Servlet/JSP

Instructor - https://in.linkedin.com/in/suryakantsurve

JEE

- Java Enterprise Edition (Java EE now officially known as Jakarta EE is a mature, multi-tiered architecture designed for building scalable, reliable, and secure enterprise applications.

- The architecture is built on a four-tier model that separates application logic into distinct, isolated layers, allowing for easier maintenance and scalability.

The 4-Tier Java EE Model

- Client Tier: Runs on the client machine (e.g., web browser, mobile app, or desktop application) and handles user interaction.

- Web Tier: Runs on the server and processes requests from the client. It uses Servlets, Jakarta Server Pages (JSP), and Jakarta Faces (JSF) to generate dynamic content.

- Business Tier: Contains the core logic of the application. It typically utilizes Enterprise JavaBeans (EJB) to handle complex tasks like calculations, transactions, and security.

- Enterprise Information System (EIS) Tier: Manages the permanent storage of data, usually in a relational database like MySQL or Oracle, often accessed via the Jakarta Persistence API (JPA).

Core Architectural Components

- Containers: These provide the runtime environment for components, handling low-level details like security, transaction management, and resource pooling. Major types include the Web Container and the EJB Container.

- APIs & Specifications: Java EE defines standard "contracts" (like JPA, JMS, and CDI) that are implemented by various vendors such as GlassFish, Payara, and Red Hat JBoss.

- Jakarta EE 11 Evolution: As of 2026, the architecture has evolved to be cloud-native and microservices-ready. It supports modern features like Virtual Threads (from Java 21) and the new Jakarta Data specification for simplified data access.

Key Difference: Java EE vs. Jakarta EE

- Starting with Jakarta EE 9, the primary change is the namespace migration: all APIs moved from the javax.* namespace to jakarta.*.

Java Servlets

- Java Servlets now part of **Jakarta EE** remain the core technology for handling HTTP requests and generating dynamic web content.

Core Concept & Lifecycle

- A Servlet is a Java class that extends a web server's capabilities.
- It is managed by a Servlet Container (like Apache Tomcat 10+)
- Loading & Instantiation: The container loads the class and creates an instance.
- Initialization (init): Called once to set up resources like database connections.
- Service (service): Called for every request; it dispatches to specific methods like doGet() or doPost().
- Destruction (destroy): Called before the servlet is removed to clean up resources.

Creating a Simple Servlet

```
// Maps this servlet to the URL: http://localhost:8080/your-app/hello
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
                // 1. Set the content type
        response.setContentType("text/html");
        // 2. Get the output writer
        PrintWriter out = response.getWriter();
        // 3. Write the HTML response
        out.println("<html><body>");
        out.println("<h1>Hello from Jakarta EE!</h1>");
        out.println("</body></html>");
    }
}
```

Servlet API Class Hierarchy

- Servlet (Interface): The root of the hierarchy defining lifecycle methods like init(), service(), and destroy().

- GenericServlet (Abstract Class): Implements Servlet and ServletConfig. It provides default behavior for all methods except service(), making it protocol-independent.

- HttpServlet (Abstract Class): Extends GenericServlet specifically for the HTTP protocol. It provides methods like doGet(), doPost(), and doPut().

Request and Response Architecture

- ServletRequest / HttpServletRequest: Provides data from the client, such as parameters, headers, and cookies.

- ServletResponse / HttpServletResponse: Used to send data back to the client, including the output stream and status codes.

Servlet Collaboration (The "Dispatch" Layer)

- RequestDispatcher: An interface used to wrap a server resource (Servlet, JSP, HTML). It allows for:
  - Forward: The request is passed to another resource entirely on the server side.
  - Include: The content of another resource is embedded into the current response.

- sendRedirect: A method in HttpServletResponse that instructs the client's browser to make a completely new request to a different URL.

Servlet Lifecycle

- Loading & Instantiation: The container loads the class and creates an instance.
  - Initialization: The init() method is called once.
  - Request Handling: The service() method handles multiple client requests by dispatching to specific handlers (like doGet).
  - Destruction: The destroy() method is called before removal to clean up resources.

