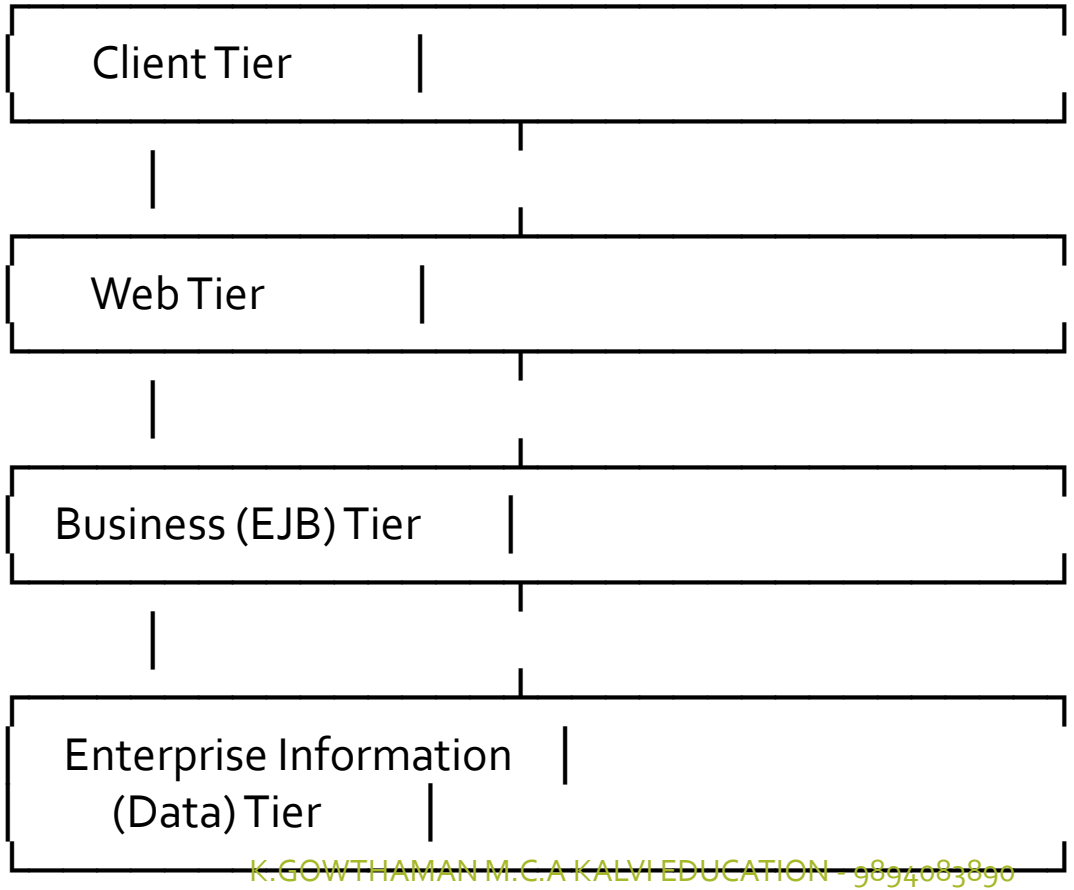
JAVA SERVLETS

What is Java EE (Jakarta EE)?

Java EE (Enterprise Edition) is a **set of specifications** that extends **Java SE** (Standard Edition) with features for building **distributed, multi-tier, web-based, and enterprise applications**.

It provides APIs and runtime environments for:

- Web applications
- Enterprise-level services
- Transaction management
- Database connectivity
- Messaging systems



Java EE Application Server Examples

Application servers implement the **Java EE specifications** and provide all container services.

