Servelt Notes

1. What is Maven?

Maven is a **build automation and dependency management tool** used primarily for Java projects. It simplifies the build process, handles project dependencies, and manages project lifecycles using a central XML file.

□ Key Features:

- **Project Object Model (POM):** Central configuration file (pom.xml)
- **Dependency Management:** Automatically downloads required JAR files from Maven Central Repository
- **Build Lifecycle:** Phases like validate, compile, test, package, install, deploy
- **Standard Directory Structure**

□ Sample pom.xml:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.prathamesh</groupId>
  <artifactId>servlet-demo</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>4.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

---

2. Introduction to Servlet

What is a Servlet?

A **Servlet** is a **Java class** used to build **dynamic web applications**. It runs on a **Servlet container (like Tomcat)** and handles **HTTP requests and responses**.

Think of a servlet as a Java program that responds to web browser requests.

□ Key Characteristics:

- Part of **Java EE (Jakarta EE)** platform
- Managed by **Servlet container**
- Works with **HttpServletRequest** and **HttpServletResponse**
- Uses **web.xml** or @WebServlet annotations for deployment

□ Advantages of Servlets over CGI

Feature	CGI	Servlet
Performance	Creates new process per request	Uses a single instance (threaded)
Portability	Limited	Fully portable (Java-based)
Efficiency	High memory usage	Low memory usage
Platform	OS-dependent scripts	Platform-independent (Java)
Security	Less secure	Supports modern Java security

Servlet Lifecycle

The lifecycle of a servlet is managed by the servlet container (e.g., Apache Tomcat). It involves **3 main methods**:

1. init()

Called **once** when the servlet is first loaded. Used to initialize resources (like DB connections).

```
public void init() throws ServletException {
```

```
    // Initialization code
}
```

2.service()

Called **every time** the servlet is requested. Handles **HTTP requests and responses**.

```
public void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    // Main logic
}
```

Internally, service() delegates to doGet(), doPost() based on the request type.

3.destroy()

Called **once** when the servlet is being removed. Used to **release resources**.

```
public void destroy() {
    // Cleanup code
}
```

Creating Your First Servlet

Writing a Servlet Class using HttpServlet

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

public class HelloServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<h1>Hello, Servlet!</h1>");
    }
}
```

□ Overriding doGet() and doPost() Methods

- **doGet()** handles GET requests (like typing a URL or clicking a link).
- **doPost()** handles form submissions or API POST calls.

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   PrintWriter out = res.getWriter();
   out.println("This is a POST request.");
}
```

Deploying Servlet on Tomcat

1. **Compile the Servlet class** → Place .class in WEB-INF/classes/
2. **Create a WAR file** (optional):
    - Structure:

```
MyApp/
├── WEB-INF/
│   ├── web.xml
│   └── classes/
│       └── HelloServlet.class
```

3. Copy folder to:
   C:\apache-tomcat\webapps\MyApp
4. Start Tomcat and visit:
   http://localhost:8080/MyApp/hello

Mapping Servlet using web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
       version="3.1">

 <servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
 </servlet>
```

```
  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>

</web-app>
```

Or use **annotation-based mapping**:

```
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    // ...
}
```

Servlet API and Core Classes

The **Servlet API** is provided in the javax.servlet and javax.servlet.http packages (or jakarta.servlet in newer versions). These packages define **interfaces and classes** that a servlet uses to handle requests and responses in a web application.

□ Core Interfaces and Classes

Component	Description
Servlet	The base interface for all servlets
GenericServlet	Abstract class implementing Servlet, protocol-independent
HttpServlet	Extends GenericServlet, provides HTTP-specific methods
ServletRequest	Provides client request information
HttpServletRequest	Extends ServletRequest, adds HTTP request methods
ServletResponse	Used to send responses to the client

Component	Description
HttpServletResponse	Extends ServletResponse, adds HTTP-specific features

---

HttpServletRequest and HttpServletResponse

HttpServletRequest

Used to **retrieve information** about the HTTP request sent by the client.

Common Methods:

```
String getParameter(String name);        // Get form field value
String getMethod();                      // GET, POST, etc.
String getRequestURI();                  // URI of the request
String getHeader(String name);           // Fetch request header
```

HttpServletResponse

Used to **send a response** back to the client (HTML, file, JSON, etc.).

Common Methods:

```
void setContentType(String type);        // Set response content type
PrintWriter getWriter();                 // Write response body
void sendRedirect(String location);      // Redirect to another page
void setHeader(String name, String value);// Set custom headers
void setStatus(int sc);                  // Set HTTP status code
```

Retrieving Request Parameters

If an HTML form sends data like:

```
<form action="login" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
</form>
```

The servlet reads the parameters using:

```
protected void doPost(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   String user = req.getParameter("username");
   String pass = req.getParameter("password");

   PrintWriter out = res.getWriter();
   out.println("Username: " + user);
   out.println("Password: " + pass);
}
```

Sending Response to the Client

Example: Sending HTML response

```
res.setContentType("text/html");
PrintWriter out = res.getWriter();
out.println("<h1>Welcome to Servlet World</h1>");
```

Example: Sending Redirect

```
res.sendRedirect("https://www.google.com");
```

Setting HTTP Status Codes and Headers

- **Setting Status Code**

```
res.setStatus(HttpServletResponse.SC_OK);            // 200 OK
res.setStatus(HttpServletResponse.SC_NOT_FOUND);     // 404 Not Found
res.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR); // 500
```

- **Setting Headers**

```
res.setHeader("Content-Type", "text/html");
res.setHeader("Cache-Control", "no-cache");
```

Servlet Configuration

Servlets can be configured using two main approaches:

1. web.xml Configuration (Deployment Descriptor)

Located in:
WebContent/WEB-INF/web.xml or src/main/webapp/WEB-INF/web.xml

 Example:

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="3.1">
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>com.example.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

 Useful when:

- You want centralized configuration
- Using older versions of Java EE

---

2. Annotation-based Configuration (@WebServlet)

Modern alternative to web.xml.

 Example:

```
import javax.servlet.annotation.WebServlet;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    // doGet() or doPost()
}
```

 Advantages:

- Simple and concise
- No need to touch web.xml
- Automatically detected by servlet container

---

3. Understanding WAR File Structure

A **WAR (Web Application Archive)** is a deployable unit containing all files related to a web application.

```
MyApp.war
├── index.html
├── login.jsp
├── WEB-INF/
│   ├── web.xml
│   └── classes/
│       └── com/
│           └── example/
│               └── HelloServlet.class
│
└── lib/ (optional JAR dependencies)
```

 WAR files can be deployed directly to tomcat/webapps/ directory.

---

5. Handling Form Data

Using doGet() vs doPost() for Form Submission

Aspect	doGet()	doPost()
Data sent via	URL parameters (?name=value)	HTTP body
Size limit	Limited (URL length)	No size limit
Security	Less secure (data visible in URL)	More secure (data hidden in body)

Aspect	doGet()	doPost()
Use case	Search, bookmarking	Login forms, sensitive data

HTML Example:

```
<form action="login" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
  <input type="submit">
</form>
```

Reading Form Input Values

```
protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
    String user = req.getParameter("username");
    String pass = req.getParameter("password");

    PrintWriter out = res.getWriter();
    out.println("Username: " + user);
    out.println("Password: " + pass);
}
```

Redirect vs Forward

sendRedirect() (Client-side Redirect)

- Redirects to a **new URL**
- **Client is involved**
- URL changes in browser
- Can redirect to **external** websites

```
res.sendRedirect("dashboard.jsp");
```

 Use when:
You want to go to another servlet or website **with a new request**

RequestDispatcher.forward() (Server-side Forward)

- Forwards the request internally
- **Client is unaware**
- URL remains the same
- Shares **request object**

