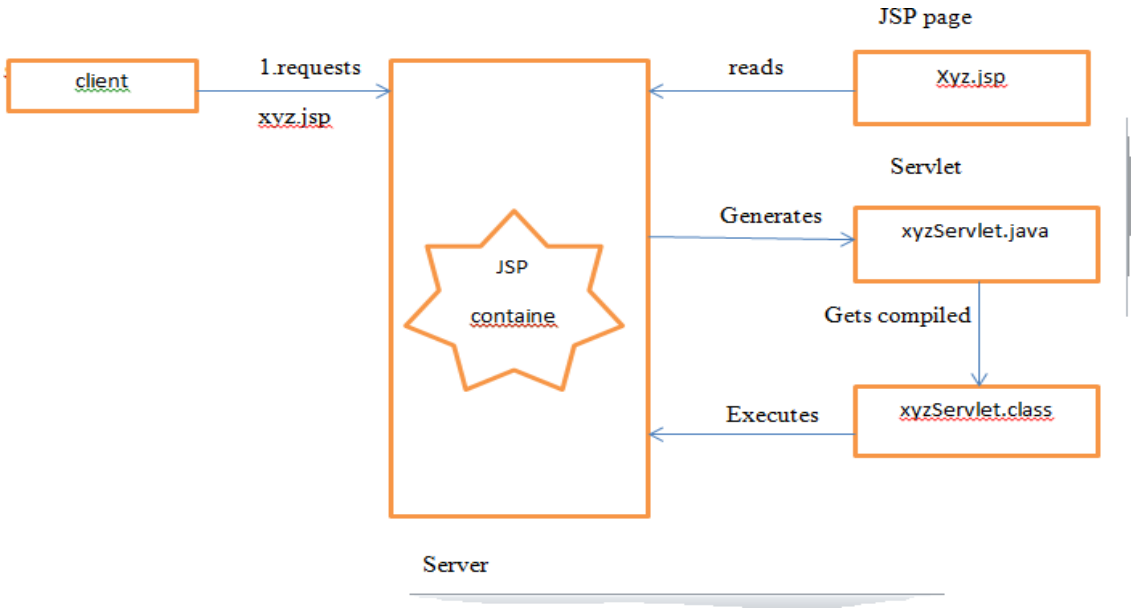
1. Client makes a request for required JSP page to server. The server must have JSP container so that JSP request can be processed.For instance let the client makes request for xyz.jsp page
2. On receiving this request the JSP container searches and then reads the desired JSP page. Then this JSP page is straight away converted to corresponding **Servlet**. Basically any JSP page is a combination of Template text and JSP element.every template text is translated into corresponding **println** statement.

For instance:



3. This servlet is compiled to generate servlet class file. Using this class the response can be generated. This phase is called **request processing phase**
4. The JSP container thus executes the servlet class file.

5. A requested page is then returned to the client as



JSP Application design with MVC

The design model of JSP application is called MVC model. The MVC stands for Model-View-Controller. The basic idea in MVC design is to separate out design logic into three parts-modelling, viewing & controlling.

Any server application is classified in three parts such as business logic, Presentation & request processing.

Business logic means coding logic applied for manipulation of application data. **Presentation** refers to the code written for look and feel of the web page.(ex: background color,fontstyle,font style, placing of controls such as text boxes,command buttons and so on. The request processing is nothing but combination of **business logic** and **presentation .**The request processing is always done in order to generate the response.

According to MVC design model, the model corresponds to business logic, view corresponds to presentation & Controller corresponds to request processing.

The advantages of MVC are:

- Easy to maintain
- Easy to extend
- Easy to test
- Navigation control is centralized

JSP Application Development:

GENERATING DYNAMIC CONTENT:

Dynamic contents means the contents that get changed based on the user inputs or states of external system or on some runtime conditions. JSP helps in handling such conditions. There are various ways by which JSP handles the dynamic contents such as use of:

1. Directive elements
2. Java Beans
3. Scripting elements
4. Standard tag libraries
5. Standard and custom actions
6. Expression language.

Directive Elements:

Consider the example

Ex: datedemo.jsp

```
<%@page language=”java” contentType=”text/html” %>
<%@page import=”java.util.*”%>
<html>
<body>Today’s Date is:
<%=new Date().toString() %>
</body>
</html>
```

The first line in the JSP is JSP **directive element**. The directives are used to specify the attributes of page. Using directive we can even import some packages. We have used the page directive . The language=”java” tells that the page is using java language.

- Directive elements are used to specify attributes of page. syntax is
  <%@ directivename attr1=”value1” attr2=”value2” %>
- There are three basic directives **page**, **include**&**taglib**. The Page is used to provide the information about the page. The include directive is used to provide the name of the file that can be included in JSP. For example if we want to include a file named abc.jsp then we can write the statement as

<%@ include file=”abc.jsp” %>

- The directive taglib is used to specify the custom tags. The custom tags are those tags that can be created by the user.

