A Java Servlet is a server-side Java program that handles HTTP requests and generates dynamic web responses. It runs inside a Servlet Container (e.g., Apache Tomcat) and follows the Java EE (Jakarta EE) specification.

#### When to Use Jakarta EE?

- If you are developing Java web applications using Servlets, JSP, JSF, or EJB.
- If you want to follow the latest Java enterprise standards (Jakarta EE is the successor to Java EE).

#### When to Use Maven?

- If you are managing project dependencies automatically instead of manually downloading JAR files.
- If you need to compile, test, and deploy your Java web app in a structured way.
- If you are working with Jakarta EE or Spring Boot, as they use Maven for dependency management.

**Best Practice: Use Both Together!** 

For a Java Servlet-based web application, the ideal setup is:

- 1. Use Jakarta EE for writing Servlets, JSP, and handling web requests.
- Use Maven to manage dependencies (like jakarta.servlet-api) and build the project.

# 1. HTTP Methods

#### **Definition**

HTTP (HyperText Transfer Protocol) methods define the type of action to be performed on a given resource. Servlets process these requests and generate responses accordingly.

#### **Common HTTP Methods**

- 1. **GET** Retrieves data from the server (e.g., fetching a web page).
- 2. **POST** Sends data to the server (e.g., submitting a form).
- 3. PUT Updates a resource on the server.
- 4. **DELETE** Removes a resource from the server.
- 5. **HEAD** Similar to GET but retrieves only the headers.
- 6. **OPTIONS** Retrieves supported HTTP methods for a resource.
- 7. **TRACE** Performs a loopback test to check request handling.

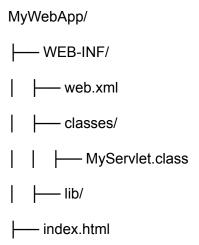
# **Handling HTTP Methods in Servlets**

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.getWriter().println("GET method called");
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
    response.getWriter().println("POST method called");
}
```

# 2. Structure and Deployment Descriptor

## **Servlet Directory Structure**



## **Deployment Descriptor (web.xml)**

A deployment descriptor is an XML file (web.xml) used to configure servlets, mappings, and initialization parameters.

### Sample web.xml Configuration

# 3. ServletContext and ServletConfig Interfaces

## ServletConfig Interface

- Provides configuration information to a specific servlet.
- Defined in javax.servlet.ServletConfig.
- Used to read initialization parameters from web.xml.

#### Example:

```
public void init(ServletConfig config) throws ServletException {
   String paramValue = config.getInitParameter("paramName");
   System.out.println("Init Parameter Value: " + paramValue);
}
```

#### ServletContext Interface

- Provides configuration information at the application level.
- Defined in javax.servlet.ServletContext.
- Allows sharing of data among servlets in the same web application.

#### Example:

```
ServletContext context = getServletContext();

String appValue = context.getInitParameter("globalParam");

System.out.println("Global Parameter: " + appValue);
```

# 4. Attributes in Servlet

Attributes allow servlets to share data within the request, session, or application scope.

### **Scopes of Attributes**

```
Request Scope: Data is available during a single request.

request.setAttribute("name", "John");

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

Session Scope: Data persists across multiple requests from the same client.

HttpSession session = request.getSession();

session.setAttribute("user", "Alice");

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

Application Scope: Data is shared across all servlets in the application.

ServletContext context = getServletContext();
```

#### Conclusion

context.setAttribute("count", 100);

int count = (int) context.getAttribute("count");

- HTTP methods determine how a servlet handles client requests.
- The deployment descriptor (web.xml) configures servlet mappings.
- ServletConfig and ServletContext provide initialization and application-wide parameters.
- Attributes facilitate data sharing within different scopes in a servlet application.

## 5. Introduction to Filter API

Filters in Java Servlets provide a way to modify requests and responses before they reach servlets or after servlet processing is complete. Filters are used for functionalities like authentication, logging, compression, encryption, and more.

#### 1. Filter Interface

The <code>javax.servlet.Filter</code> interface is the main interface for implementing filters in Java Servlets. A filter intercepts requests and responses and can modify them before passing them along the filter chain.

#### **Methods in Filter Interface**

- init(FilterConfig config): Initializes the filter.
- doFilter(ServletRequest request, ServletResponse response, FilterChain chain): Performs filtering operations and passes the request/response to the next entity in the chain.
- destroy(): Cleans up any resources before the filter is destroyed.

## **Example of a Simple Filter**

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;

@WebFilter("/*")

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

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

