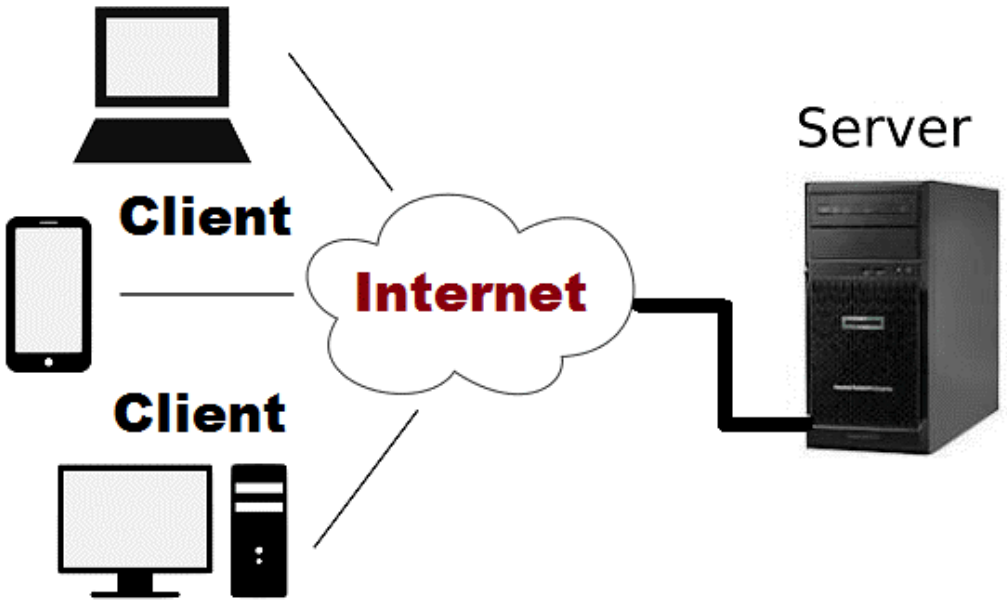
JSP and SERVLET

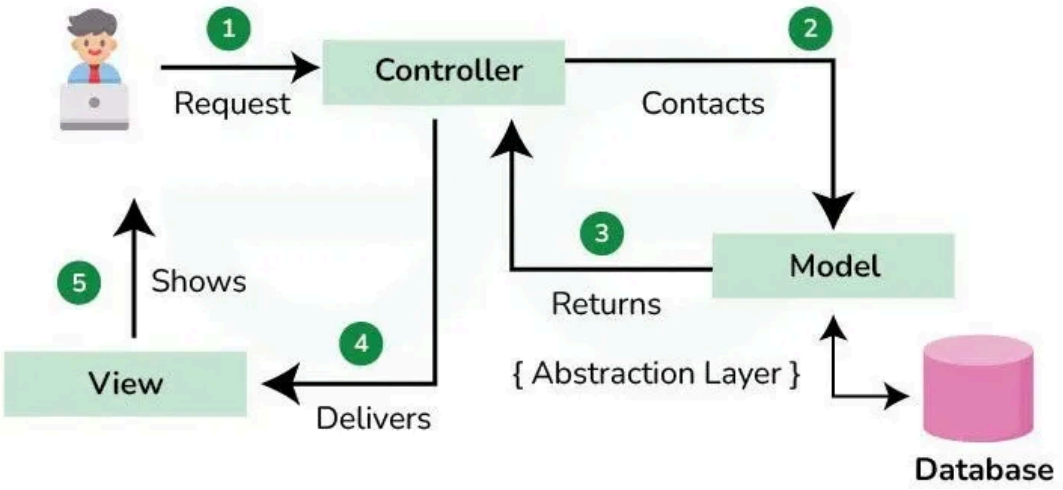
What is JSP?

JavaServer pages is a collection of technologies that helps software developers create dynamically generated web pages based on HTML, XML, SOAP, or other document types.

Client - Server Architecture



MVC Architecture



Why JSP?

JSP is easy to learn, easy to understand, and easy to develop.

JSP is a tag-based approach to develop a dynamic web application.

It’s extended to servlet and all its features.

It’s powerful because of the byte code.

Java code can be inserted in HTML/XML pages or both.

It has access to entire API of java.

So, it’s robust dynamic, secure, and platform-independent.

Use of JSP

- Ecommerce
- Management Site
- Social Site
- Travel Site

🛠️ Installing and Configuring Maven on Windows

1. Install Java (if not already)

- Maven requires Java.