- Attribute values can be-
  - **language** specifies the language used by JSP
    <%@page language=”java”
  - **contentType** specifies the MIME type. Hence we write it as
    <%@page language=”java” contentType=”text/html” %>
  - **import**specifies the class which we want to extend. Hence we write it as
    <%@page language=”java” import=”MyTest.Mytestclass” %>

- **errorPage**is used to report the unhandled exceptions. We can write it as <%@ page language=”java”errorPage=”MyBugs.jsp” %>
- **session**specifies the session data. The default value is **true**when the session data is available to JSP this value is true otherwise it is false. It can be written as <%@ page language=”java” session=”true” %>

What is template text?

Everything other than a JSP statement in the JSP is called a template text. For example

Templatetext.jsp

```
<%@ page language=”java” contentType=”text/html” %>
<html>
<head>
<title>Demo for Template text</title>
</head>
<body bgcolor=”gray”>
<h1>abcdefgh</h1>
<h2>123456789</h2>
<li>qwerty</li>
<p>
<%out.print(“JSP equal to HTML and JAVA”); %>
</p>
<h3>Today Date is:
<%= new Date.toString() %>
</h3>
</body>
</html>
```

In the above code boldfaced code is template text.

Action Elements:

The elements which are used to acess the bean in a page is called JSP action elements. The JSP actions are XML tags that can be used in JSP page. JSP action elements get executed when JSP makes some request. The action elements are used to handle the dynamic actions that take place at runtime.

JSP defines three types of actions

1. Standard action
2. Custom action
3. JSP standard tag library(JSTL)

Standard action

Standard actions are those actions that can be defined by the JSP specification itself. Following is a list of standard actions.

Action Element	Description
<jsp:forward>	This tag helps in forwarding the current request to servlet or to JSP page.
<jsp:usebean>	This tag is used to instantiate the object of java bean
<jsp:setProperty>	This tag is used to set the property value for java bean
<jsp:getProperty>	This tag is used to get the property value from java bean
<jsp:include>	This tag works as a sub-routine. It includes the response from the servlet or JSP when the request is being processed.
<jsp:param>	For adding the specific parameter to the request this tag is used.
<jsp:plugin>	This tag is used to generate the HTML code and to embed the applet into it.
<jsp:attribute>	This tag sets the value of action attribute.
<jsp:element>	This tag generates the XML elements dynamically.
<jsp:text>	This tag is used to handle template text. When JSP pages are written as XML documents then this tag is used.
<jsp:body>	This tag is used to set the body element.

a) Include action:

Using the<jsp:include> action we can include different files in current jsp pages. In the following example we have created one main JSP page Include_action.jsp which will include one html file test_include.html and one jsp page test_jsp.jsp . The test_include.html is showing the static contents and test_jsp.jsp is showing the dynamic contents

Include_action.jsp

```
<html>
<head>
<title>Include Demo</title>
</head>
<body>
<h1>this page contains include action</h1>
<jsp:include page="test_include.html"/>
<jsp:include page="test_jsp.jsp" />
</body>
</html>
```

Test_Include.html

```
<html>
<body>
Hello How are you?
</body>
</html>
```

test_jsp.jsp

```
<html>
<body>
<br>Time is<%=new java.util.Date().getHours()%>
</body>
</html>
```

b) Forward action

using forward action we can forward current request to corresponding JSP page or servlet. Here is a simple demonstration in which we have created two web pages the first one is main JSP page named first.jsp in which we are using JSP action forward and forwarding current request to an HTML page which is named Second.html

first.jsp

```
<html>
 <head>
    <title>Forward action demo</title>
 </head>
<body>
<h1>This page contains forward action</h1>
<jsp:forward page=”Second.html” />
</body>
</html>
```

Second.html

```
<html>
<body>
<strong>you are forwarded to second page</strong>
</body>
</html>
```

c) Plugin action

The plugin action is used to embed an applet or a bean.

ColorDemo.java

```
Import java.awt.*;
Import java.applet.*;
Public class ColorDemo extends Applet
{
Public void paint(Graphics g)
{
setBackground(Color.cyan);
setForeground(Color.red);
```

```
g.drawString("colourful applet",50,30);
ColornewColor=new Color(255,255,0);
g.setColor(newColor);
g.drawString("colourful applet",50,70);
}
}
```

When we compile ColorDemo.class file is generated. Now create a folder named**PluginAction**in your tomcat directory in the subfolder webapps/jsp-examples copy the class file in PluginAction directory

Plugin_action.jsp

```
<jsp:plugin type="applet" code="ColorDemo" codebase="/jsp-examples/PluginAction" width="190" height="190">

<jsp:fallback><strong>sorry applet cannot be loaded</strong></jsp:fallback>

</jsp:plugin>
```

Custom action:

Custom action allows us to create user-defined tag. The programmer can hide the java class code

And can acess that code from the JSP page

Advantages of Custom Tags

The key advantages of Custom tags are as follows:

1. **Eliminates the need of srciptlet tag** The custom tags eliminates the need of scriptlet tag which is considered bad programming approach in JSP.
2. **Separation of business logic from JSP** The custom tags separate the the business logic from the JSP page so that it may be easy to maintain.
3. **Reusability** The custom tags makes the possibility to reuse the same business logic again and again.
4. **Readability** The use of custom tags enhances the readability of the used codes in a JSP page.The readability increases by encapsulating the use of the Java codes in a JSP page.

Maintainability this feature enables the user to reduce the duplicity of codes present in a JSP page..

For creating the custom tag the following steps must be adopted.