Popular servers:

- **Apache TomEE**
- **GlassFish**
- **WildFly (formerly JBoss)**
- **IBM WebSphere**
- **Oracle WebLogic**
- **Payara Server**

Advantages of Java EE

- ✓ **Platform-independent** (Write once, run anywhere)
- ✓ **Scalable and distributed**
- ✓ **Built-in security and transaction management**
- ✓ **Component-based architecture (easy maintenance)**
- ✓ **Supports web services (SOAP & REST)**
- ✓ **Reusable and modular code**

Layer	Example Components
Client Tier	Web browser or mobile app
Web Tier	JSP for login page, Servlet for requests
Business Tier	EJB for account management & transactions
Data Tier	MySQL Database via JPA

Tier	Description	Technologies
Client Tier	End-user interaction	HTML, JS, JavaFX
Web Tier	Presentation logic	Servlet, JSP, JSF
Business Tier	Business logic	EJB, CDI
Data Tier	Data storage	JDBC, JPA

How It All Works (Flow)

1. **Client Tier** sends request (e.g., via browser).
2. **Web Tier** (Servlet/JSP) handles input and calls business logic.
3. **Business Tier** (EJBs) processes data and interacts with database.
4. **Data Tier** retrieves or updates data.
5. Response flows back up to the **Client Tier**.

Interfaces in Servlet API

Interface	Description
Servlet	The base interface. All servlets implement this.
GenericServlet	Abstract class implementing Servlet, supports protocol-independent servlets.
HttpServlet	Extends GenericServlet for HTTP-specific functionality (GET, POST, etc.)
ServletRequest	Encapsulates client request data (parameters, headers).
ServletResponse	Encapsulates response data (HTML, JSON, etc.).
ServletConfig	Configuration info for a servlet.
ServletContext	Application-wide info shared among servlets.

Application Server in Java EE

An **Application Server** is a **software framework** that provides the **runtime environment** for Java EE applications.

It **implements the Java EE specifications,** providing services like:

- Web request handling
- Business logic execution
- Transaction management
- Security
- Persistence and database connectivity
- Messaging

Containers in Java EE

A **Container** is a **runtime environment** within the application server that provides **services to components** (like Servlets, EJBs).

It **manages lifecycle, security, transactions, and other services,** so developers don't have to handle them manually.

Container	What It Manages	Components Supported	Key Services
Web Container (Servlet Container)	Manages web components	Servlets, JSP, JSF	HTTP requests, sessions, JSP compilation, security

Web Container (Details)

- **Handles HTTP requests and responses**
- **Manages session tracking and cookies**
- **Converts JSP pages into Servlets at runtime**
- **Provides lifecycle callbacks: init(), service(), destroy()**

Example: Apache Tomcat (Servlet container) is sometimes called a web container

What is a Servlet?
A **Servlet** is a **Java class** that handles **HTTP requests and responses** on a web server.
It runs inside a **Web Container** (like Tomcat, Jetty, or GlassFish) and is used to build **dynamic web content**.

Servlets are the backbone of **Java web applications**, often working with JSP/JSF to generate dynamic HTML pages.

The **Servlet API** is provided by the package:

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

It defines **interfaces and classes** for handling:
•Request/Response objects
•Sessions
•Cookies
•Filters and listeners

Lifecycle of a Servlet

A servlet has a **well-defined lifecycle** managed by the **Web Container**:

1. Loading and Instantiation
   ▼
2. Initialization (init())
   ▼
3. Request Handling (service())
   ▼
4. Destruction (destroy())

1. init()

- Called **once** when servlet is loaded.
- Used to perform **initialization** tasks.

```
public void init() throws ServletException
{
   // initialization code
}
```

2. service()

- Called every time a request arrives.
- Handles GET, POST, PUT, DELETE requests.
- In HttpServlet, you override doGet() or doPost() instead.

```
protected void doGet(HttpServletRequest req,
HttpServletResponse res)
      throws ServletException, IOException {
   res.getWriter().println("Hello, Servlet!");
}
```

3. destroy()

- Called **once** when servlet is unloaded.
- Used for **cleanup**, like closing resources.

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

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

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

Steps to Deploy:

1.Place .class file in WEB-INF/classes folder of web app.