- Check with:

  cmd

  `java -version`
- If not installed, download and install the latest JDK (e.g., from Adoptium).

2. Download Maven

- Go to the official site: Apache Maven Downloads.
- Download the **binary zip archive** (e.g., `apache-maven-3.x.x-bin.zip`).

3. Extract Maven

- Unzip to a folder, e.g.:

  Code

  `C:\Program Files\Apache\Maven`
- Inside, you'll see folders like `bin`, `conf`, `lib`.

4. Set Environment Variables

1. Open **Control Panel → System → Advanced system settings → Environment Variables**.
2. Under **System variables**, click **New**:
   - **Variable name:** `MAVEN_HOME`
   - **Variable value:** `C:\Program Files\Apache\Maven`
3. Edit the **Path** variable:
   - Add: `C:\Program Files\Apache\Maven\bin`
4. Also ensure your **JAVA_HOME** is set:
   - **Variable name:** `JAVA_HOME`
   - **Variable value:** path to your JDK (e.g., `C:\Program Files\Java\jdk-17`).

5. Verify Installation

- Close and reopen Command Prompt.
- Run:

  cmd

`mvn -v`

- **Expected Output:** Maven version, Java version, and OS info.

📋 Dynamic Web Project (Community Edition)

Step 1: Initialize Maven Web Project

- **Action:** Run in terminal:

  bash

  ```
  mvn archetype:generate ^
    -DgroupId=com.training ^
    -DartifactId=webapp-demo ^
    -DarchetypeGroupId=org.apache.maven.archetypes ^
    -DarchetypeArtifactId=maven-archetype-webapp ^
    -DinteractiveMode=false
  ```

- **Expected Output:** A folder `webapp-demo` with `src/main/webapp/index.jsp`.

Step 2: Open in IntelliJ IDEA Community

- **Action:** Open IntelliJ → *Open Project* → select `webapp-demo`.
- **Expected Output:** IntelliJ recognizes Maven and shows project structure.

Step 3: Add Source Directory

- **Action:** Create `src/main/java` for servlet classes.
- **Expected Output:** A folder ready to hold `.java` files.

Step 4: Configure `pom.xml`

- **Action:** Edit `pom.xml` to include Servlet API:

  xml

```
<packaging>war</packaging>
<dependencies>
 <dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>5.0.0</version>
  <scope>provided</scope>
 </dependency>
</dependencies>
```

- **Expected Output:** Maven reloads, dependency appears in External Libraries.

Step 5: Create Deployment Descriptor

- **Action:** Add `src/main/webapp/WEB-INF/web.xml`:

  xml

```
<web-app version="5.0">
 <servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.training.web.HelloServlet</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/hello</url-pattern>
 </servlet-mapping>
</web-app>
```

- **Expected Output:** IntelliJ recognizes `web.xml` under `WEB-INF`.

Step 6: Write a Servlet

- **Action:** Create `src/main/java/com/training/web/HelloServlet.java`:

  java

```
package com.training.web;
```

```
import jakarta.servlet.http.*;
import jakarta.servlet.ServletException;
import java.io.*;

public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse r
esp)
            throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<h1>Hello from Training WebApp</h1>");
    }
}
```

- **Expected Output:** Servlet compiles successfully.

Step 7: Update JSP

- **Action:** Edit `index.jsp`:

  jsp

```
<html>
  <body>
    <h2>Welcome</h2>
    <a href="hello">Go to HelloServlet</a>
  </body>
</html>
```

- **Expected Output:** JSP loads with a link to `/hello`.

Step 8: Build WAR

- **Action:** Run:

  bash

  `mvn clean package`

- **Expected Output:** `target/webapp-demo.war` created.

Step 9: Deploy to Tomcat

- **Action:** Copy WAR to Tomcat's `webapps` folder and start Tomcat:

  bash

  ```
  C:\Users\Admin\Desktop\JSPServlet\webapp-demo>copy target\webapp-demo.war C:\apache-tomcat-10.0.27\webapps\
  ```

- **Expected Output:** Tomcat console shows deployment logs.

Step 10: Verify in Browser

- **Action:** Open:
  - `http://localhost:8080/webapp-demo/`
  - `http://localhost:8080/webapp-demo/hello`
- **Expected Output:** JSP welcome page and servlet response appear.

🌐 How the Web Works (Brief)

- **Internet** → A global network that connects computers so they can share information.
- **DNS (Domain Name System)** → Like a phonebook: it converts website names (e.g., `google.com`) into IP addresses.
- **IP Address** → A unique number that identifies a computer/server on the network.
- **Ports** → Doors on a computer that let specific services run (HTTP uses port 80, HTTPS uses 443).
- **Server** → A computer that stores websites and responds to requests from clients (browsers).

