Java Servlet API

Server-side technologies:

Server-side technology refers to the software and frameworks used to develop and run applications on the server, handling the processing of data and delivering dynamic content to the client (such as a web browser). These technologies manage tasks such as database interaction, business logic, and user authentication. When a client (like a browser) makes a request (e.g., submitting a form), the server-side technology processes the request, communicates with the database if needed, and returns a response (usually an HTML page or data like JSON) to the client.

Different Server-Side Technologies:

- **Java (Java EE or Jakarta EE):** Java is a popular object-oriented programming language, and Java Enterprise Edition (Java EE) (now known as Jakarta EE) is its platform for developing large-scale, enterprise-level applications. It includes technologies like **Servlets**, **JSP** (JavaServer Pages), and **Enterprise JavaBeans** (EJB) to build dynamic web applications and APIs.
    - **Popular Frameworks:**
        - **Spring**: A powerful Java framework used to build Java-based web applications, including Spring MVC for web development.
        - **Struts**: A framework for developing Java-based web applications.
- **ASP.NET (C#):** ASP.NET is a server-side framework developed by Microsoft for building dynamic web applications and services using C#. It provides robust tools and libraries for creating large, scalable web applications.
    - **Popular Frameworks:**
        - **ASP.NET Core:** A cross-platform version of ASP.NET, designed for building modern, cloud-based, and internet-connected applications.
- **Node.js (JavaScript):** Node.js is a server-side runtime environment that allows you to run JavaScript on the server. It is event-driven, non-blocking, and highly efficient for building scalable, real-time applications.
    - **Popular Frameworks:**
        - **Express.js:** A minimal and flexible Node.js web framework that provides a set of robust features for web and mobile applications.
        - **NestJS:** A progressive Node.js framework for building efficient and scalable server-side applications.
- **PHP (Hypertext Preprocessor):** PHP is a server-side scripting language designed for web development. It is especially suited for creating dynamic and interactive websites and is widely used because of its simplicity and integration with databases like MySQL.
    - **Popular Frameworks:**
        - **Laravel:** A modern PHP framework that simplifies development with elegant syntax and a range of built-in features like routing, authentication, and database management.

        - **Symfony:** A PHP framework for building scalable, high-performance web applications.
- **Python:** Python is a versatile, high-level programming language widely used for web development, data science, and machine learning. It is known for its readability and ease of use.
    - **Popular Frameworks:**
        - **Django**: A high-level Python web framework that encourages rapid development and clean, pragmatic design. It includes tools for authentication, database management, and URL routing.
        - **Flask**: A lightweight micro-framework for Python that is flexible and easy to extend.
- **Ruby on Rails:** Ruby on Rails (or Rails) is a server-side web application framework written in Ruby. It follows the Model-View-Controller (MVC) pattern and emphasizes convention over configuration, making development faster and more efficient.
    - **Popular Frameworks:** Rails itself is the primary framework for Ruby on Rails applications.
- **Go (Golang):** Go, or Golang, is a statically typed, compiled programming language created by Google. It is known for its performance, efficiency, and simplicity, making it well-suited for building scalable web applications and APIs.
    - **Popular Frameworks:**
        - **Gin**: A high-performance Go web framework.
        - **Echo**: A minimal and fast Go web framework for building scalable APIs.

What is Java EE or Jakarta EE?

Java EE (Java Enterprise Edition) and Jakarta EE are frameworks designed to simplify the development of enterprise-level applications in Java. They provide a standardized set of APIs and services that enable developers to build scalable, robust, and secure applications. Here's a detailed explanation of Java EE and Jakarta EE:

Java EE (Java Enterprise Edition): Java EE is a set of specifications and APIs that extend the core Java SE (Standard Edition) platform to support enterprise-level features. It is designed to simplify the development of large-scale, distributed, and transactional applications. Java EE provides a standardized approach to building enterprise applications, making it easier to develop, deploy, and manage complex systems.

The key Components of Java EE are

- **Servlet API:** For handling HTTP requests and generating dynamic web content.
- **JavaServer Pages (JSP):** For creating dynamic web pages using a combination of HTML and Java code.
- **Enterprise JavaBeans (EJB):** For building distributed, transactional, and secure business components.
- **Java Persistence API (JPA):** For object-relational mapping and database access.
- **Java Message Service (JMS):** For messaging and asynchronous communication between applications.

- **JavaServer Faces (JSF):** For building user interfaces for web applications.
- **Contexts and Dependency Injection (CDI):** For managing the lifecycle and dependencies of components.
- **Web Services:** For building and consuming web services using SOAP and REST.
- **Java Transaction API (JTA):** For managing transactions across multiple resources.
- **JavaMail API:** For sending and receiving emails.