Step 1: Creation of Tag Handler

Step 2: creation of tag library descriptor

Step 3: creation of JSP with registration and use of tags declared in TLD

Let us understand these steps with illustrating example.

Step 1: The tag handler is a java class which responds to the events raised by the container. These events get raised when the custom tag is encountered. Every tag handler class must extend the **TagSupport .**This class is supported by JSPAPI. Now we will create a .java file in

C:\your_tomcat_directory\WEB-INF\classes create a folder say **CustomTag** and store firsttag.java

firstTag.java

```
package CustomTag;

import java.io.*;

import javax.servlet.http.*;

import javax.servlet.jsp.*;

import javax.servlet.jsp.tagext.*;

import java.util.*;

public class firsttag extends TagSupport

{

  public int doStartTag() throws JspException

   {

     try

      {

          JspWriter out = pageContext.getOut();

         Out.println(“Hello this is my first custom tag”);

     }

  catch(Exception e)

    {

      throw  new JspException(e.getMessage());

    }

 return EVAL_PAGE;

 }

}
```

In the above Java program out class is extended from the interface TagSupport which has following methods

- **doStartTag()** for processing the **start** tag
- **doEndTag()** for processing the **end** tag
- **getId()** for **getting the value of Id** attribute
- **setId()** for **setting the value of Id** attribute
- **getParent()** for **returning the parent tag instance**.
  and so on…

The doStartTag method returns EVAL_PAGE which means continue evaluating the page. This is valid return value. This method throws an exception JspException. In this java program we are simply displaying the message "Hello!!! This is my first custom Tag".

Field Name	Description
public static int EVAL_BODY_INCLUDE	**it evaluates the body content.**
public static int EVAL_PAGE	**it evaluates the JSP page content after the custom tag.**
public static int SKIP_BODY	**it skips the body content of the tag.**
public static int SKIP_PAGE	**it skips the JSP page content after the custom tag.**

Step 2: now we will create a .tld file. For that purpose switch to the path

C:\your_tomcat_directory\WEB-INF and write the following file named as **MyTag.tld**

MyTag.tld

<?xml version ="1.0" encoding = "ISO-8859-1"?>

<taglib>

<tlib-version>1.0</tlib-version>

<jsp-version>2.0</jsp-version>

<short-name>MyTag</short-name>

<tag>

<name>MyFirstTag</name> ←**-- tag name which will be referenced in jsp**

<tag-class> CustomTag.firstTag</tag-class> ←**--java class created in step 1**

<body-content>empty</body-content>

</tag>

</taglib>

Step 3: Now we have to register custom tag from MyTag.tld file with JSP page. We can do this using **taglib** directive now go to path C:\your_tomcat_directory\WEB-INF and create folder say **CustTagDemo and** create following jsp page in that folder

custtag.jsp

<% @ taglib uri **= "/WEB-INF/Mytag.tld"** prefix = "my">

<html>

<body>

<h1>welcome following sentence is written using custom tag</h1>

<**my:MyFirstTag**/>

</body>

</html>

3. Introduction to JSTL

JSTL stands for JSP Standard Tag Library. J2EE is typically used for server side programming using java and JSTL a component of J2EE web application devolopment platform. The JSTL makes use of JSP along with a tag library. This tag library is useful for performing some common tasks such as condition execution,data procession and so on. JSTL allows the programmer to embedded the logic in JSP page using JAVA code. The tag library makes use of some standard set of tags. There are five categories in which JSTL tags are defined.

1. core
2. xml
3. sql
4. formatting
5. functions

To use JSTL you must have the JSP(1.2 or higher version)container. The default JSP container is Tomcat web server. Hence JSTL comes along with higher version of Tomcat.for instance if you load Tomcat 5.X then the required jar files of JSTL are already installed in Tomcat's lib directory. If you have got lower version of tomcat you need to copy these jar files in Tomcat directory. To get ensured with support of JSTL,just open the WEB-INF directory and check if **jstl.jar** and **standard.jar** files are there or not.

The best option to install JSTL is to download the JWSDP package. This package is **J**ava **W**eb **S**ervice **D**evolopment's pack and you can get it fom the URL

http://java.sun.com/products/jsp/jstl

To install JSTL in tomcat just copy the **jstl.jar** and **standard.jar** files in the lib folder of

WEB-INF directory to your Tomcat.

Core Tag:

1. **<c:out>**
2. **<c:set>**
3. **<c:if>**
4. **<c:choose>**
5. **<c:when>**
6. **<c:otherwise>**
7. **<c:forEach>**
8. **<c:forTokens>**
9. **<c:url>**
10. **<c:redirect>**
11. **<c:param>**
12. **<c:import>**
13. **<c:remove>**

1.c-out.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Tag Example</title>
</head>
<body>
<c:out value="${'Welcome to javaTpoint'}"/>
</body>
</html>
```

Output:

Welcome to javaTpoint

2. c-set.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="Income" scope="session" value="${4000*4}"/>
<c:out value="${Income}"/>
</body>
</html>
```

Output:

16000

3.c-if.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="income" scope="session" value="${4000*4}"/>
<c:if test="${income > 8000}">
  <p>My income is: <c:out value="${income}"/><p>
</c:if>
</body>
</html>
```

Output:

My income is:16000

4. c-choose.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:set var="income" scope="session" value="${4000*4}"/>
<p>Your income is : <c:out value="${income}"/></p>
<c:choose>
   <c:when test="${income <= 1000}">
      Income is not good.
   </c:when>
   <c:when test="${income > 10000}">
       Income is very good.
```

```
   </c:when>
   <c:otherwise>
      Income is undetermined...
   </c:otherwise>
</c:choose>
</body>
</html>
```

Output:

Income is very good.