2.Configure in web.xml or use @WebServlet annotation:

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

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

http://localhost:8080/yourApp/hello

HTTP-Specific Methods in HttpServlet

Method	Purpose
doGet()	Handles HTTP GET requests
doPost()	Handles HTTP POST requests
doPut()	Handles HTTP PUT requests
doDelete()	Handles HTTP DELETE requests
doHead()	Handles HEAD request (headers only)
doOptions()	Returns supported HTTP methods
doTrace()	Returns diagnostic trace of request

Servlet Client Request and Response

```
String name = request.getParameter("name");
String ip = request.getRemoteAddr();
```

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<h1>Hello</h1>");
```

Session Management

- **Session Management**
- Servlet API provides **HttpSession** for maintaining state:

```
HttpSession session = request.getSession();
session.setAttribute("user", "Gowthaman");
String user = (String) session.getAttribute("user");
```

Servlet Context

•Used to share data across all servlets in a web app.

```
ServletContext context = getServletContext();
context.setAttribute("appName", "MyWebApp");
String app = (String) context.getAttribute("appName");
```

What is RequestDispatcher?
In **Java EE**, a **RequestDispatcher** is an interface that allows you to **forward a request from one servlet/JSP to another resource** (servlet, JSP, or HTML) or **include content** from another resource in the response.

Servlet API:
import javax.servlet.RequestDispatcher;

Methods of RequestDispatcher

Method	Description
forward(request, response)	Forwards request to another resource on the server. Client doesn't know about it.
include(request, response)	Includes content of another resource in the current response.

1) Forward() Method

- The control is transferred to another resource.
- The browser URL remains the same.
- Must be called before sending response to client (before response.getWriter().println()).

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

public class FirstServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {

      request.setAttribute("message", "Hello from FirstServlet");

      // Forward request to SecondServlet
      RequestDispatcher rd =
request.getRequestDispatcher("second");
      rd.forward(request, response);
   }
}

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

public class SecondServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      String msg = (String) request.getAttribute("message");
      out.println("<h1>Second Servlet: " + msg + "</h1>");
   }
}
```

2) include() Method

- Includes **output from another resource** into the **current response**.
- Useful for **headers, footers, menus**, or **shared content**.

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

public class HeaderServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      PrintWriter out = response.getWriter();
      out.println("<h2>Header Content</h2>");
   }
}

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

public class MainServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // Include header
      RequestDispatcher rd =
request.getRequestDispatcher("header");
      rd.include(request, response);

      out.println("<p>Main content of the page</p>");
   }
}
```

Servlet works with a Database (JDBC) in Java EE

A **Servlet** can connect to a **database** using **JDBC (Java Database Connectivity)** to:

- Store data (INSERT)
- Retrieve data (SELECT)
- Update data (UPDATE)
- Delete data (DELETE)

Steps in Servlet–Database Interaction

Load the JDBC Driver

Before connecting to the database, the driver must be loaded:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Establish a Connection

Use DriverManager to connect to the database.

```
Connection con = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/testdb", "root", "password");
```

Create a SQL Statement

Use a PreparedStatement or Statement to execute SQL queries.

```
PreparedStatement ps = con.prepareStatement("SELECT *
FROM users WHERE email=?");
ps.setString(1, email);
ResultSet rs = ps.executeQuery();
```

Process the Result

Handle the result set from the query.

```
if (rs.next()) {
   out.println("Welcome, " + rs.getString("name"));
} else {
   out.println("Invalid credentials!");
}
```

Close the Connection

Always close resources to avoid memory leaks.