Jakarta EE: Jakarta EE is the successor to Java EE. In 2017, Oracle transferred the stewardship of Java EE to the Eclipse Foundation, and it was rebranded as Jakarta EE. The goal was to foster innovation, increase community involvement, and accelerate the evolution of the platform. Jakarta EE builds on the foundation of Java EE and continues to provide a standardized set of APIs for enterprise application development.

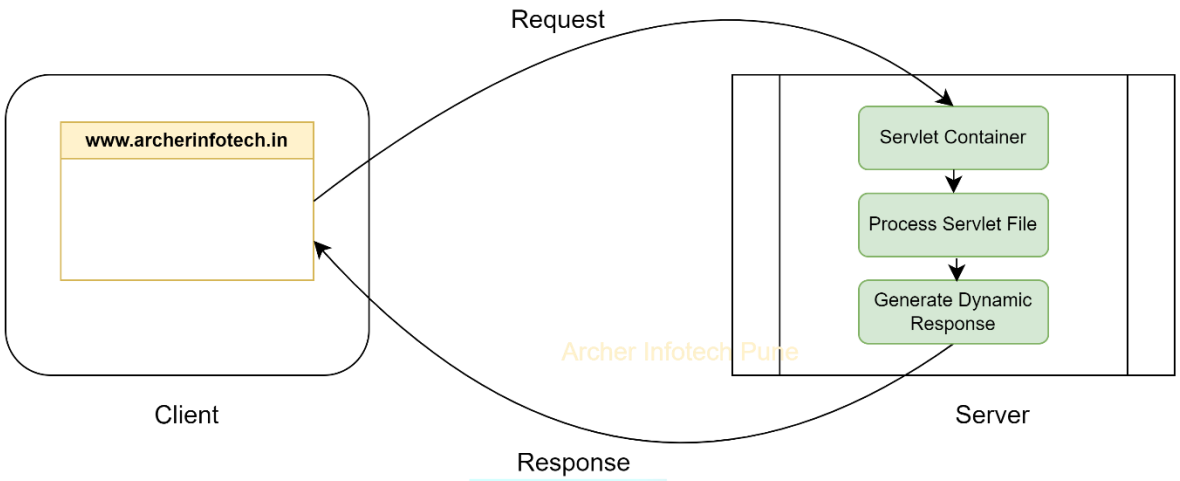
Jakarta EE includes all the components of Java EE, with some enhancements and updates. The key components are:

- **Jakarta Servlet:** For handling HTTP requests and generating dynamic web content.
- **Jakarta Server Pages (JSP):** For creating dynamic web pages.
- **Jakarta Enterprise Beans (EJB):** For building distributed, transactional, and secure business components.
- **Jakarta Persistence:** For object-relational mapping and database access.
- **Jakarta Messaging:** For messaging and asynchronous communication between applications.
- **Jakarta Faces:** For building user interfaces for web applications.
- **Jakarta Contexts and Dependency Injection (CDI):** For managing the lifecycle and dependencies of components.
- **Jakarta RESTful Web Services:** For building and consuming RESTful web services.
- **Jakarta Transaction:** For managing transactions across multiple resources.
- **Jakarta Mail:** For sending and receiving emails.

Comparison between Java EE and Jakarta EE:

- **Name and Governance:** Java EE was governed by Oracle, while Jakarta EE is governed by the Eclipse Foundation.
- **Community Involvement:** Jakarta EE has a more open and community-driven development process.
- **Namespace:** Java EE uses the javax namespace, while Jakarta EE uses the jakarta namespace.
- **Innovation:** Jakarta EE aims to accelerate innovation and modernize the platform to keep up with industry trends.

Servlet: A servlet in Java is a server-side program that extends the capabilities of a web server by generating dynamic content. Servlets are part of the Java Enterprise Edition (Java EE) and are used to ***handle requests and generate responses in a web application***. They are typically used to create dynamic web content, process form data, manage user sessions, and interact with databases.



Servlet Container: A Servlet Container is a part of a web server that interacts with the servlets. It manages the servlet lifecycle and acts as an intermediary between the servlet and the client. Some popular servlet containers include: Apache Tomcat, Jetty, GlassFish.