Key Development Steps

- Environment Setup: Install Java JDK 17+ and a Jakarta EE compatible server like Apache Tomcat 10.1+.
- Project Structure: Create a Maven project with the jakarta.servlet-api dependency in your pom.xml.
- Deployment: Package your application as a .war (Web ARchive) file and place it in the server's webapps folder.

Maven pom.xml Setup

```
<dependencies>
    <!-- Core Jakarta Servlet API 6.1 -->
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>6.1.0</version>
        <scope>provided</scope>
    </dependency>
    <!-- Jakarta Server Pages (JSP) API 4.0 -->
    <dependency>
        <groupId>jakarta.servlet.jsp</groupId>
        <artifactId>jakarta.servlet.jsp-api</artifactId>
        <version>4.0.0</version>
        <scope>provided</scope>
    </dependency>
    <!-- JSTL for logic in JSPs (Optional but recommended) -->
    <dependency>
        <groupId>jakarta.servlet.jsp.jstl</groupId>
        <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
        <version>3.0.0</version>
    </dependency>
</dependencies>
```

Servlet-to-JSP Communication

- In a standard Model-View-Controller (MVC) architecture, the Servlet acts as the Controller (processing data) and the JSP as the View (displaying it).

Step A: Create the View (index.jsp)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Welcome Page</title>
</head>
<body>
    <!-- Displaying the dynamic message from the Servlet -->
    <h1>Hello, ${userName}!</h1>
    <p>Current Time: ${serverTime}</p>
</body>
</html>
```

Step B: Create the Controller (UserServlet.java)

```
@WebServlet("/greet")
public class UserServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        // 1. Logic: Prepare data
        String name = request.getParameter("user");
        if (name == null || name.isEmpty()) name = "Guest";

        // 2. Set Attributes for the JSP to read
        request.setAttribute("userName", name);
        request.setAttribute("serverTime", LocalTime.now());

        // 3. Forward the request to the JSP
        request.getRequestDispatcher("/WEB-INF/views/welcome.jsp")
               .forward(request, response);
    }
}
```

Running the App

- Build: Run mvn clean package to generate a .war file.
- Deploy: Place the .war in the webapps folder of Apache Tomcat 11.
- Access: Open http://localhost:8080/your-app/greet?user=John in your browser.

ServletContext

- ServletContext is an interface that acts as the "Big Picture Manager" for an entire web application.

- Provides a bridge between servlets and the web container, allowing them to share global information and interact with the server environment.

Key Characteristics

- Single Instance: Only one ServletContext object is created by the web container per web application per JVM.
- Global Scope: It is shared across all servlets and JSPs within the application.
- Life Cycle: It is created when the web application is deployed and destroyed when the application is removed or the server stops.

Primary Uses

- Shared Configuration: It retrieves application-wide initialization parameters defined in the web.xml file using the <context-param> tag.

- Data Sharing (Attributes): Servlets can set, get, and remove attributes in the context (using setAttribute(), getAttribute(), and removeAttribute()), allowing them to share dynamic data with each other.

- Resource Access: It provides methods to get the real path of files (getRealPath()), access resources as input streams (getResourceAsStream()), or obtain a RequestDispatcher for forwarding requests.

- Logging: It allows servlets to write messages to the server's event log file.

Code Access

- Directly in a Servlet: Using the inherited getServletContext() method.
- Via ServletConfig: Calling getServletConfig().getServletContext().
- From a Request: Calling request.getServletContext().
- In JSPs: Using the implicit application object.

Example Configuration (web.xml)

```
<web-app>
   <!-- Define a global parameter for the whole app -->
   <context-param>
      <param-name>dbUrl</param-name>
      <param-value>jdbc:mysql://localhost:3306/mydb</param-value>
   </context-param>
</web-app>
```

ServletContextListener

```
@WebListener
public class MyAppContextListener implements ServletContextListener {
    // Runs exactly when the application starts (before any servlets are initialized)
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("Application Starting...");
         // Example: Initialize a Database Connection Pool
        // sce.getServletContext().setAttribute("myDbPool", new ConnectionPool());
    }

    // Runs exactly when the application stops (after all servlets are destroyed)
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("Application Stopping...");

        // Example: Clean up resources
        // ConnectionPool pool = (ConnectionPool) sce.getServletContext().getAttribute("myDbPool");
        // pool.close();
    }
}
```

Example Configuration (Annotation)