```
System.out.println("Request received at " + new java.util.Date());
chain.doFilter(request, response); // Pass the request along the chain
}

public void destroy() {}
}
```

#### 2. FilterChain Interface

The javax.servlet.FilterChain interface allows filters to pass requests and responses along the chain of filters and servlets.

#### Methods in FilterChain Interface

• void doFilter(ServletRequest request, ServletResponse response): Passes control to the next filter or servlet in the chain.

#### **Example Usage in a Filter**

## 3. FilterConfig Interface

The javax.servlet.FilterConfig interface is used to retrieve configuration parameters and servlet context information.

## **Methods in FilterConfig Interface**

- String getFilterName(): Returns the filter's name.
- ServletContext getServletContext(): Returns the ServletContext associated with the filter.
- String getInitParameter(String name): Retrieves initialization parameters defined in web.xml.
- Enumeration<String> getInitParameterNames(): Retrieves all initialization parameter names.

## **Example Usage in a Filter**

```
public void init(FilterConfig config) {
    String paramValue = config.getInitParameter("exampleParam");
    System.out.println("Filter initialized with param: " + paramValue);
}
```

# Conclusion

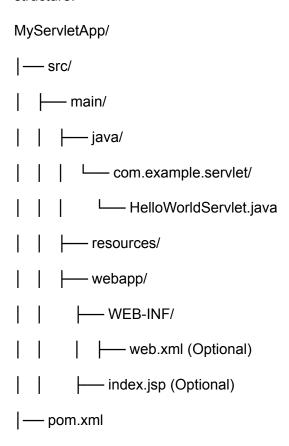
Filters in Java Servlets allow pre-processing and post-processing of requests and responses. The **Filter API** includes the Filter interface to define filter behavior, the FilterChain interface to manage filter execution, and the FilterConfig interface to handle filter configuration. Filters enhance security, logging, and request modification in Java web applications.

When you create a **Java Servlet project** with **Maven Web Application** as the architecture type, the execution flow follows a structured process. Below is the step-by-step execution flow for a **"Hello World" Servlet** project.

**Execution Flow of a Maven Web App with a Servlet** 

### 1. Project Creation and Directory Structure

When you create a Maven Web Application, your project will have the following standard structure:



# 2. Dependency Management (Maven - pom.xml)

Maven manages the dependencies. A typical pom.xml includes: <dependencies>

<dependency>

• **Scope: Provided** means the Servlet API is provided by the application server (e.g., Apache Tomcat).

# 3. Servlet Implementation (HelloWorldServlet.java)

```
response.getWriter().println("<h1>Hello, World!</h1>");
}
```

- The @WebServlet("/hello") annotation registers the servlet.
- The doGet() method handles HTTP GET requests and prints "Hello, World!" in response.

# 4. Deployment Descriptor (web.xml) - Optional

# 5. Compilation and Build (mvn clean package)

mvn clean package

- •
- This compiles the Java code, packages it into a .war file, and prepares it for deployment.

# 6. Deploying the Web Application

- 1. Copy the generated .war file from target/ to the webapps/ folder of Tomcat.
- 2. Start Apache Tomcat:

startup.bat (Windows)

./startup.sh (Linux/Mac)

3. Access the servlet in the browser:

http://localhost:8080/MyServletApp/hello

This should display:

Hello, World!

# **Execution Flow Summary**

- 1. **Client Request:** A browser or client sends an HTTP request to http://localhost:8080/MyServletApp/hello.
- 2. **Tomcat Processes Request:** The application server (Tomcat) identifies the request URL and maps it to the HelloWorldServlet.

- 3. **Servlet Execution:** The doGet() method of HelloWorldServlet is executed.
- 4. **Response Generation:** The servlet generates an HTML response (<h1>Hello, World!</h1>).
- 5. Response Sent to Client: The browser displays "Hello, World!".

## Conclusion

- The Maven Web App structure makes dependency management easier.
- The servlet is mapped via annotations (@WebServlet) or web.xml.
- Tomcat executes the servlet and returns the response to the client.