Servlet Lifecycle:

The servlet lifecycle is a well-defined sequence of stages that a servlet goes through from its creation to its destruction. Understanding the servlet lifecycle is crucial for developing efficient and robust web applications. Here's a detailed explanation of each stage in the servlet lifecycle:

The Servlet Life Cycle consists of five main phases:

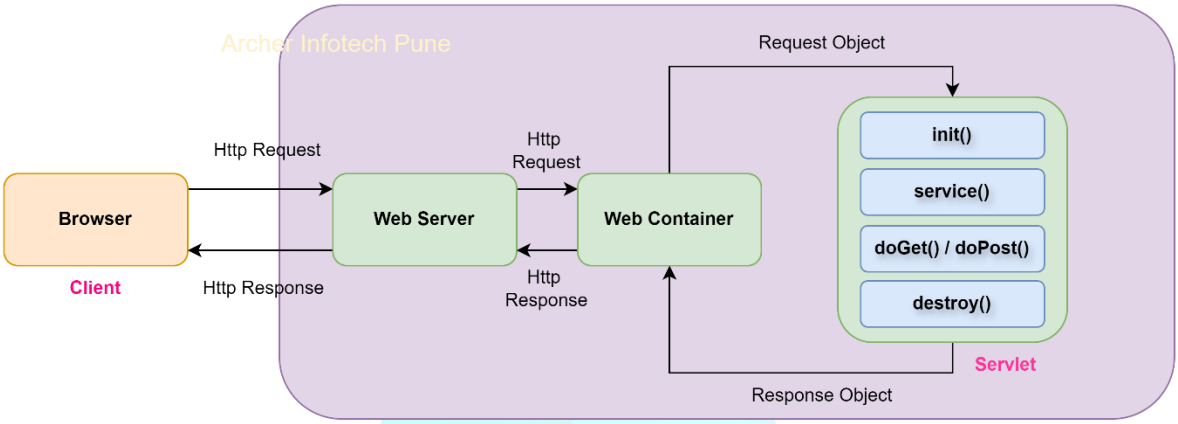
1. Loading and Instantiation:
2. Initialization (init method)
3. Request Handling (service method)
4. Request Handling Methods (doGet, doPost, etc.)
5. Destruction (destroy method)

1. **Loading and Instantiation:**
   - **Loading**: The servlet container (e.g., Apache Tomcat) loads the servlet class into memory. This typically happens when the servlet is first requested or when the container starts up, depending on the configuration.
   - **Instantiation**: The servlet container creates an instance of the servlet class using the default constructor. This instance is used to handle all requests to that servlet.
2. **Initialization (init method):**
   The init() method is called once after the servlet is instantiated. It is used to perform any initialization tasks, such as setting up database connections, loading configuration files, or initializing resources.
   - **Signature**: public void init(ServletConfig config) throws ServletException
   - **Parameters**:

- **ServletConfig config**: Provides the servlet with information about its initialization parameters and the servlet context.

- **Exceptions**:
  - **ServletException**: Thrown if the servlet encounters an error during initialization.
- **Example**

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    // Initialization code here
}
```



Servlet Life Cycle

3. **Request Handling (service method):**
   The service method is called for each request the servlet receives. It determines the type of request (e.g., GET, POST) and delegates the request to the appropriate method (doGet, doPost, etc.).
   - **Signature:** public void service(ServletRequest req, ServletResponse res) throws ServletException, IOException
   - **Parameters:**
     - **ServletRequest req:** Represents the request from the client.
     - **ServletResponse res:** Represents the response to the client.
   - **Exceptions:**
     - **ServletException:** Thrown if the servlet encounters an error while handling the request.
     - **IOException:** Thrown if an I/O error occurs while handling the request.
   - **Example:**

```
protected void service(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
    String method = req.getMethod();
    if (method.equals("GET")) {
        doGet(req, res);
    } else if (method.equals("POST")) {
        doPost(req, res);
```

```
    } else {
        super.service(req, res);
    }
}
```

4. **Request Handling Methods (doGet, doPost, etc.):**
   These methods handle specific types of HTTP requests. The most commonly used methods are doGet and doPost.
   - **Signatures**:

     protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException

     protected void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException

   - **Parameters:**
     - **HttpServletRequest req:** Represents the HTTP request from the client.
     - **HttpServletResponse res:** Represents the HTTP response to the client.
   - **Exceptions:**
     - **ServletException**: Thrown if the servlet encounters an error while handling the request.
     - **IOException**: Thrown if an I/O error occurs while handling the request.
   - **Example**:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html><body>");
    out.println("<h1>Hello, World!</h1>");
    out.println("</body></html>");
}
protected void doPost(HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {
    // Handle POST request here
}
```