```
RequestDispatcher rd = req.getRequestDispatcher("welcome.jsp");
rd.forward(req, res);
```

 Use when:
You want to **pass control internally** (like from login servlet to home.jsp)

Session Management in Servlets

Session Management is used to maintain **state** (user data) across multiple HTTP requests in web applications.

HTTP is stateless — session management helps identify requests from the same user.

---

1. Cookies

A **cookie** is a small piece of data stored on the client-side (browser) and sent with each request to the server.

 How to Create and Send Cookies:

```
Cookie cookie = new Cookie("username", "Prathamesh");
cookie.setMaxAge(3600); // 1 hour
res.addCookie(cookie);
```

 How to Read Cookies:

```
Cookie[] cookies = req.getCookies();
for (Cookie c : cookies) {
   if (c.getName().equals("username")) {
```

```
        out.println("Welcome " + c.getValue());
    }
}
```

 Use Case: "Remember Me" functionality
Limitation: Can be disabled by users; limited data (usually 4KB)

---

2. HttpSession Usage

HttpSession is a server-side session tracking mechanism. It stores data for a user across multiple requests.

 Create/Retrieve Session:

```
HttpSession session = req.getSession();
session.setAttribute("username", "Prathamesh");
```

 Get Stored Data:

```
String user = (String) session.getAttribute("username");
out.println("Welcome " + user);
```

 Invalidate Session (Logout):

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

 Use Case: Login systems, shopping cart
Each session has a unique **Session ID**

---

3. URL Rewriting

URL Rewriting adds session data as part of the URL query string.

 Syntax:

```
<a href="nextServlet?username=Prathamesh">Click Here</a>
```

 Retrieve in next servlet:

```
String user = req.getParameter("username");
```

 Use Case: When cookies are disabled
Limitation: URL becomes long and less secure

---

4. Hidden Form Fields

Hidden fields store data in HTML forms, invisible to the user.

 HTML Form:

```
<form action="nextServlet" method="post">
  <input type="hidden" name="username" value="Prathamesh">
  <input type="submit" value="Submit">
</form>
```

 Servlet:

```
String user = req.getParameter("username");
```

 Use Case: Passing temporary data
Limitation: Works only with form submission

ServletContext and ServletConfig

Both are configuration objects provided by the Servlet container, but they serve **different scopes** and purposes.

ServletConfig

- Specific to **one servlet**
- Used to pass **initialization parameters** to a servlet
- Defined in web.xml

 In web.xml:

```
<servlet>
  <servlet-name>DemoServlet</servlet-name>
  <servlet-class>DemoServlet</servlet-class>
  <init-param>
    <param-name>company</param-name>
    <param-value>JadhavTech</param-value>
```

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

 In Servlet:

```
public void init(ServletConfig config) throws ServletException {
   String company = config.getInitParameter("company");
}
```

ServletContext

- **Application-wide** scope
- Shared by **all servlets** in the web app
- Used to **share data, get metadata**, or access resources

 Usage:

```
ServletContext context = getServletContext();
context.setAttribute("institute", "ADTech");

String inst = (String) context.getAttribute("institute");
```

Difference: ServletConfig vs ServletContext

Feature	ServletConfig	ServletContext
Scope	Per servlet	Application-wide (all servlets)
Defined in	Inside servlet tag in web.xml	Globally accessible
Accessed via	getServletConfig()	getServletContext()
Used for	Init parameters	Sharing global data, resources

Sharing Data Between Servlets

- **Using ServletContext:**

```
ServletContext context = getServletContext();
context.setAttribute("user", "Prathamesh");

// In another servlet
String user = (String) context.getAttribute("user");
```

- **Using HttpSession:**

```
HttpSession session = req.getSession();
session.setAttribute("cart", cartObject);

// Accessible in other servlet
Cart cart = (Cart) session.getAttribute("cart");
```

---

8. MVC Pattern (Servlet + JSP)

What is MVC?

MVC stands for **Model-View-Controller**. It is a design pattern that separates an application into 3 layers:

Component	Responsibility
Model	Business logic and data (Java classes, DB)
View	UI layer (JSP, HTML)
Controller	Handles user input (Servlet)

Basics of JSP

- **JSP (Java Server Pages)** are used to create **dynamic web pages**
- Combines HTML with Java code

- Usually used as the **View layer** in MVC

