Client and Server Architecture

A client-server architecture is a networking model in which the server provides services to clients to perform user-based tasks. A client and a server are two pieces of software that might be on the same computer, or two different computers that might be separated by miles but connected by the Internet.

Server - A server is software designed to process requests and deliver responses to another computer over the internet.

Client - A client is a program that runs on a local machine requesting service from the server.

A Client and a Server establish a connection according to a set of rules called a protocol. There are quite a few protocols for different purposes, but one of the most popular is the **HTTP protocol**. Once the connection is established, the Client sends **HTTP Requests** to the server in the form of XML or JSON, which both entities (Client and Server) understand. After parsing the request, the Server responds with appropriate data by sending back an **HTTP Response**.

Types of Client-Server Architecture

2 tier architecture - The user interface stored at the client machine and the database stored on the server. If Business Logic & Data Logic collected at a client-side, then it is known as a *fat client thin server architecture*. If Business Logic & Data Logic handled on the server, then it is known as a *thin client fat server architecture*. 2 tier architecture has some limitations in performance, security, and portability.

3 tier architecture - Three-tier architecture has a middleware between the user interface and database. The 3 tiers are named the presentation tier, application tier, and data tier. The presentation tier is the front end layer and consists of the user interface. The application tier contains the functional business logic which drives an application’s core capabilities. The data tier consists of a database system and the data access layer.

n-tier architecture - In n-tier architecture, there are multiple Business Logic & Data Logic layers. It increases the flexibility and reusability of applications but can be difficult to implement.

Servlets

A website can consist of both static and dynamic webpages. A static webpage is a pre-built HTML page with the content explicitly written into the code, and stored in the webserver. Static web pages display the same content each time we visit. A dynamic webpage loads dynamic content *such as stock prices, weather information, news, and sports updates* at different points of time. In Java, there exists a way to generate static webpages with dynamic data, and that's with **Java Servlets**.

A **Servlet** is a Java class that takes incoming requests, processes them, and generates a response to send back to the user. For example, an `HttpServlet` takes an HTTP request, processes its headers and content, and uses that information to write HTML, CSS, and JavaScript code into an HTTP response that can be sent back to the user's browser.
The **Servlet container** is the component of an **application server** that interacts with Java servlets and is responsible for managing the execution of servlets and JSP pages for Java applications.

```
User  -->  HTTP Request  -->   Application Server  -->  Servlet Container
                                                                |
                                                                V
User  <--  HTTP Response  <--  Application Server    <--   Servlet(s)
```

How do servlets work?

When a client sends a request to the application server, the application server receives and passes the request to the appropriate servlet. The servlet processes the request, generates the response, and sends the response back to the application server. The application server sends the response back to the client. Most servlets are HTTPServlets, which receive HTTP requests and generate HTTP Responses out of HTML, CSS, and JavaScript code.

References

- Java Servlet Technology Overview
- Introduction to Servlets - Video Tutorial
- Servlet Documentation - Oracle

Installation: Tomcat server

Apache Tomcat

Apache Tomcat is an **open-source** implementation of the Java Servlet, Java Server Pages, Java Expression Language, and WebSocket technologies. Apache Tomcat is an **application server** that allows us to run servlet and JavaServer Pages (JSP) based web applications. The default port for the Apache Tomcat service is **8080**.

Installing Apache Tomcat web server in Eclipse IDE

Before installing Apache Tomcat, ensure that the JDK installed and the Java environment variable configured.

Step 1: Download Apache Tomcat server from tomcat.apache.org. Extract files from the downloaded zip file.

Step 2: In the Eclipse IDE, click on the **Servers** Tab. You'll find a link to create a new server. Click on that will open a New Server dialog box. In that dialog box, select the server type. Select **Apache -> Tomcat v9.0 Server** and click Next.

If the **Servers** tab is not visible, it can be configured by navigating to the *Window* tab in the Eclipse navbar. Click on the **Window** tab and select **Show View** -> **Other**. In the *Show View* dialog box, select ***Servers*** under the **Server** Folder and click **Open**. Now, you able see the *Servers* tab on the Eclipse window.

Step 3: Then, specify the Tomcat installation directory. Just browse the location where you extracted the `apache-tomcat-9.0.26` files and then click FINISH. Under Server Tab, you should see *Tomcat v9.0 Server at localhost [Stopped, Republish]*.

Step 4: Right-click on the server runtime and click Start to start the Tomcat server. Double click on the *Tomcat v9.0 Server at localhost [Started, Synchronized]* to view the HTTP port on which the tomcat server is running, and its configurations.