5. **Destruction (destroy method)**
   The destroy method is called once when the servlet is being taken out of service. This method is used to perform any cleanup tasks, such as closing database connections or releasing resources.
   Signature: *public void destroy()*
   Example:

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

Key Features of Servlets

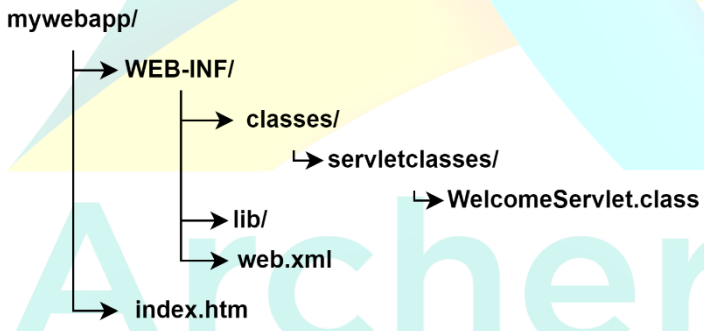
- **Platform Independence:** Servlets are written in Java, which makes them platform-independent. They can run on any server that supports the Java Servlet API.
- **Dynamic Content Generation:** Servlets can generate dynamic content based on user input, database queries, or other factors.
- **Session Management:** Servlets can manage user sessions, allowing for stateful interactions between the client and the server.
- **Scalability:** Servlets are designed to handle multiple requests concurrently, making them suitable for high-traffic web applications.
- **Integration with Java EE:** Servlets can easily integrate with other Java EE technologies like JavaServer Pages (JSP), Enterprise JavaBeans (EJB), and Java Message Service (JMS).

Servlet Welcome Program:

Creating a simple "Welcome" program using a servlet involves several steps, including setting up the development environment, writing the servlet code, configuring the deployment descriptor, and deploying the servlet to a servlet container. Below is a detailed procedure to create a "Welcome" program in a servlet:

- Install JDK: https://www.oracle.com/in/java/technologies/downloads/
- Apache tomcat download: https://tomcat.apache.org/download-90.cgi

1. **Set Up the Project Structure:** Create a directory structure for your web application. The typical structure for a web application looks like as:



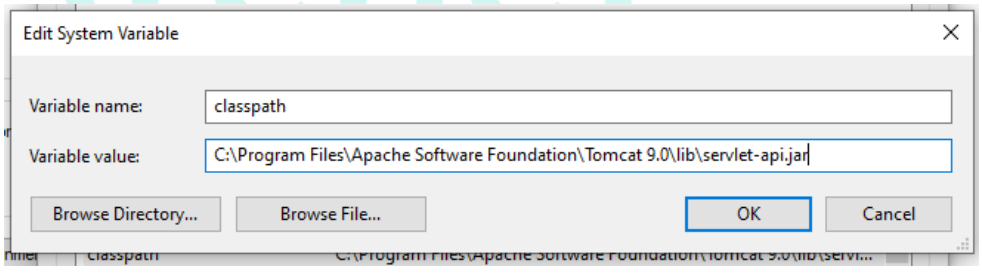
2. **Write the Servlet Code:** Create a Java class for your servlet. For example, create a file named WelcomeServlet.java in the classes.servletclasses package.

```
package servletclass;

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

public class WelcomeServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h1>Welcome to the Servlet World!</h1>");
      out.println("</body></html>");
   }
}
```

3. **Compile the Servlet:** Compile the servlet code to generate the .class file. You can use the javac command from the command line or compile it within your IDE.
   For that,
   - download the servlet-api.jar, Place it into WEB-INF/lib, go to the classes and open cmd at same path and compile using cmd as,
     C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\FirstServlet\WEB-INF\classes>javac -classpath "C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\FirstServlet\WEB-INF\lib\servlet-api.jar"; servletclass\WelcomeServlet.java

     Or
   - Use the servlet-api.jar from Tomcat xx\bin as,
     C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\FirstServlet\WEB-INF\classes>javac -classpath "C:\Program Files\Apache Software Foundation\Tomcat 9.0\lib\servlet-api.jar"; servletclass\WelcomeServlet.java

