Servlets In Java
Real Life & Hands-On Approach

Scenario: You open Swiggy, select your favourite biryani, and hit the **Order** button. What happens behind the scenes?

Step-by-Step Flow 🪜

1. **Client (Your Browser/App)** → Sends the order request (with details like food items, address, payment info) over the internet.
2. **Server** → Receives the request.
3. **Servlet (The Middleman)** → Processes that request:
    - Reads your order data.
    - Checks the database for restaurant availability.
    - Calculates delivery time.
    - Sends the confirmation back to you.
4. **Database** → Stores the order for tracking and kitchen processing.
5. **Response** → Your browser/app shows "Order Confirmed" with an ETA.

Key Takeaway 🎯

- A Servlet is not just "some Java file" — it's the **traffic controller** for web requests.
- It receives a **request**, processes it, and sends a **response** — just like Swiggy's backend does for millions of orders daily.
- Understanding Servlets means understanding the foundation of how most dynamic web apps work — even modern frameworks like Spring Boot still use them under the hood.

---

2. What is a Servlet?

A **Servlet** is a Java program that runs inside a **server** and handles **HTTP requests** and **responses**.

- Think of it as the *waiter* in a restaurant:
  - Takes your **order** (request).
  - Passes it to the **kitchen** (business logic/database).
  - Brings back your **food** (response).

Servlet Life Cycle Methods 🛟

1. **Loading and Instantiation:** The servlet class is loaded, and an instance is created.
2. **init()** – Runs once when the servlet is loaded. The init() method sets up resources.

   Like a chef preparing the kitchen before the day starts.

```
public void init() {
   // Initialization code (DB connection, config)
}
```

3. **service()** – Runs every time a request comes in. The service() method processes requests, delegating to doGet() or doPost().

   Like preparing each dish for a customer.

```
public void service(HttpServletRequest req, HttpServletResponse
res) {
   // Handle request & generate response
}
```

4. **destroy()** – Runs once when the servlet is being unloaded.
   The destroy() method releases resources.

   Like cleaning up the kitchen at closing time.

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

Role of the Servlet Container (e.g., Tomcat)

- Loads servlet classes.
- Manages lifecycle (init, service, destroy).
- Handles HTTP request/response behind the scenes.
- Provides built-in features like session management and security.

Ways to Implement Servlets 💻

Servlets can be implemented in three ways:

- **Implementing Servlet Interface**: This approach provides full control over all lifecycle methods (init(), service(), destroy(), etc.). It is the most basic method but requires more boilerplate code.
- **Extending GenericServlet**: This is a protocol-independent abstract class that simplifies servlet creation by handling common functionality. Developers only need to override the service() method.
- **Extending HttpServlet**: This is the most commonly used method, designed specifically for handling HTTP requests. It provides convenient methods like doGet(), doPost(), etc., making it ideal for web-based applications.

Setting Up the Environment

Required Tools

- **JDK**: Java Development Kit
- **Apache Tomcat**: Web server and servlet container
- **Eclipse/IntelliJ IDEA**: Integrated Development Environment (IDE)
- **Configure tomcat with IDE**

Creating a Simple Project

1. **Eclipse Setup**:
    - Open Eclipse and create a new Dynamic Web Project.
    - Set up Apache Tomcat in Eclipse.
    - Create a new servlet class.

Creating Servlets Using:

1. Implementing Servlet Interface:

```
package Servlets; //Create this package inside src/main/java

import java.io.IOException;
import jakarta.servlet.Servlet;
import jakarta.servlet.ServletConfig;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;
import jakarta.servlet.annotation.WebServlet;

@WebServlet("/first")
public class FirstOne implements Servlet{

  private ServletConfig servletconfig;
  @Override
  public void init(ServletConfig arg0) throws ServletException {
           this.servletconfig = arg0;
           System.out.println("Second Step In SLC");
  }

  @Override
  public void service(ServletRequest arg0, ServletResponse arg1)
throws ServletException, IOException {
           System.out.println("Third Step In SLC");
           String userInput = arg0.getParameter("userInput");
           arg1.setContentType("Text/Html");
```

```
            String webcontent = "<h1>Hello</h1>" + userInput;
            arg1.getWriter().println(webcontent);
  }

  @Override
  public void destroy() {
            System.out.println("Fourth Step In SLC");
  }

  @Override
  public ServletConfig getServletConfig() {
            return this.servletconfig;
  }

  @Override
  public String getServletInfo() {
            String name = "I have create this servlet";
            return name;
  }
}
```