6. **c-forEach.jsp**

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
</head>
<body>
<c:forEach var="j" begin="1" end="3">
  Item <c:out value="${j}"/><p>
</c:forEach>
</body>
</html>
```

Output:

Item 1

Item 2

Item 3

7. c-forTokens.jp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Core Tag Example</title>
```

```
</head>
<body>
<c:forTokens items="Rahul-Nakul-Rajesh" delims="-" var="name">
   <c:out value="${name}"/><p>
</c:forTokens>
</body>
</html>
```

Output:

Rahul

Nakul

Rajesh

9. c-remove.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
 <title>Core Tag Example</title>
</head>
<body>
<c:set var="income" scope="session" value="${4000*4}"/>
<p>Before Remove Value is: <c:out value="${income}"/></p>
<c:remove var="income"/>
<p>After Remove Value is: <c:out value="${income}"/></p>
</body>
</html>
```

Output:

Before Remove Value is: 16000

After Remove Value is:

IMPLICIT JSP OBJECTS:

The objects which we access in our JSP page, without any explicit declaration are called as implicit objects. Implicit objects are exposed by the JSP container and can be seen in the generated servlet of a JSP page.
Implicit objects are advantageous because, they don't require the JSP authors to explicitly declare and initialize a few of the servlet objects, which is a difficult task.

Object	**Class/Interface**	**Description**
application	javax.servlet.ServletConfig	Represents the context for the JSP page servlet
session	javax.servlet.http.HttpSession	This variable is used to access current client's session.
request	javax.servlet.http.HttpServletRequest	It provides the method for accessing the information made by current request.
Response	javax.servlet.http.HttpServletResponse	It provides the methods related to adding cookies, session, setting headers.
PageContent	javax.servlet.jsp.PageContent	It provides access to several JSP attributes.
Page	Java.lang.object	This variable is assigned to instance of JSP implementation class.
Out	Javax.Servlet.jsp.JspWriter	It provides methods related to I/O.
Exception	Java.lang.Throwable	This object is used for handling error pages and contain information about runtime errors.
config	Javax.servlet.ServletConfig	It helps in passing the information to the servlet or JSP page during initialization.

Scripting elements:

Scripting allows us to embed java code in a JSP page and scripting elements (scriptlets, declaration and expressions) allows us to do scripting. Scripting elements are executed at request processing time and are used for a variety of purposes including manipulation of objects, perform calculation on runtime variables values, etc.

Conditional processing:

Sometimes to built the complex logic we require conditional statements. We can embed JAVA conditional statements in JSP. This task can be done by scriptlets. The syntax for scriptlets is:

<% any java code %>

Sample program for conditional processing:

```
<%@ page language=”java” ContentType=”text/html” %>
<html>
<head><title>Introduction to scriptlet</title></head>
<body>
<h1>
<% out.println(“JSP is very interesting”); %>
</h1>
</body>
</html>
```

Web.xml file:

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

Displaying Values Using An Expression To Set An Attribute:

The JSP expressions are used to insert java values directly into the output. The syntax of using expression is as follows:

<%= Expression code %>

This expression gets evaluated at runtime and then the result will get displayed on the web browser. Thus tags <%= and %> is used.

Sample program for using an expression:

Scripting.jsp file:

```
<%@ page import="java.util.Date"%>
<html>
<head><title>Scripting elements example</title></head>
<body>
<%! int count=0; %>
<% count++;
out.println("You are accessing this page for "+count+ "times");
out.println(new Date().toString());
%>
<%= count %>
</body>
</html>
```

Web.xml file:

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

Directory structure:

There are some predefined variables that can be used while defining the expressions. These variables can be used along with the implicit object.

Declaring Variables And Methods:

JSP allow us to use the user defined variables and methods using declaration. The variables and methods can be used along with scriptlets and expression. The syntax for declaration is as follows:

<%! Any java code %>

Sample program for declaring variables and methods:

```
<%@ page language="java" contentType="text/html" %>
<% String msg="Hello"; %>
<%! public String MyFunction(String msg)
{
returnmsg;
}
%>
<html>
<head>
<title>Use of Method</title>
</head>
<body>
<%
out.println("Before function call:"+msg); %>
<br>
After function call: <%= MyFunction("Web Technologies") %>
</body>
</html>
```

Web.xml file:

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

UNIT-III

Error Handling and Debugging:

While developing any application we come across several errors.We may get syntactical errors during development of JSP pages. When we are writing JSP code, a programmer may leave a coding errors which can occur at any part of the code. You can have following type of errors in your JSP code:

•**Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

•**Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.

•**Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Error Handling:

Using Exception Object:

•The exception object is an instance of a subclass of Throwable (e.g., java.lang. NullPointerException) and is only available in error pages. Following is the list of important medthods available in the Throwable class.

1 **public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

2 **public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.

3 **public String toString()** Returns the name of the class concatenated with the result of getMessage()

4 **public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream. 5 **public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.

6 **public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

UNIT-III

JSP gives you an option to specify Error Page for each JSP. Whenever the page throws an exception, the JSP container automatically invokes the error page.

•Following is an example to specifiy an error page for a main.jsp. To set up an error page, use the

Main.jsp

<%@ page errorPage="xxx" %> directive.