Here, we can configure:

- Server Name - The name of the server which appears under the server tab.
- Configuration Path - used to specify the location of the server configuration.
- Server Locations – used to specify the server path (location of the Apache Tomcat server installed) and the deploy path.
- Publishing – used to configure how the modules are publishing.
- Timeouts - used to specify the time limit to complete the server operations. We can set the timeouts for starting/stopping the server. If the server timeout is very low, the server might fail to start.
- Ports – used to set the server ports.
- MIME mappings – used to set the various MIME type mappings.
- Server Launch Configuration – used to configure the VM arguments, classpath, etc.
- Server Options – used to enable/disable features like security, auto-reload of modules by default, etc.

Step 5: The Tomcat server is up and running on port *8080*. To verify, open your browser and visit http://localhost:8080/. You may see the HTTP-404 error page, but that response should come from the Apache Tomcat server.

You can also view the *Apache Tomcat/9.0.26* home page by selecting the *Use Tomcat installation* option under the *Server Locations*.

Restart the *Tomcat server* and visit http://localhost:8080/, you'll see the *Apache Tomcat/9.0.26* home page.

References

- Apache Tomcat 9 Docs
- Apache Tomcat 9 Configuration Reference
- Video Tutorial - Install Apache Tomcat in Eclipse IDE

Life Cycle of a Servlet

A **servlet container** manages the life cycle of a servlet. Servlet is an interface defined in `javax.servlet` package. A servlet container uses the Servlet interface to understand a specific Servlet object and manage it.

There are three life cycle methods of a Servlet:

- `init()`
- `service()`
- `destroy()`

The steps involved in the servlet life cycle are listed below:

Step-1: Loading of Servlet

When the application server (e.g. Apache Tomcat) starts up, the servlet container deploys and loads all the servlet classes.

Step-2: Creating an instance of Servlet

Once all the Servlet classes are loaded, the servlet container creates only one instance for each servlet class. All requests to the servlet are executed on that same servlet instance. Some application servers can create multiple instances of a servlet to handle a high volume of incoming requests, but that is not the default behavior.

Step-3: Invoke `init()` method once

Once all the servlet classes are instantiated, the init() method is invoked for each instantiated servlet. The `init()` method is used to initialize the servlet. The `init()` method is called only once.

The `init()` method signature:

```
public void init() throws ServletException {

}
```

Step-4: Invoke `service()` method repeatedly for each client request

The servlet container calls the service method each time a request for the servlet is received. The service() method determines the type of Http request (GET, POST, PUT, DELETE, etc.) also calls `doGet()`, `doPost()`, `doPut()`, `doDelete()`, etc. methods as appropriate.

The `service()` method signature:

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

}
```

Step-5: Invoke destroy() method once

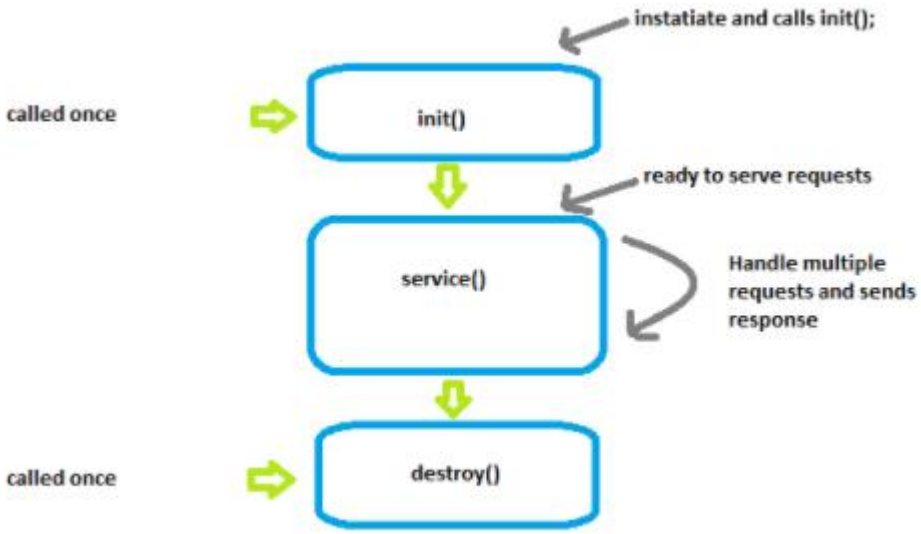
The `destroy()` method is called only once at the end of the servlet's life. The servlet container calls this method before removing the servlet instance from the service.

The `destroy()` method signature:

```
public void destroy() {

}
```

Life Cycle of a Servlet:



Servlet API

The Servlet API provides interfaces and classes that are required to build servlets. These interfaces and classes represented in two packages:

1. `java.servlet` package - used by the servlet or web container.
2. `javax.servlet.http` package - used for handling http requests.

Servlet Class Hierarchy:

The Servlet interface is the root interface of the servlet class hierarchy. The GenericServlet class implements Servlet, ServletConfig, and Serializable interfaces. The HttpServlet class extends the GenericServlet class and implements the Serializable interface. It provides HTTP methods such as doGet, doPost, doHead, doTrace, etc.

Java Servlet Class Hierarchy:

```
   Servlet    ServletConfig       Serializable                    {Built-in Interface
s}
     |             |                    |
     ----------------------------------
                    |
                    V
              GenericServlet                                       {Built-in Class}
                    |
                    V
               HttpServlet                                         {Built-in Class}
                    |
                    V
             UserDefinedServlet                                     {User defined ser
vlet Class}
```

The user defined servlet class is created by implementing the *Servlet* interface, usually by extending the *GenericServlet* class or (more commonly) the *HttpServlet* class.

In order to initialize a Servlet, a server application loads the user-defined servlet class and creates an instance. Then it calls the Servlet's `init (ServletConfig config)` method. Since the `init()` method is run once, it stores the initial parameters or configuration information in the *ServletConfig* object. This information can be retrieved later by calling the Servlet's `getServletConfig()` method. This is implemented in the *GenericServlet* class definition. The *ServletConfig* object contains Servlet parameters and a reference to the Servlet's *ServletContext*. The **ServletContext** is an interface which helps to communicate with other servlets. Then, the `service (ServletRequest request, ServletResponse response)` method is called for every request to the Servlet. When the Servlet needs to be unloaded the `destroy()` method is called.

Declaring and mapping servlets

To configure a servlet in the web.xml file,

1. Configure the servlet using the `<servlet>` element.
2. Map the servlet to a URL or URL pattern using the `<servlet-mapping>` element.

The `<servlet>` element is used to declare the servlet name, the fully-qualified class name of the servlet, and any initialization parameters. The name for every servlet must be unique across the deployment descriptor.

The `<servlet-mapping>` element is used to specify a URL pattern, and the name of the servlet which handles requests whose URL matches the given pattern. The URL pattern uses an asterisk (*) at the beginning or end of the pattern to indicate zero or more of any character.

Example: A simple *web.xml* file which has `<servlet>` and `<servlet-mapping>` for *servlet1*.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

<servlet>
        <servlet-name>servlet1</servlet-name>
        <servlet-class>com.revature.MyFirstServlet</servlet-class>
</servlet>

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

</web-app>
```

Deployment descriptor (web.xml)

Deployment descriptor

Java web applications use a **deployment descriptor file** to define the URLs that map to servlets, and to determine which URLs require authentication and additional information.

A deployment descriptor file specifies the classes, resources, and configuration of the application and how the web server uses them to serve HTTP requests.

The deployment descriptor is a file named **web.xml**. It resides within the app's WAR beneath the WEB-INF/ directory. The root element of the web.xml file is `<web-app>`.

The **web.xml** file defines mappings between URL paths and the servlets that will handle requests with those paths. The application server uses this configuration to find the servlet that handles a given request, and calls the servlet method that corresponds to the HTTP request method used.

To map a URL to a servlet, you declare the servlet with the `<servlet>` element, then define a mapping from a URL path to a servlet declaration with the `<servlet-mapping>` element.

Below we have simple **web.xml** file that maps all URL paths (/*) to the servlet class `mysite.server.MyFirstServlet`.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">
    <servlet>
        <servlet-name>servlet1</servlet-name>
        <servlet-class>com.revature.MyFirstServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>servlet1</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Form Processing

Handling form data from the request

In this section, we'll discuss how to handle HTML form data from on the server-side with a Java Servlet. When a user fills in the fields of a form and submits it, the servlet processes the request based on the submitted data and sends a response back to the client.

- Right click on the **webapp** folder and click New -> HTML File. If you do not see the option to create a new HTML file, click Other and search for HTML.

- Name the new file `add.html` and click `Finish`.

To make the form work with a Java servlet, we need to specify the following attributes for the `<form>` tag:

- **method="post" or method="get"** : to send the form data as an HTTP POST or HTTP GET request to the server.
- **action= "URL of the servlet"**: specifies URL of the servlet which is responsible for handling form data.