```
<%@ page language="java" %>
<html>
<body>
  <h1>Welcome, <%= request.getAttribute("username") %></h1>
</body>
</html>
```

Using Servlet as Controller, JSP as View

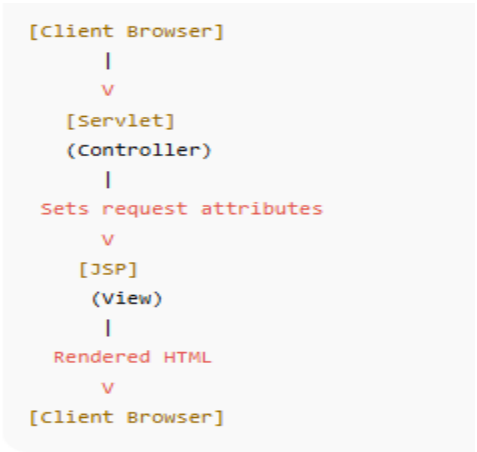
Servlet (Controller):

```
protected void doPost(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException {
    String user = req.getParameter("username");
    req.setAttribute("username", user);
    RequestDispatcher rd = req.getRequestDispatcher("welcome.jsp");
    rd.forward(req, res);
}
```

JSP (View):

```
<h2>Hello, <%= request.getAttribute("username") %>!</h2>
```

Flow of Data in MVC Architecture



Advantages of MVC:

- Separation of concerns
- Easier to maintain and test
- Code reusability

Filters and Listeners (Advanced but Useful)

What are Servlet Filters?

Filters are Java classes that intercept and process **incoming requests** and/or **outgoing responses** *before* they reach a servlet or JSP.

 Use Cases:

- Logging
- Authentication
- Compression
- Content transformation (like adding headers)

Creating a Servlet Filter

Step 1: Create Filter Class

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

public class LoggingFilter implements Filter {
   public void init(FilterConfig config) {}

   public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
         throws IOException, ServletException {
      System.out.println("Request received at: " + new java.util.Date());

      // Forward to next filter or servlet
      chain.doFilter(req, res);
   }
```

```
    public void destroy() {}
}
```

Step 2: Register Filter

In web.xml:

```
<filter>
  <filter-name>LogFilter</filter-name>
  <filter-class>LoggingFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>LogFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Or using annotation:

```
@WebFilter("/*")
public class LoggingFilter implements Filter { ... }
```

Introduction to Servlet Listeners

Listeners allow you to listen to lifecycle events of:

- **ServletContext**
- **HttpSession**
- **ServletRequest**

 Use Cases:

- Count total visitors
- Initialize DB connection
- Monitor session creation/destruction

Creating a Listener

```
@WebListener
public class AppContextListener implements ServletContextListener {
   public void contextInitialized(ServletContextEvent sce) {
      System.out.println("Application Started");
   }

   public void contextDestroyed(ServletContextEvent sce) {
      System.out.println("Application Stopped");
   }
}
```

Other Listener Types

Listener Interface	Purpose
ServletContextListener	App start/stop
HttpSessionListener	Session creation/destruction
ServletRequestListener	Request init/destroy

Build and Deploy WAR File

Creating a WAR File (Manually)

A WAR (Web Application Archive) file is used to **deploy Java web applications**.

Folder Structure:

```
MyApp/
├── index.html
├── WEB-INF/
│   ├── web.xml
│   └── classes/
│       └── MyServlet.class
```

Creating WAR via Command Line:

jar -cvf MyApp.war *

 Manual Deployment to Tomcat

1. Place MyApp.war in:

   apache-tomcat/webapps/

2. Start Tomcat:

   /bin/startup.bat

3. Access:

   http://localhost:8080/MyApp/

Deployment from IDE (Eclipse)

Step-by-step in Eclipse:

1. Right-click project → **Run As → Run on Server**
2. Or export:

   File → Export → WAR File

3. Deploy to:

   Tomcat/webapps/

 Eclipse auto-detects Tomcat and deploys automatically when configured.