```
<%@ page errorPage="ShowError.jsp" %>
<html> <head> <title>Error Handling Example</title> </head>
<body>
<% // Throw an exception to invoke the error page
int x = 1;
if (x == 1)
{
throw new RuntimeException("Error condition!!!");
}
%>
</body> </html>
```

Now you would have to write one Error Handling JSP ShowError.jsp, which is given below.

•Notice that the error-handling page includes the directive <%@ page isErrorPage="true" %>. This directive causes the JSP compiler to generate the exception instance variable.

ShowError.jsp

```
<%@ page isErrorPage="true" %>
<html> <head> <title>Show Error Page</title> </head>
<body> <h1>Opps...</h1>
<p>Sorry, an error occurred.</p> <p>Here is the exception stack trace: </p> <pre>
<% exception.printStackTrace(response.getWriter()); %>
</pre> </body> </html>
```

Now try to access main.jsp, it should generate something as follows:

Out Put

java.lang.RuntimeException: Error condition!!! ..... Opps... Sorry, an error occurred. Here is the exception stack trace:

Example:

```
<%@ page language="java"contentType="text/html">
<%@ page import="java.util.*"'%>
<html>
<body>
Today's date is:
<%= new Date().toString() %>
</body>
</html>
```

In the above code there is starting tag <% but the ending tag %> is missing in first line which returns an error. HTTP returns status code 500.This value indicates that some internal error has occurred and therefore the request cannot be fulfilled.

Debugging JSP Action:

After fixing all the syntactical errors, still the application does not work properly due to local errors. Logical errors cannot be highlighted by the tomcat explicitly. When an application does not work according to our expectations then we say that logical errors are there. To get rid of such errors we need to debug our JSP page.

There are debuggers for JSP pages. Visual age is a debugging tool from IBM which allows us to trace the program line by line. For tracing the JSP page without any debugging tool we can use print statements which print the values on browser. The runtime errors can be handled by using exceptions.

JSP Implicit Objects:

There are 9 jsp implicit objects. These objects are created by the web container that are available to all the jsp pages and the developer can call them directly without being explicitly declared. JSP Implicit Objects are also called pre-defined variables.

Following table lists out the nine Implicit Objects that JSP supports

S.No.	Object	Description
1	**request**	This is the **HttpServletRequest** object associated with the request.

2	**response**	This is the **HttpServletResponse** object associated with the response to the client.
3	**out**	This is the **PrintWriter** object used to send output to the client.
4	**session**	This is the **HttpSession** object associated with the request.
5	**application**	This is the **ServletContext** object associated with the application context.
6	**config**	This is the **ServletConfig** object associated with the page.
7	**pageContext**	This encapsulates use of server-specific features like higher performance **JspWriters**.
8	**page**	This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
9	**Exception**	The **Exception** object allows the exception data to be accessed by designated JSP.

1. The request Object:

The request object is an instance of a **javax.servlet.http.HttpServletRequest** object. Each time a client requests a page the JSP engine creates a new object to represent that request.

The request object provides methods to get the HTTP header information including form data, cookies, HTTP methods etc.

Example:

req.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

req.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

2. The response Object:

The response object is an instance of a **javax.servlet.http.HttpServletResponse** object. Just as the server creates the request object, it also creates an object to represent the response to the client.

The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes, etc.

res.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

res.jsp

```
<%
response.sendRedirect("http://www.google.com");
%>
```

The out Object:

The out implicit object is an instance of a **javax.servlet.jsp.JspWriter** object and is used to send content in a response.

The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the **buffered = 'false'** attribute of the page directive.

The JspWriter object contains most of the same methods as the **java.io.PrintWriter** class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws **IOExceptions**.

Following table lists out the important methods that we will use to write **boolean char, int, double, object, String**, etc.

S.No.	Method & Description
1	**out.print(dataType dt)** Print a data type value

2	**out.println(dataType dt)** Print a data type value then terminate the line with new line character.
3	**out.flush()** Flush the stream.

Example:

In this example we are simply displaying date and time.

outimpl.jsp

```
<html>
<body>
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
</body>
</html>
```

The session Object:

The session object is an instance of **javax.servlet.http.HttpSession** and behaves exactly the same way that session objects behave under Java Servlets.The session object is used to track client session between client requests.

Example:

ssn.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

ssn.jsp

```
<html>
<body>
<%

String name=request.getParameter("uname");
out.print("Welcome "+name);
session.setAttribute("user",name);
<a href="second.jsp">second jsp page</a>
```

```
%>
</body>  </html>
```

second.jsp

```
<html>
<body>
<%
String name=(String)session.getAttribute("user");
out.print("Hello "+name);
%>
</body>
</html>
```

The application Object:

The application object is direct wrapper around the **ServletContext** object for the generated Servlet and in reality an instance of a **javax.servlet.ServletContext** object.

This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the **jspDestroy()** method.By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

Example:

appl.html