For example,'*add.html*' uses a `<form>` element for adding two numbers:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Addition</title>
</head>
<body>
	<form method="get" action="add">
		Enter Number 1: <input name="num1" /> <br/>
		Enter Number 2: <input name="num2" /> <br/>
		<input type="submit" value="Add">
	</form>
</body>
</html>
```

We have to configure the **web.xml** file, so that the servlet container sends the form's submit request to the corresponding servlet.

For example,'*web.xml*' file maps the '*/add*' URL pattern to the '*AddServlet*' class:

```
<servlet>
    <servlet-name>servlet1</servlet-name>
    <servlet-class>AddServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>servlet1</servlet-name>
    <url-pattern>/add</url-pattern>
</servlet-mapping>
```

The servlet handles **GET** requests using the `doGet()` method and **POST** requests using the `doPost()` method. The servlet uses the `HttpServletRequest.getParameter()` method to get the value of form parameters like *num1* and *num2*. It does so by using the **name** attribute of the HTML `<input>` elements in the form.

For example, '*AddServlet.class*' handles the form data:

```
public class AddServlet extends HttpServlet {


    public void doGet(HttpServletRequest req,HttpServletResponse res) throws ServletE
xception,IOException
       {


        int i = Interger.parseInt(request.getParameter("num1"));
        int j = Interger.parseInt(request.getParameter("num2"));


        int k = i + j;
        response.setContentType("text/plain");


        PrintWriter out = response.getWriter();
        out.println("The sum is: " + k);


     }
}
```

Retrieving request parameters from the request

Servlets use the following methods for retrieving request/form parameters from the HTTP request:

- `getParameter()` method - used to get the value of a specified parameter.
- `getParameterValues()` method - used to get the multiple values of a specified parameter.(for example checkbox)
- `getParameterNames()` method - used to get complete list of all parameters.

Example *HTML form*:

```
<p> Enter the following details</p>
<form method="post" action="details">
        Name: <input type="text" name="name"/>
        Age: <input type="password" name="age"/>
        Gender:
        <input type="radio" name="gender" value="male" />Male
        <input type="radio" name="gender" value="female" />Female
        Speaking language:
        <input type="checkbox" name="language" value="english" />English
        <input type="checkbox" name="language" value="french" />French
        <input type="submit" />
</form>
```

The *web.xml* file:

```
<servlet>
    <servlet-name>servlet2</servlet-name>
    <servlet-class>MyServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>servlet2</servlet-name>
    <url-pattern>/details</url-pattern>
</servlet-mapping>
```

Servlet example for retrieving request parameters from the request using the `getParameter()` and `getParameterValues()` methods:

```
public class MyServlet extends HttpServlet {


    public void doGet(HttpServletRequest req,HttpServletResponse res) throws ServletE
xception,IOException
   {
                String name = request.getParameter("name");
```

```
                int age = Interger.parseInt(request.getParameter("age"));
                String gender = request.getParameter("gender");
                String languages[] = request.getParameterValues("language");



                response.setContentType("text/plain");

                PrintWriter out = response.getWriter();
                out.println("Your details:")
                out.println("Name: " + name);
                out.println("Age: "+ age);
                out.println("Gender: "+ gender);
                if (languages != null) {
                        out.println("Languages are: ");
                        for (String lang : languages) {
                                out.println(lang);
                        }
                }
        }
}
```

ServletConfig and ServletContext parameters

ServletConfig

ServletConfig is an object created by the Servlet Container, used to pass initial parameters or configuration information to a particular servlet during initialization. The `<servlet>` XML element in the deployment descriptor (web.xml) has a sub element called `<init-param>` used to pass parameters to a servlet from the web.xml file. The ServletConfig object is returned by the `getServletConfig()` method of HttpServlet, and so the XML properties added to ServletConfig are only shared with the single servlet indicated.

ServletContext

ServletContext is the object created by the Servlet Container to share initial parameters or configuration information to *all* servlets and other components. The `<context-param>` element used to declare the parameters of ServletContext. It present outside

the `<servlet>` element and inside the `<web-app>` element. This object is returned by the `getServletContext()` method of HttpServlet.

The `<param-name>` and `<param-value>` used to declare the parameter name and its value.

ServletConfig vs. ServletContext

Example

Here '**message**' parameter can be accessed only by *servlet1*. The '**username**' and '**password**' parameters can be accessed by both *servlet1* and *servlet2*.

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
         http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
         version="3.1">

        <servlet>
                <servlet-name>servlet1</servlet-name>
                <servlet-class>com.revature.MyFirstServlet</servlet-class>
                <init-param>
                        <param-name>message</param-name>
                        <param-value>Hello World</param-value>
                </init-param>
        </servlet>
        <servlet-mapping>
                        <servlet-name>servlet1</servlet-name>
                        <url-pattern>/FirstServlet</url-pattern>
        </servlet-mapping>

        <servlet>
                        <servlet-name>servlet2</servlet-name>
                        <servlet-class>com.revature.MySecondServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                        <servlet-name>servlet2</servlet-name>
                        <url-pattern>/secondServlet</url-pattern>
```

```
        </servlet-mapping>

        <context-param>
                <param-name>username</param-name>
                <param-value>system</param-value>
        </context-param>

        <context-param>
                <param-name>password</param-name>
                <param-value>pass123</param-value>
        </context-param>

</web-app>
```