🔄 Client–Server Model

1. **Browser (Client)** → Sends a request using **HTTP**.
2. **Web Server** → Receives the request.
3. **Backend Logic** → Processes data (e.g., Java, Python, Node.js).
4. **Database** → Stores and retrieves information.
5. **Response** → Sent back to the browser, which displays the page.

📌 HTTP Request–Response

- **GET** → Ask for data (view a page).
- **POST** → Send data (submit a form).
- **Status Codes**:
  - **200 OK** → Success
  - **404 Not Found** → Page doesn't exist
  - **500 Internal Server Error** → Something broke on the server

🧪 Lab Activity

1. Open **Chrome DevTools → Network tab**.
2. Reload a website.
3. Observe each request:
   - Method (GET/POST)
   - Status code (200, 404, etc.)
   - Response time (ms)
4. Ask students to **sketch a diagram** showing how a request flows:

   Code

   ```
   Browser → HTTP → Web Server → Backend → Database → Response
   ```

📘 Introduction to Servlets

1. What is a Servlet?

- A **Servlet** is a Java class that runs inside a web server (like Apache Tomcat).
- It handles **HTTP requests** (GET, POST, etc.) and generates **responses** (usually HTML).
- Servlets are the foundation of Java web applications.

2. Deployment via Apache Tomcat

- Tomcat is a **Servlet container**: it manages servlet lifecycle and request routing.
- To deploy:
  1. Package your app as a **WAR file** (`.war`).
  2. Place it inside Tomcat's `webapps` folder.
  3. Tomcat expands the WAR and makes it accessible via `http://localhost:8080/<appname>`.

3. Directory Structure of a Simple Web App

Code

```
webapp-demo/
├── index.jsp
└── WEB-INF/
    ├── classes/      (compiled .class files)
    ├── lib/          (JAR dependencies)
    └── web.xml       (deployment descriptor)
```

- **WEB-INF**: Hidden from direct browser access.
- **web.xml**: Maps URLs to servlet classes.
- **classes**: Contains compiled servlet `.class` files.

4. Servlet Lifecycle

Servlets are managed by Tomcat. Lifecycle methods:

1. **init()**
   - Called once when the servlet is first loaded.
   - Used for initialization (e.g., DB connections).
   - Example: `System.out.println("Servlet initialized");`
2. **service()**
   - Called for every request.
   - Dispatches to `doGet()` or `doPost()` depending on HTTP method.
3. **destroy()**
   - Called once when the servlet is unloaded or server shuts down.
   - Used for cleanup (e.g., closing resources).

- Example: `System.out.println("Servlet destroyed");`

5. Lab Exercise

Servlet Code (HelloServlet.java)

java

```
`import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.annotation.WebServlet;

@WebServlet("/HelloServlet")
public class HelloServlet extends HttpServlet {

    @Override
    public void init() throws ServletException {
        System.out.println("Servlet initialized");
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("Handling GET request");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>Hello from Server</h1>");
        out.println("</body></html>");
    }

    @Override
    public void destroy() {
        System.out.println("Servlet destroyed");
    }
}`
```

Access in Browser

Code

`http://localhost:8080/HelloServlet`

📘 JSP and MVC Integration

1. Problem with Printing HTML in Servlets

- In Servlets, we often write:

  java

  `out.println("<html><body><h1>Hello</h1></body></html>");`

- **Issues**:
  - Hard to read and maintain (mixing Java + HTML).
  - Difficult to design visually — developers must embed markup in Java code.
  - Not designer-friendly: UI/UX teams can't easily edit HTML inside Java.

2. How JSP Solves It

- **JSP (JavaServer Pages)** allows writing HTML directly with embedded Java snippets.
- JSP is compiled into a Servlet by Tomcat, but developers write it like HTML.
- **Benefit**: Clear separation — HTML stays in JSP, logic stays in Servlets.

3. JSP Syntax Basics

- **Direct HTML**: Write normal HTML tags.
- **Expressions**: Output values directly.

  jsp

  `Hello, ${name}`

- **Scriptlets** (Java code inside JSP):

  jsp

  `<% int x = 5; %>`

⚠️ Discourage heavy use — scriptlets mix logic with presentation.

- **Declarations**:

  jsp

  `<%! int counter = 0; %>`

- **Best Practice**: Use EL (`${}`) and JSTL instead of scriptlets.