```
@WebListener
public class MyGlobalConfig implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext context = sce.getServletContext();

        // This is the programmatic equivalent of <context-param>
        context.setInitParameter("globalTheme", "Dark-Mode");
    }
}
```

Common Use Cases

- Database Connectivity: Opening a global connection pool when the server starts and closing it cleanly on shutdown.

- Resource Loading: Loading application-wide configuration files (like .properties or .xml) into the ServletContext.

- Background Tasks: Starting or stopping worker threads (e.g., scheduled jobs or monitoring services).

Important Execution Order

- Startup: All contextInitialized() methods are called before any Filter or Servlet is initialized.

- Shutdown: All Filters and Servlets are destroyed before any contextDestroyed() methods are called.

ServletConfig

- ServletConfig remains a core interface in the Jakarta Servlet API used to pass initialization parameters to a single, specific servlet.

- Unlike ServletContext, which is global, ServletConfig is local and private to the servlet it was created for.

Key Characteristics

- One per Servlet: The web container creates a unique ServletConfig object for every individual servlet instance in a web application.
- Local Scope: Parameters defined in one servlet's ServletConfig are not accessible to any other servlet.
- Initialization Timing: It is created when the servlet is initialized and passed as an argument to the servlet's init(ServletConfig config) method.
- Lifecycle: The object exists only as long as its corresponding servlet is in service; it is destroyed when the servlet is destroyed.

Core Methods

- getInitParameter(String name): Returns the value of a specific initialization parameter as a String.
- getInitParameterNames(): Returns an Enumeration of all parameter names defined for that servlet.
- getServletName(): Returns the name of the servlet instance as defined in the deployment descriptor.
- getServletContext(): Returns a reference to the ServletContext, allowing the servlet to access global application data.

Defining Parameters (web.xml)

```
<servlet>
   <servlet-name>LoginServlet</servlet-name>
   <servlet-class>com.example.LoginServlet</servlet-class>
   <init-param>
      <param-name>maxUploadSize</param-name>
      <param-value>10MB</param-value>
   </init-param>
</servlet>
```

Accessing in Code


```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
    // Obtain the config object
    ServletConfig config = getServletConfig();

    // Retrieve a specific local parameter
    String limit = config.getInitParameter("maxUploadSize");
}
```

Implementation with Annotations

```
@WebServlet(
    urlPatterns = "/config-demo",
    initParams = {
        @WebInitParam(name = "adminEmail", value = "support@example.com"),
        @WebInitParam(name = "maxItems", value = "50")
    }
)
public class ConfigDemoServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        // 2. Obtain the ServletConfig object
        ServletConfig config = getServletConfig();
        // 3. Retrieve values using the parameter names defined above
        String email = config.getInitParameter("adminEmail");
        String items = config.getInitParameter("maxItems");
        // Use the values in your logic
        System.out.println("Admin: " + email + ", Limit: " + items);
    }
}
```

RequestDispatcher

- In Java Servlets, the RequestDispatcher interface is used to dispatch a request to another resource (Servlet, JSP, or HTML) on the same server.

- It is a key tool for "Servlet Collaboration," allowing multiple resources to handle a single request.

Methods of RequestDispatcher

- forward(ServletRequest request, ServletResponse response):
  - Transfers the entire request from the current servlet to another resource.
  - The second resource is responsible for generating the final response to the client.
  - The browser's URL remains unchanged because the transfer happens entirely on the server side.

- include(ServletRequest request, ServletResponse response):
  - Includes the content of another resource (like a common header or footer) into the current servlet's response.
  - Control remains with the original servlet, which sends the final response to the client.

RequestDispatcher Object

request.getRequestDispatcher("path")

The path can be relative to the current servlet.

getServletContext().getRequestDispatcher("/path")

The path must be absolute (starting with /).

Basic Code Example

```
// Setting data to be passed to the next resource
request.setAttribute("user", "JohnDoe");