     Or
   - Simply set classpath in Environment variable as,



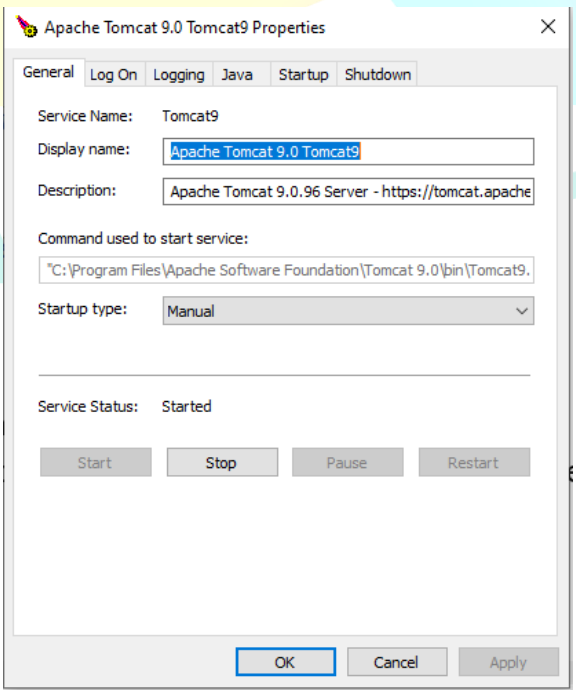
4. **Configure the Deployment Descriptor (web.xml):** Create a web.xml file in the WEB-INF directory to configure the servlet.

```
<web-app>
  <!-- Servlet Declaration -->
  <servlet>
    <servlet-name>WelcomeServlet</servlet-name>
    <servlet-class>servletclass.WelcomeServlet</servlet-class>
  </servlet>

  <!-- Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>WelcomeServlet</servlet-name>
    <url-pattern>/welcome</url-pattern>
  </servlet-mapping>
</web-app>
```

5. **Create an HTML File (Optional):** Create an index.html file in the root directory of your web application to provide a link to the servlet.

```
<!DOCTYPE html>
<html>
<head>
  <title>Welcome Page</title>
</head>
<body>
  <h1>Welcome to the Web Application@Archer Infotech Pune</h1>
  <a href="welcome">Go to Welcome Servlet</a>
</body>
</html>
```

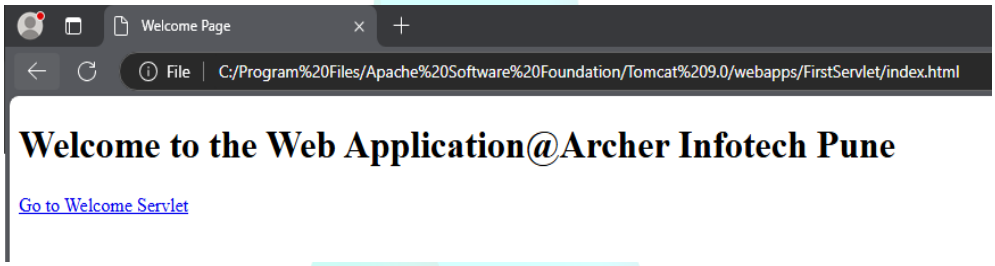
6. **Start the Servlet Container:** Start your servlet container (e.g., Apache Tomcat). You can start Tomcat using the startup.sh or startup.bat script in the bin directory of your Tomcat installation.



7. **Access the Servlet**: Open a web browser and navigate to the URL of your servlet. For example:

Or

Open the index.html present in the ..\webapps\FirstServlet folder



Eclipse and Apache Tomcat Configuration:

- **Download and install the Eclipse and Apache Tomcat.**
  - Eclipse IDE for Enterprise Java and Web Developers: https://www.eclipse.org/downloads/packages/release/2024-09/r/eclipse-ide-enterprise-java-and-web-developers
  - apache tomcat 9: https://tomcat.apache.org/download-90.cgi
- **Set Runtime Environment of Tomcat in Eclipse.**
  - In Eclipse, go to **Window** > **Preferences**.
  - In the Preferences dialog, navigate to **Server** > **Runtime Environments.**
  - Click the **Add** button.
  - In the New Server Runtime Environment dialog, select **Apache** > **Tomcat vX.X Server** (where X.X is the version you downloaded) and click **Next**.
  - In the Apache Tomcat dialog, click the **Browse** button and navigate to the directory where you extracted Tomcat (e.g., **C:\apache-tomcat-9.0.50**).
  - Click **Finish**.
- **Add the server.**
  - In Eclipse, go to the **Servers** tab (if it's not visible, you can open it via **Window** > **Show View** > **Servers**).
  - Right-click in the Servers tab and select **New** > **Server**.
  - In the New Server dialog, select **Apache** > **Tomcat vX.X** Server (where X.X is the version you configured) and click **Next**.
  - In the Server's host name field, you can leave the default value (**localhost**).
  - Click **Next**.