4. MVC Separation

- **Model** = Data (e.g., Employee object, DAO classes).
- **View** = JSP (presentation layer, displays data).
- **Controller** = Servlet (handles requests, processes data, forwards to JSP).
- **Flow**:
  1. User submits form → Servlet (Controller).
  2. Servlet processes data → sets attributes in request.
  3. Servlet forwards to JSP (View).
  4. JSP displays data using EL.

5. RequestDispatcher and Forwarding

Servlet forwards request to JSP:

java

```
RequestDispatcher rd = request.getRequestDispatcher("confirm.jsp");
rd.forward(request, response);
```

- **forward()** keeps the same request → attributes survive.
- **sendRedirect()** starts a new request → attributes lost.

6. Lab Exercise

Step 1: JSP Form (input.jsp)

jsp

```
<html>
<body>
  <h2>Enter Your Details</h2>
```

```
  <form action="ProcessServlet" method="post">
    Name: <input type="text" name="name"/><br/>
    Email: <input type="text" name="email"/><br/>
    <input type="submit" value="Submit"/>
  </form>
</body>
</html>
```

Step 2: Servlet (ProcessServlet.java)

java

```
@WebServlet("/ProcessServlet")
public class ProcessServlet extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String name = request.getParameter("name");
        String email = request.getParameter("email");

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

        RequestDispatcher rd = request.getRequestDispatcher("confirm.jsp");
        rd.forward(request, response);
    }
}
```

Step 3: Confirmation JSP (confirm.jsp)

jsp

```
<html>
<body>
  <h2>Submission Successful!</h2>
  <p>Name: ${name}</p>
  <p>Email: ${email}</p>
```

```
</body>
</html>
```

📘 Database Connectivity with JSP/Servlet

1. Why Database Connectivity?

- Web apps need to **store and retrieve data** (e.g., employee records).
- JSP/Servlets use **JDBC (Java Database Connectivity)** to interact with relational databases like MySQL.
- JDBC provides a standard API for executing SQL queries from Java.

2. Connection Pooling

- **Problem**: Opening a new DB connection for every request is slow and resource-heavy.
- **Solution**: Tomcat manages a **pool of reusable connections**.
- **How it works**:
  1. At startup, Tomcat creates a pool of connections.
  2. Servlets borrow a connection (`getConnection()`).
  3. After use, the connection is returned to the pool.
- **Benefits**: Faster response, scalable under load, avoids exhausting DB resources.

3. JDBC Setup in Tomcat

Step 1: Add MySQL Connector

Copy `mysql-connector-j.jar` into:

Code

`apache-tomcat-10.0.27/lib`

Step 2: Configure Connection Pool (`context.xml`)

xml

```
<Context>
  <Resource name="jdbc/PayrollDB"
            auth="Container"
            type="javax.sql.DataSource"
            maxTotal="20"
            maxIdle="10"
            maxWaitMillis="10000"
            username="root"
            password="yourpassword"
            driverClassName="com.mysql.cj.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/PayrollDB"/>
</Context>
```

Step 3: Lookup in Servlet

java

```
Context initCtx = new InitialContext();
DataSource ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/PayrollDB");
Connection conn = ds.getConnection();
```

4. CRUD Operations via DAO Pattern

Employee Model

java

```
public class Employee {
    private int id;
    private String name;
    private String email;
    private double salary;
    // getters and setters
}
```

EmployeeDAO (Reusable)

java

```
public class EmployeeDAO {
    private DataSource dataSource;

    public EmployeeDAO(DataSource ds) {
        this.dataSource = ds;
    }

    public void addEmployee(Employee emp) throws SQLException {
        String sql = "INSERT INTO employees (name, email, salary) VALUES
(?, ?, ?)";
        try (Connection conn = dataSource.getConnection();
             PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setString(1, emp.getName());
            ps.setString(2, emp.getEmail());
            ps.setDouble(3, emp.getSalary());
            ps.executeUpdate();
        }
    }

    public List<Employee> getAllEmployees() throws SQLException {
        List<Employee> list = new ArrayList<>();
        String sql = "SELECT * FROM employees";
        try (Connection conn = dataSource.getConnection();
             Statement st = conn.createStatement();
             ResultSet rs = st.executeQuery(sql)) {
            while (rs.next()) {
                Employee emp = new Employee();
                emp.setId(rs.getInt("id"));
                emp.setName(rs.getString("name"));
                emp.setEmail(rs.getString("email"));
                emp.setSalary(rs.getDouble("salary"));
                list.add(emp);
            }
        }
        return list;
```

```
    }
}
```