// Forwarding to a JSP page
RequestDispatcher rd = request.getRequestDispatcher("welcome.jsp");
rd.forward(request, response);
```

sendRedirect

- In Java Servlets, sendRedirect is a method of the HttpServletResponse interface used to redirect a client to a different URL.

- Unlike the RequestDispatcher.forward() method, which happens entirely on the server, sendRedirect is a client-side redirection.

How it Works

- Response to Client: When response.sendRedirect("url") is called, the server sends a response back to the browser with an HTTP status code (typically 302 Found) and a Location header containing the new URL.

- New Request: The browser receives this response and automatically issues a completely new GET request to the specified URL.

- URL Update: Because the browser initiates a new request, the URL in the browser's address bar updates to the new location.

Key Characteristics

- Scope: It can redirect to resources within the same server or to external websites (e.g., response.sendRedirect("https://www.google.com")).

- Object Lifecycle: Since a new request is created, the original request and response objects are lost. Data from the first request cannot be accessed in the second unless it is passed via the URL (query parameters) or stored in a HttpSession.

- Performance: It is generally slower than forward() because it requires an extra network round-trip between the client and the server.

Implementation Example

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
    // Logic here...

    // Redirecting to another page
    response.sendRedirect("welcome.jsp");
}
```

Servlet Filters

- Servlet filters are Java objects used in web applications to intercept and process client requests and server responses before they reach or after they leave their target resource (such as a servlet, JSP, or static file)

- They provide a way to encapsulate common functionalities like logging, authentication, and data compression without modifying the core logic of the target resource.

Key Concepts

- Pluggable components: Filters are configured separately (e.g., in web.xml or using the @WebFilter annotation) and can be easily added or removed without recompiling the servlets they interact with.

- Filter Chain: When multiple filters are mapped to the same request, they are arranged in a specific order (a "chain"). The request passes through each filter in the defined sequence before reaching the target resource, and the response passes back through the filters in reverse order. The chain.doFilter() method is used to pass control to the next entity in the chain.

- Request/Response Wrapping: Filters can wrap the request and response objects to modify their headers or content, allowing for custom processing like data encryption or content transformation.

Common Uses

- Authentication and Authorization
- Logging and Auditing
- Input Validation
- Data Compression/Encryption
- Content Transformation

Lifecycle Methods

- init(FilterConfig config): Called once by the servlet container when the filter is first instantiated and placed into service to perform initial setup and allocate resources.

- doFilter(ServletRequest request, ServletResponse response, FilterChain chain): The core method called for every request/response pair that matches the filter's configuration. It performs the filtering logic and calls chain.doFilter() to continue the chain.

- destroy(): Called once by the web container before the filter is taken out of service to clean up resources (e.g., closing database connections).

```
@WebFilter("/*")
public class ExecutionTimeFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
                // Pre-processing: Record start time
        long startTime = System.currentTimeMillis();
        String ipAddress = request.getRemoteAddr();
                // Cast to HttpServletRequest to access specific HTTP details
        if (request instanceof HttpServletRequest httpReq) {
            System.out.println("Request URI: " + httpReq.getRequestURI());
        }
        // Pass the request along the chain to the next filter or target servlet
        chain.doFilter(request, response);
        // Post-processing: Calculate and log duration
        long duration = System.currentTimeMillis() - startTime;
        System.out.println("Processed request from " + ipAddress + " in " + duration + "ms");
    }
}
```

Alternative: web.xml Configuration

```
<filter>
    <filter-name>ExecutionTimeFilter</filter-name>
    <filter-class>com.example.ExecutionTimeFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>ExecutionTimeFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

ServletListener

- Servlet Listeners continue to be essential components of the Jakarta Servlet API , used to monitor and respond to lifecycle events within a web application.

- Listeners allow developers to perform automated tasks when specific events occur, such as initializing resources (like database connections) when the application starts or cleaning up data when a session expires.

Key Listener Categories

- ServletContextListener
- HttpSessionListener
- ServletRequestListener
- ServletContextAttributeListener

```
@WebListener
public class AppInitListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("Web application starting up...");
        // Initialize global resources here
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("Web application shutting down...");
        // Close resources here
    }
}
```

```
@WebListener
public class UserSessionListener implements HttpSessionListener, HttpSessionAttributeListener {
    private static int activeSessions = 0;
    @Override
    public void sessionCreated(HttpSessionEvent se) {
        activeSessions++;
        System.out.println("New Session Created. Active Users: " + activeSessions);
    }
    @Override
    public void attributeAdded(HttpSessionBindingEvent se) {
        if ("user_id".equals(se.getName())) {
            System.out.println("User Logged In: " + se.getValue());
        }
    }
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        activeSessions--;
        System.out.println("Session Destroyed. Active Users: " + activeSessions);
    }
}
```