```
<form action="welcome">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

web.xml file

```
<web-app>
<servlet>
<servlet-name>sonoojaiswal</servlet-name>
<jsp-file>/welcome.jsp</jsp-file>
</servlet>
  <servlet-mapping>
<servlet-name>sonoojaiswal</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
  <context-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</context-param>
  </web-app>
```

applobj.jsp

```
<%
  out.print("Welcome "+request.getParameter("uname"));
String driver=application.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

The config Object

The config object is an instantiation of **javax.servlet.ServletConfig** and is a direct wrapper around the **ServletConfig** object for the generated servlet. This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc. This object can be used to get initialization parameter for a particular JSP page. The config object is created by the web container for each jsp page.

The following **config** method is the only one you might ever use, and its usage is trivial config.getServletName();

This returns the servlet name, which is the string contained in the **<servlet-name>** element defined in the **WEB-INF\web.xml** file.

Generally, it is used to get initialization parameter from the web.xml file.Example

con.html

```
<form action="welcome">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

web.xml file

```
<web-app>
  <servlet>
<servlet-name>sonoojaiswal</servlet-name>
<jsp-file>/welcome.jsp</jsp-file>
<init-param>
<param-name>dname</param-name>
<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>
</init-param>
</servlet>
  <servlet-mapping>
<servlet-name>sonoojaiswal</servlet-name>
<url-pattern>/welcome</url-pattern>
</servlet-mapping>
</web-app>
```

welcome.jsp

```
<%
out.print("Welcome "+request.getParameter("uname"));
String driver=config.getInitParameter("dname");
out.print("driver name is="+driver);
```

```
%>
```

The pageContext Object

The pageContext object is an instance of a **javax.servlet.jsp.PageContext** class. The pageContext object is used to represent the entire JSP page. This object is intended as a means to access information about the page while avoiding most of the implementation details. This object stores references to the request and response objects for each request. The **application, config, session**, and out objects are derived by accessing attributes of this object. The pageContext object also contains information about the directives issued to the JSP page, including the buffering information, the errorPageURL, and page scope.

The PageContext class defines several fields, including **PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE,** and **APPLICATION_SCOPE**, which identify the four scopes. It also supports more than 40 methods, about half of which are inherited from the **javax.servlet.jsp.JspContext class**.

One of the important methods is **removeAttribute**. This method accepts either one or two arguments. For example, **pageContext.removeAttribute ("attrName")** removes the attribute from all scopes, while the following code only removes it from the page scope –

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

Example

pagcon.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
</body>
</html>
```

pagcon.jsp

```
<html>
<body>
<%
String name=request.getParameter("uname");
out.print("Welcome "+name);

pageContext.setAttribute("user",name,PageContext.SESSION_SCOPE);
<a href="second.jsp">second jsp page</a>
%>
</body>
</html>
```

second.jsp

```
<html>
<body>
<%
  String name=(String)pageContext.getAttribute("user",PageContext.SESSION_SCOPE);
out.print("Hello "+name);
  %>
</body>
</html>
```

The page Object

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The page object is really a direct synonym for the **this** object.

Object page=this;

For using this object it must be cast to Servlet type.For example:

<% (HttpServlet)page.log("message"); %>

Since, it is of type Object it is less used because you can use this object directly in jsp.For example:

<% this.log("message"); %>

The exception Object

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

```
<%@ page isErrorPage="true" %>
<html>
<body>
Sorry following exception occured:<%= exception %>
  </body>
</html>
```

index.jsp

```
<form action="process.jsp">
No1:<input type="text" name="n1" /><br/><br/>
No1:<input type="text" name="n2" /><br/><br/>
<input type="submit" value="divide"/>
</form>
```

process.jsp

```
<%@ page errorPage="error.jsp" %>
<%
String num1=request.getParameter("n1");
String num2=request.getParameter("n2");
int a=Integer.parseInt(num1);
```

```
        int b=Integer.parseInt(num2);
        int c=a/b;
        out.print("division of numbers is: "+c);
         %>
```

error.jsp