5. Lab Exercise

Step 1: JSP Form (register.jsp)

jsp

```
<form action="RegisterEmployee" method="post">
  Name: <input type="text" name="name"/><br/>
  Email: <input type="text" name="email"/><br/>
  Salary: <input type="text" name="salary"/><br/>
  <input type="submit" value="Register"/>
</form>
```

Step 2: Servlet (RegisterEmployeeServlet.java)

java

```
@WebServlet("/RegisterEmployee")
public class RegisterEmployeeServlet extends HttpServlet {
    private EmployeeDAO dao;

    @Override
    public void init() throws ServletException {
        try {
            Context initCtx = new InitialContext();
            DataSource ds = (DataSource) initCtx.lookup("java:comp/env/jdbc/P
ayrollDB");
            dao = new EmployeeDAO(ds);
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
response)
```

```
            throws ServletException, IOException {
        Employee emp = new Employee();
        emp.setName(request.getParameter("name"));
        emp.setEmail(request.getParameter("email"));
        emp.setSalary(Double.parseDouble(request.getParameter("salary")));

        try {
            dao.addEmployee(emp);
            response.sendRedirect("listEmployees.jsp");
        } catch (SQLException e) {
            throw new ServletException(e);
        }
    }
}
```

Step 3: JSP Table View (listEmployees.jsp)

jsp

```
<%@ page import="java.util.*, yourpackage.EmployeeDAO, yourpackage.E
mployee" %>
<%
    Context initCtx = new javax.naming.InitialContext();
    javax.sql.DataSource ds = (javax.sql.DataSource) initCtx.lookup("java:co
mp/env/jdbc/PayrollDB");
    EmployeeDAO dao = new EmployeeDAO(ds);
    List<Employee> employees = dao.getAllEmployees();
%>
<table border="1">
  <tr><th>ID</th><th>Name</th><th>Email</th><th>Salary</th></tr>
  <%
    for(Employee emp : employees){
  %>
    <tr>
      <td><%= emp.getId() %></td>
      <td><%= emp.getName() %></td>
      <td><%= emp.getEmail() %></td>
      <td><%= emp.getSalary() %></td>
```

```
    </tr>
  <%
    }
  %>
</table>
```

📄 Sessions, Cookies, and Authentication

Why Sessions Exist

- HTTP is **stateless** → server doesn't remember users between requests.
- **Sessions** solve this by storing user data on the server, linked via a session ID (often passed in a cookie).
- Example: login → server creates session → user stays logged in until logout.

Example 1: Login with HttpSession (Java Servlet)

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse res
p) throws IOException {
        String user = req.getParameter("username");
        String pass = req.getParameter("password");

        // Simple check (in real apps, validate from DB with hashed password)
        if("admin".equals(user) && "secret".equals(pass)) {
            HttpSession session = req.getSession();
            session.setAttribute("username", user);
            resp.sendRedirect("dashboard.jsp");
        } else {
            resp.sendRedirect("login.jsp?error=1");
        }
    }
}
```

Example 2: Access Control with Session

```
@WebServlet("/dashboard")
public class DashboardServlet extends HttpServlet {
   protected void doGet(HttpServletRequest req, HttpServletResponse res
p) throws IOException {
      HttpSession session = req.getSession(false); // don't create new
      if(session != null && session.getAttribute("username") != null) {
         resp.getWriter().println("Welcome " + session.getAttribute("userna
me"));
      } else {
         resp.sendRedirect("login.jsp");
      }
   }
}
```

Example 3: Logout (Invalidate Session)

```
@WebServlet("/logout")
public class LogoutServlet extends HttpServlet {
   protected void doGet(HttpServletRequest req, HttpServletResponse res
p) throws IOException {
      HttpSession session = req.getSession(false);
      if(session != null) {
         session.invalidate(); // remove session
      }
      resp.sendRedirect("login.jsp");
   }
}
```

Example 4: Using Cookies Directly

```
@WebServlet("/setCookie")
public class CookieServlet extends HttpServlet {
   protected void doGet(HttpServletRequest req, HttpServletResponse res
p) throws IOException {
      Cookie c = new Cookie("theme", "dark");
```

```
        c.setMaxAge(60*60); // 1 hour
        resp.addCookie(c);
        resp.getWriter().println("Cookie set!");
    }
}
```

Lab Activity

1. Create a **login page** with username/password.
2. On successful login, store user info in a **session**.
3. Restrict dashboard access → only logged-in users can view.
4. Add **logout** → invalidate session.
5. Experiment with **cookies** (e.g., store theme preference).