2. Extending GenericServlet

```
package Servlets;

import java.io.IOException;
import jakarta.servlet.GenericServlet;
import jakarta.servlet.ServletException;
import jakarta.servlet.ServletRequest;
import jakarta.servlet.ServletResponse;

public class SecondOne extends GenericServlet {

  @Override
  public void service(ServletRequest arg0, ServletResponse arg1)
throws ServletException, IOException {

            System.out.println("This is the second way to create
servlet!");
```

```
            arg1.setContentType("Text/Html");
            arg1.getWriter().println("""
                        <!DOCTYPE html>
                        <html>
                        <head>
                        <meta charset="UTF-8">
                        <title>Generic Servlet</title>
                        <style>
                        h1{
                              color: red;
                        }
                        </style>
                        </head>
                        <body>
                              <h1>Hello From Generic
Servlet!</h1>
                        </body>
                        </html>
                        """);
  }
}
```

Why web.xml Configuration is Needed for GenericServlet 🥨

- **GenericServlet** is an **abstract class** in the jakarta.servlet (or javax.servlet) package that implements the Servlet interface.
- Unlike HttpServlet (which can use the @WebServlet annotation in newer versions), GenericServlet is **protocol-independent** and does not inherently support the annotation mapping mechanism.
- **This means:**
    - The servlet container **doesn't automatically know** what URL should trigger your servlet.
    - You must **manually map it** in web.xml.

Example web.xml Configuration for GenericServlet

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <display-name>ServletWithJsp</display-name>
  <welcome-file-list>
            <welcome-file>index.html</welcome-file>
            <welcome-file>index.htm</welcome-file>
            <welcome-file>index.jsp</welcome-file>
            <welcome-file>index.xhtml</welcome-file>
            <welcome-file>default.html</welcome-file>
            <welcome-file>default.htm</welcome-file>
            <welcome-file>default.jsp</welcome-file>
            <welcome-file>default.xhtml</welcome-file>
  </welcome-file-list>

  <!-- Define the servlet -->
  <servlet>
            <servlet-name>SecondOne</servlet-name>
            <servlet-class>Servlets.SecondOne</servlet-class>
  </servlet>

  <!-- Map URL to servlet -->
  <servlet-mapping>
            <servlet-name>SecondOne</servlet-name>
            <url-pattern>/second</url-pattern>
  </servlet-mapping>
</web-app>
```

Analogy 📌

Think of GenericServlet as a *freelancer* who needs to register in the “job directory” (web.xml) so clients (browsers) know how to reach them.
In contrast, @WebServlet is like posting your services directly on social media — quick, but not always the format older systems use.

3. **Extending HttpServlet**:

```
package Servlets;

import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

@WebServlet("/form")
public class Form extends HttpServlet{

	@Override
	protected void doPost(HttpServletRequest req,
HttpServletResponse resp) throws ServletException, IOException {
		System.out.println("This is 3rd way to create Servlet!");

		resp.setContentType("Text/Html");
		resp.getWriter().println("<h1>Hello From Http
Servlet</h1>");
		String name = req.getParameter("name");
		String email = req.getParameter("email");
		resp.getWriter().println("Your Name: " + name + "<br />");
		resp.getWriter().println("Your Email: " + email);
	}
}
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>SignUp Page</title>
<style>
	body{
		width: 700px;
		margin: 0 auto;
	}
	div{
			text-align: left;
	}
</style>
</head>
<body>
	<h1>SignUp Form!</h1>
	<form method="post" action="<%= request.getContextPath()
%>/form">
			<div>
			<label>Name:</label>
			<p><input name="name" type="text" placeholder="Enter
Your Name!" /></p>
			</div>
			<div>
			<label>Email:</label>
			<p><input name="email" type="email" placeholder="Enter
Your Email!" /></p>
			</div>
			<div>
			<label>Password:</label>
			<p><input name="password" type="password"
placeholder="Enter Your Password!" /></p>
			</div>
			<button>Submit</button>
	</form>
</body>
</html>
```

To deepen your understanding of Servlets, JSP, and JSF, check out this insightful blog post: Servelts, JSP, and JSF