```
        <%@ page isErrorPage="true" %>
        <h3>Sorry an exception occured!</h3>
        Exception is: <%= exception %>
```

Sharing Data between JSP Pages, Requests, and Users

•Any real application consists of more than a single page, and multiple pages often need access to the same information and server-side resources.

•When multiple pages are used to process the same request, for instance one page that retrieves the data the user asked for and another that displays it, there must be a way to pass data from one page to another.

•In an application in which the user is asked to provide information in multiple steps, such as an online shopping application, there must be a way to collect the information received with each request and get access to the complete set when the user is ready.

•Other information and resources need to be shared among multiple pages, requests, and all users. Examples are information about currently logged-in users, database connection pool objects, and cache objects to avoid frequent database lookups.

To share Data

Need to know how to

1) Structure a JSP application
2) Pass control from one JSP page to another
3) Pass data from one JSP to another

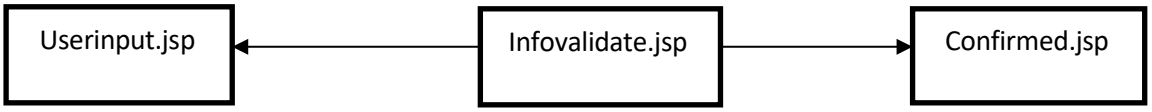
Example

•A shopping application that accepts a user's payment details on a form, validates the details and displays a confirmation page or error depending upon user input

•Various ways that JSP pages for collection of payment details can be structured. Can even have everything on a single JSP page (messy for anything but simplest apps)

•Have used 3 JSP pages: - payment details on a form (userinput.jsp) - validation of the details (validateinfo.jsp) - display of a confirmation page (confirmed.jsp) or error (userinput.jsp) depending upon user input .

Structure of JSP pages: example



Userinput.jsp:

Displays a form into which user enters their details.

infovalidate.jsp

Validates the info provided in userinput.jsp and passes back to userinput.jsp if there's an error, or confirmed.jsp if info is valid.

confirmed.jsp

Displays a message to user to confirm payment made.

Passing Control and Data between JSP Pages:

A web based application is a client/server application that requires to handle multiple pages. Multiple pages may access same kind of information where the data consistency must be provided. When particular request is processing some data when that data is navigating from one page to another it should remain the same. If at all request makes some desired modifications on that data then those modifications should be reflected over all the pages that are using that data.

For a web application JSP technology uses Model View Controller architecture model. The logic is classified in 3 partitions.

- Request Processing
- Business Logic
- Presentation

For example consider mail login page. First access the site to login which is view page. After entering username and password it will be validated for authentication which is referred as process page. If invalid you will remain in login page otherwise the inbox page will be displayed again this is a view page. This scenario is as follows.

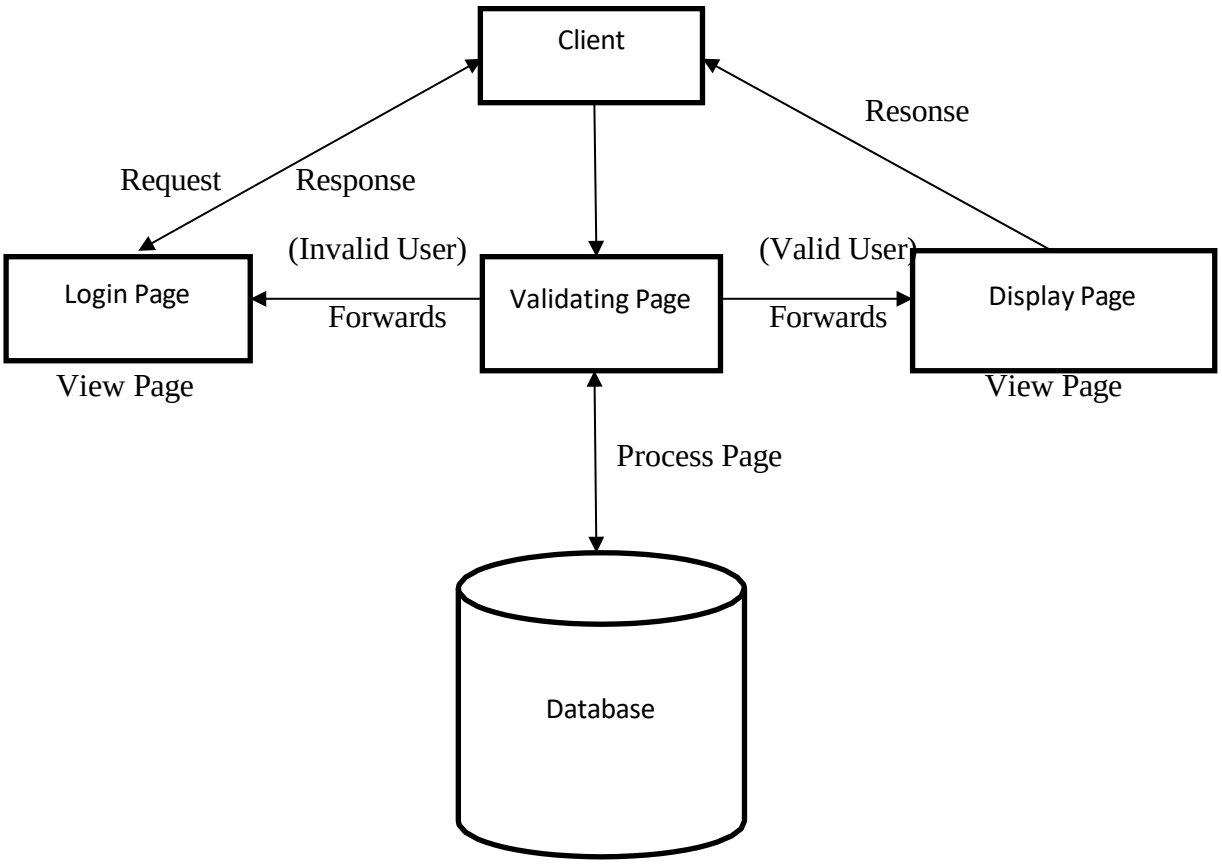


Fig. Model View Architecture

Passing Control between Pages:

Here we are passing the parameters along with forward request. For passing parameters we are using <jsp:param> action tag. In this example we are passing 4 parameters along with <jsp:forward> and later we are displaying them on the forwarded page. In order to fetch the parameters on display.jsp page we are using getParameter method of request implicit object.