```
@WebListener
public class RequestAuditListener implements ServletRequestListener {
    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        HttpServletRequest req = (HttpServletRequest) sre.getServletRequest();
        System.out.println("Incoming Request: " + req.getMethod() + " " + req.getRequestURI());
    }
    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        System.out.println("Request processing completed for one thread.");
    }
}
```

Servlet Sessions

- Servlet Sessions remain the standard way to maintain "state" in Java web applications using the Jakarta Servlet API (specifically version 6.1 and beyond).

- Because the HTTP protocol is stateless, a server cannot naturally recognize that two separate requests come from the same user.

- Sessions bridge this gap by allowing the server to "remember" a user's data (like a shopping cart or login status) across multiple pages.

How Sessions Work

- When a client first interacts with a servlet-based application, the web container (like Apache Tomcat) creates a unique HttpSession object and assigns it a Session ID.

- The Identifier: This ID is sent back to the browser, typically as a cookie named JSESSIONID.

- The Cycle: For every subsequent request, the browser automatically sends this ID back to the server. The server then uses the ID to find and retrieve the specific user's session data from its memory.

Core Interface: HttpSession

- The HttpSession object acts as a server-side "dictionary" where you can store and retrieve Java objects using key-value pairs.

Session Tracking Techniques

- Cookies: The standard method.
- URL Rewriting: If cookies are disabled, the session ID is appended to every URL (e.g., .../page;jsessionid=12345). Developers must use response.encodeURL() for this to work.
- Hidden Form Fields: Storing the ID in an invisible input field in HTML forms.

Lifecycle & Configuration

- Timeout: Sessions expire automatically after a period of inactivity (default is usually 30 minutes) to free up server resources.

- You can change this using session.setMaxInactiveInterval(seconds) or in the web.xml configuration.

- Scope: Session data is private to an individual user and is scoped to a single web application. It is not shared between different users or different apps on the same server.

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
   protected void doPost(HttpServletRequest request, HttpServletResponse response)
         throws IOException {
            String user = request.getParameter("username");
            // request.getSession() retrieves existing or creates a new session
      HttpSession session = request.getSession();
            // Store user info in the session "basket"
      session.setAttribute("user_name", user);
            // Optional: Set session timeout to 15 minutes (900 seconds)
      session.setMaxInactiveInterval(900);
            response.sendRedirect("welcome");
   }
}
```

```
@WebServlet("/welcome")
public class WelcomeServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
                // getSession(false) returns null if no session exists
        HttpSession session = request.getSession(false);
                if (session != null && session.getAttribute("user_name") != null) {
            String user = (String) session.getAttribute("user_name");
            response.getWriter().println("Welcome back, " + user + "!");
        } else {
            response.sendRedirect("login.html");
        }
    }
}
```

```
WebServlet("/logout")
public class LogoutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {

        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate(); // Destroys the session immediately
        }
        response.sendRedirect("login.html");
    }
}
```

JSP

- In 2016 Jakarta Server Pages (formerly JavaServer Pages or JSP) continues as a core server-side template engine within the Jakarta EE 11 ecosystem.

- While modern web development has largely shifted toward client-side frameworks (like React or Angular), JSP remains a critical technology for enterprise systems and legacy application maintenance.

Core Concept

- JSP allows developers to create dynamic web content by embedding Java code directly within HTML pages.

- Extension of Servlets: Internally, every JSP page is translated into a Java Servlet by the web container (like Apache Tomcat) the first time it is accessed.

- Separation of Concerns: It is designed to act as the View in Model-View-Controller (MVC) architectures, where servlets handle business logic and JSP renders the user interface.

The JSP Lifecycle

- Translation: The .jsp file is converted into a .java servlet source file.
- Compilation: The source file is compiled into a .class bytecode file.
- Initialization: The container calls jspInit() to set up resources.
- Execution: For every request, the _jspService() method is executed to generate the HTML output.
- Cleanup: The jspDestroy() method is called before the page is removed from memory.

JSP Scriptlets

- In 2026, JSP Scriptlets refer to blocks of raw Java code embedded directly within a Jakarta Server Page using <% ... %> tags.