```
rs.close();
ps.close();
con.close();
```

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

public class LoginServlet extends HttpServlet {

   protected void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

     response.setContentType("text/html");
     PrintWriter out = response.getWriter();

     String email = request.getParameter("email");
     String password = request.getParameter("password");

     try {
       // 1. Load driver
       Class.forName("com.mysql.cj.jdbc.Driver");

       // 2. Connect to DB
       Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/testdb", "root", "password");

       // 3. Prepare SQL query
       PreparedStatement ps = con.prepareStatement(
            "SELECT * FROM users WHERE email=? AND password=?");

       ps.setString(1, email);
       ps.setString(2, password);

       // 4. Execute query
       ResultSet rs = ps.executeQuery();

       if (rs.next()) {
         out.println("<h2>Welcome " + rs.getString("name") + "!</h2>");
       } else {
         out.println("<h2>Invalid email or password</h2>");
       }

       // 5. Close connections
       rs.close();
       ps.close();
       con.close();

     } catch (Exception e) {
       e.printStackTrace(out);
     }
   }
}
```

Cookies and Sessions in Servlets, two key mechanisms for **maintaining user state** across multiple HTTP requests.

Why We Need Cookies and Sessions
HTTP is stateless, meaning:
Each request from a client to the server is independent — the server does not remember previous requests.
To track user activity (like logins, preferences, cart items, etc.), we use:

- Cookies (client-side storage)
- Sessions (server-side storage)

What is a Cookie?
A **cookie** is a small piece of text data stored on the **client browser** by the server.
It travels with every request to the same domain.

Creating and Sending a Cookie

```
Cookie userCookie = new Cookie("username", "Gowthaman");
userCookie.setMaxAge(60 * 60 * 24);  // 1 day (in seconds)
response.addCookie(userCookie);     // Send cookie to client
```

Reading Cookies from Request

```
Cookie[] cookies = request.getCookies();
if (cookies != null) {
   for (Cookie c : cookies) {
      if (c.getName().equals("username")) {
         out.println("Welcome " + c.getValue());
      }
   }
}
```

Deleting a Cookie

```
Cookie c = new Cookie("username", "");
c.setMaxAge(0);  // Set expiry to 0
response.addCookie(c);
```

SetCookieServlet.java

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

public class SetCookieServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      Cookie c = new Cookie("user", "Gowthaman");
      c.setMaxAge(3600); // 1 hour
      response.addCookie(c);
      response.getWriter().println("Cookie set
successfully!");
   }
}
```

GetCookies

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

public class GetCookieServlet extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      response.setContentType("text/html");
      Cookie[] cookies = request.getCookies();
      if (cookies != null) {
         for (Cookie c : cookies) {
            if (c.getName().equals("user")) {
               response.getWriter().println("Welcome back, " + c.getValue());
            }  }  }  }
}
```

HttpSession in Servlet

📚 What is a Session?

A **session** is a server-side object used to **store user-specific data**.
It is created automatically when a user first interacts with your application.

Create or Retrieve Session

```
HttpSession session = request.getSession(); // Creates a session if one doesn't exist
```

Store Data in Session

```
session.setAttribute("username", "Gowthaman");
```

Retrieve Data from Session

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

Invalidate Session (Logout)

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

Session Timeout

```
session.setMaxInactiveInterval(600); // 10 minutes
```

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

public class LoginServlet extends HttpServlet {
   public void doPost(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      String user = request.getParameter("user");
      String pass = request.getParameter("pass");

      if ("admin".equals(user) && "1234".equals(pass)) {
         HttpSession session = request.getSession();
         session.setAttribute("username", user);
         response.sendRedirect("welcome");
      } else {
         response.getWriter().println("Invalid login!");
      }
   }
}
```

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

public class WelcomeServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {
      HttpSession session = request.getSession(false); // Don't
create new session
      if (session != null && session.getAttribute("username") !=
null) {
         String name = (String) session.getAttribute("username");
         response.getWriter().println("Welcome, " + name);
      } else {
         response.getWriter().println("Please login first!");
      }
   }
}
```

Cookies vs Session — Comparison Table

Feature	Cookies	Session
Storage location	Client browser	Server memory
Data capacity	Limited (~4KB)	Larger (depends on server)
Security	Less secure	More secure
Lifetime	Based on expiry time	Ends when session expires
Used for	Remember user preferences	Store login/user data