```
<html>
 <head>
<title>JSP forward example with parameters</title>
</head>
<body>
<jsp:forward page="display.jsp">
<jsp:param name="name" value=Ria" />
<jsp:param name="site" value="Blogspot.com" />
<jsp:param name="tutorialname" value="jsp forward action" />
<jsp:param name="reqcamefrom" value="pctrl.jsp" />
</jsp:forward>
```

```
</body>
</html>
```

display.jsp

```
<html>
<head>
<title>Display Page</title>
</head>
<body>
<h2>Hello this is a display.jsp Page</h2>
My name is: <%=request.getParameter("name") %><br>
Website: <%=request.getParameter("site") %><br>
Topic: <%=request.getParameter("tutorialname") %><br>
Forward Request came from the page: <%=request.getParameter("reqcamefrom") %>
</body>
</html>
```

Passing Data between Jsp Pages:

In context of JSP/Servlets data can be accessed in 4 different scopes: page, request, application and session. Scope defines lifetime, availability of an object or data.

Page Scope: Objects can be accessed within that jsp page only.

Request Scope: Objects can be accessed by any page involved in the processing of request.

Session Scope: objects can be accessed for the entire browsing period by the pages origination from the same browser.

Application Scope: Objects can be accessed by any user of the web application and persist for the lifetime of server.

inputform.jsp

```
<html>
<head>
<title>Data Passing between JSP Pages</title>
</head>
<body>
<%
if(request.getParameter("UserName")=="")
{
%>
```

```
<jsp:forward page="ErrorPage.jsp"/>
<%
}
else if(request.getParameter("UserName")!=null)
{
%>
<jsp:forward page="MyJsp.jsp"/>
<%
}
else
{
%>
<form method="get" action="inputform.jsp">
<strong>Enter UserName:</strong>
<input type="text" name="UserName"><br>
<input type="submit" value="enter">
</form>
<%
}
%>
</body>
</html>
```

MyJsp.jsp

```
<html>
<head>
<title>Data Passing between JSP Pages</title>
</head>
<body>
Welcome<b>${param.UserName}</b>
</body>
</html>
```

ErrorPage.jsp

```
<html>
<body>
```

```
U have not given username
<a href="inputform.jsp">Click here</a>
</body>
</html>
```

Passing Data between Pages using Session object

In web applications user moves from one page to another.to keep track of user data and consistent throughout the web application session implicit object is used. Using session we can save the data of particular page.

Example:

First.jsp

```
<%@ page language="java"%>
<html>
<body>
<form method="post" action="second.jsp">
<strong>UserName:</strong>
<input type="text"name="UserName"><br>
<input type="password" name="Password"><br>
<input type="submit" value="submit">
</form>
</body>
</html>
```

In the below example using getParameter method of request object we are retrieving the parameters Username and Password.Using session object we cans et the values to the attributes.

second.jsp

```
<%@ page language="java"%>
<%
String un=request.getParameter("UserName");
String pwd=request.getParameter("Password");
session.setAttribute(UserName,un);
session.setAttribute(Password,pwd);
%>
<html>
<body>
```

```
<a href="third.jsp">Click here to view the next session</a>
</body>
</html>
```

third.jsp

```
<%@ page language="java"%>
<%
String un=(String)session.getAttribute("UserName");
String pwd=(String)session.getAttribute("Password");
%>
<html>
<body>
<strong>UserName is:<%= un %>
<strong>Password is:<%= pwd %>
</body>
</html>
```

Memory Usage Considerations:

Application and session scope objects consume memory at server side.

In application scope we can put the number of objects and by counting memory consumption of each object. So, total memory usage can be calculated as sum of memory consumptions made by all objects.

The memory usage for session scope is bit complex. Total no of objects in session scope depends on no of concurrent sessions. Hence to compute memory usage in current session we must know:

- The size of each object.
- How many current users are there?
- How long each session lasts?

If we store large number of objects then memory usage in session scope is heavy. Hence to put memory requirement under control follow these guidelines:

- Place only most needed objects in each session.
- The session must be ended explicitly i.e there must be logout.
- Timeout period for each session.

Scope of JSP Objects

The availability of a JSP object for use from a particular place of the application is defined as the scope of that JSP object. Every object created in a JSP page will have a scope. Object scope in JSP is segregated into four parts and they are page, request, session and application.

- **page**
  'page' scope means, the JSP object can be accessed only from within the same page where it was created. The default scope for JSP objects created using <jsp:useBean> tag is page. JSP implicit objects out, exception, response, pageContext, config and page have 'page' scope.
- **request**
  A JSP object created using the 'request' scope can be accessed from any pages that serves that request. More than one page can serve a single request. The JSP object will be bound to the request object. Implicit object request has the 'request' scope.
- **session**
  'session' scope means, the JSP object is accessible from pages that belong to the same session from where it was created. The JSP object that is created using the session scope is bound to the session object. Implicit object session has the 'session' scope.
- **application**
  A JSP object created using the 'application' scope can be accessed from any pages across the application. The JSP object is bound to the application object. Implicit object application has the 'application' scope.