Hidden Form Fields

Hidden fields can be inserted into webpages by the server for session tracking.
These fields are not visible directly to the user but can still be viewed using the *view source* option from the browser. Therefore, hidden fields should *not* be used as a form of security.

Hidden fields may be used to send information that is only pertinent to the server, and not the client.

A web server can send a hidden HTML form field along with a unique session ID:

```
<input type = "hidden" name = "session_id" value = "65349">
```

This hidden field is not displayed on the browser but the value is sent to the server when the parent `<form>` element is submitted.

The server retrieves this hidden form field value using
the `request.getParameter("session_id")` method in a servlet.

Request Dispatcher

The Request Dispatcher interface defines an object that passes along the client's request to any other resources (servlet, JSP file, or HTML file) on the server.

The servlet container creates the RequestDispatcher object.
The `getRequestDispatcher(String)` method of the **ServletRequest** interface returns the Request Dispatcher object.

RequestDispatcher interface defined in the *javax.servlet* package provides two methods:

1. forward(ServletRequest request, ServletResponse response)

`forward()` passes a request from one servlet to another resource on the server. The contents of the request and response are preserved and forwarded to the next resource which will process the data and return the response to the client.

Example for forward() method:

```
//"request" is a HttpServletRequest Object and "Welcome.html" is a resoure name
RequestDispatcher rd = request.getRequestDispatcher("Welcome.html");

//forward the request and response to "Welcome.html" page
rd.forward(request,response);
```

2. include(ServletRequest request, ServletResponse response)

`include()` does not entirely transfer control over the request and response object to the next resource. Instead, this method *includes* the content of the original resource in the response returned to the client. If you `include()` a servlet or JSP document, the included resource may not change the response code or HTTP headers.

Example for include() method:

```
//"request" is an HttpServletRequest Object and "hello.html" is a resource name
RequestDispatcher rd = request.getRequestDispatcher("hello.html");

//includes the response of "hello.html" page in current servlet response
rd.include(request,response);
```

References

- RequestDispatcher Documentation

Writing plain text to the response Object

The Servlet API provides *HttpServletResponse* interface which extends the *ServletResponse* interface to assist in sending a response to the client.

The Servlet Container informs the client browser about the type of data in the response before sending it. The Servlet container uses the `setContentType()` method to set the type of data in the response object. The response data can be in simple plain text format, HTML format, XML format, an image format, etc.

The `setContentType(String type)` method **sets the content type** of the response being sent to the client before sending the response. The`response.getWriter()` method returns a **PrintWriter** object, which sends character text to the client.

Example:

```
public class HelloWorld extends HttpServlet {


    public void doGet(HttpServletRequest req,HttpServletResponse res) throws ServletE
xception,IOException
       {
         // set the content type as plain text
         response.setContentType("text/plain");


         PrintWriter out = response.getWriter();
         out.println("Hello World");  // writes a plain text to the response object
     }
}
```

In the above code, the MIME type "*text/plain*" can be broken down as follows: "text" is known as the **type** and "plain" is known as a **subtype**. A type contains many subtypes. Some of the types used in the `setContentType()` method are "*text/html*" used for writing HTML content to the response object, "*text/css*" for writing CSS content to the response object, etc.

SendRedirect in servlets

sendRedirect(String URL) - This method defined in **HttpServletResponse** interface and used to redirect a response to another resource. It uses the URL to make another request. Therefore, it works at the client-side also can work inside and outside the server.

The `RequestDispatcher.forward()` method is used to pass the same request to a new destination resource, but the `Response.sendRedirect()` method is used to send an entirely

new request for the destination resource. Any request attributes or parameters from the original request are lost.

Example for sendRedirect():

```
//"response" is a HttpServletRequest Object redirected to the google server.
response.sendRedirect("http://www.google.com");
```

Resources

- ServletResponse Oracle Documentation
- PrintWriter Oracle Documentation
- HttpResponse Object Examples