Reading / Writing Files on the Server

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

public class WriteFileServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {

      // Get a path inside the web application
      String filePath = getServletContext().getRealPath("/WEB-
INF/data.txt");

      // Write to the file
      try (FileWriter fw = new FileWriter(filePath, true)) {
         fw.write("User accessed: " + request.getRemoteAddr() +
"\n");
      }

      response.setContentType("text/html");
      response.getWriter().println("File updated successfully!");
   }
}
```

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

public class ReadFileServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {

      String filePath =
getServletContext().getRealPath("/WEB-INF/data.txt");
      response.setContentType("text/plain");

      try (BufferedReader br = new BufferedReader(new
FileReader(filePath))) {
         PrintWriter out = response.getWriter();
         String line;
         while ((line = br.readLine()) != null) {
            out.println(line);
         }
      }
   }
}
```

File Upload in Servlet (Multipart Form Data)

Modern servlets (3.0 and above) support file uploads **without external libraries**.

```
<form action="upload" method="post" enctype="multipart/form-data">
 Select file: <input type="file" name="file"><br><br>
 <input type="submit" value="Upload">
</form>
```

Servlet — Handle File Upload

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

@MultipartConfig(fileSizeThreshold = 1024 * 1024,    // 1MB
        maxFileSize = 1024 * 1024 * 10,     // 10MB
        maxRequestSize = 1024 * 1024 * 50)   // 50MB
public class FileUploadServlet extends HttpServlet {

  protected void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {

    // Get the uploaded file part
    Part filePart = request.getPart("file");
    String fileName = filePart.getSubmittedFileName();

    // Define where to save the file
    String uploadPath = getServletContext().getRealPath("") + File.separator + "uploads";
    File uploadDir = new File(uploadPath);
    if (!uploadDir.exists()) uploadDir.mkdir();

    // Save file on server
    filePart.write(uploadPath + File.separator + fileName);

    response.setContentType("text/html");
    response.getWriter().println("<h3>File uploaded successfully: " + fileName + "</h3>");
  }
}
```

Download File

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

public class FileDownloadServlet extends HttpServlet {
   public void doGet(HttpServletRequest request,
HttpServletResponse response)
         throws ServletException, IOException {

      String fileName = "data.txt";
      String filePath =
getServletContext().getRealPath("/WEB-INF/" +
fileName);

      response.setContentType("application/octet-
stream");
      response.setHeader("Content-Disposition",
"attachment;filename=" + fileName);

      try (FileInputStream in = new
FileInputStream(filePath);
          OutputStream out =
response.getOutputStream()) {

         byte[] buffer = new byte[4096];
         int bytesRead;

         while ((bytesRead = in.read(buffer)) != -1) {
            out.write(buffer, 0, bytesRead);
         }
      }
   }
}
```

Non-Blocking I/O (Asynchronous I/O) in **Servlets** — one of the advanced features introduced in **Servlet 3.1** (Java EE 7).

What is Non-Blocking I/O?

Traditional servlet I/O is blocking — meaning:

Enabling Asynchronous Support

You must enable asynchronous mode in your servlet:

```
@WebServlet(value = "/nonblocking", asyncSupported = true)
```

When a thread reads or writes data (like reading a large request body or writing a big response), it waits until the operation finishes.

This can cause scalability problems because each request blocks a thread until the I/O completes.

Non-blocking I/O allows the servlet container to reuse threads while waiting for I/O to complete, greatly improving performance for high-traffic or long-running requests.

Concept	Description
Async I/O (Asynchronous Processing)	Servlet can process a request in another thread (Servlet 3.0).
Non-Blocking I/O	Servlet can read/write data without blocking the thread (Servlet 3.1).
ReadListener	Interface for non-blocking input stream reading.
WriteListener	Interface for non-blocking output stream writing.

END