- While they remain a core part of the Jakarta Pages 4.0 specification, their use is heavily discouraged in modern enterprise development in favor of cleaner alternatives like JSTL and Expression Language (EL)

How Scriptlets Work

- Syntax: Code is wrapped in <% java source code %>.
- Execution: During the translation phase, the JSP engine copies the code inside a scriptlet directly into the _jspService() method of the generated servlet.
- Variable Scope: Variables declared in a scriptlet are local to that specific service method but are accessible from anywhere within the same JSP page.
- Implicit Objects: Scriptlets have direct access to implicit objects like request, response, session, and out.

Example

```
<%
  String name = request.getParameter("user");
  if (name != null) {
      out.println("Hello, " + name);
  } else {
      out.println("Hello, Guest!");
  }
%>
```

Implicit Objects

- request – HttpServletRequest
- response – HttpServletResponse
- out – JspWriter
- session – HttpSession
- application – ServletContext
- confg – ServletConfig
- pageContext – PageContext
- page – Object (this)
- exception - Throwable

JSP Action Tags

- JSP Action Tags (part of Jakarta Pages 4.0) are XML-style constructs used to control the behavior of the servlet engine at runtime.

- Unlike directives, which are processed during the translation phase, action tags are re-evaluated every time a page is requested.

Action Tags

- jsp:include: Dynamically includes the response of another resource (JSP, HTML, or Servlet) at the time of the request.
- jsp:forward: Terminates the current page and forwards the request to another internal resource. The client's URL does not change.
- jsp:useBean: Locates or instantiates a JavaBean. It uses the id to reference the object and scope to define its lifespan (Page, Request, Session, or Application).
- jsp:setProperty: Sets properties of a JavaBean using setter methods. Using property="*" allows automatic mapping of all request parameters to matching bean properties.
- jsp:getProperty: Retrieves a bean property value, converts it to a string, and inserts it directly into the output.
- jsp:param: Used as a child tag within jsp:include or jsp:forward to pass specific key-value pairs to the target resource.

```
<%-- Instantiate or find the User bean --%>

<jsp:useBean id="user" class="com.example.User" scope="request" />

<%-- Automatically map all request parameters to bean properties --%>

<jsp:setProperty name="user" property="*" />

<%-- Output a specific property --%>

<p>Welcome, <jsp:getProperty name="user" property="firstName" />!</p>
```

JSP Directives

- JSP Directives are essential instructions used in Jakarta Server Pages 4.0 to guide the web container on how to translate and compile a JSP page into its corresponding servlet.

- Unlike action tags, directives are processed primarily during the translation phase and do not produce any direct HTML output for the browser.

  - Page Directive (<%@ page ... %>)
  - Include Directive (<%@ include ... %>)
  - Taglib Directive (<%@ taglib ... %>)

Custom Tags

- Custom Tags are the professional standard for extending Jakarta Server Pages (JSP) 4.0.

- They allow developers to create user-defined, reusable HTML-like tags that encapsulate complex Java logic, effectively removing all raw Java code (scriptlets) from the view layer.

Why Use Custom Tags?

- Separation of Concerns: Java developers write the logic in backend classes; web designers use simple tags in the frontend.
- Reusability: A single tag (e.g., <my:chart>) can be used across hundreds of pages.
- Readability: It makes JSP files look like clean HTML/XML instead of messy Java code

Three Core Components

- The Tag Handler Class
- The Tag Library Descriptor (TLD)
- The JSP Implementation

Steps

- Compile the Java class and ensure it is in the application's classpath (e.g., WEB-INF/classes).
- Define the tag in a .tld file inside WEB-INF.
- Include the library in your JSP using the @taglib directive and the designated prefix.

The Tag Handler Class

```
public class HelloTag extends SimpleTagSupport {
    private String name;
    // Setter for the tag attribute 'name'
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public void doTag() throws IOException {
        JspWriter out = getJspContext().getOut();
        out.print("Hello, " + (name != null ? name : "Guest") + "!");
    }
}
```

Tag Library Descriptor (TLD)

```
<taglib xmlns="jakarta.ee" version="3.0">
   <tlib-version>1.0</tlib-version>
   <short-name>ExampleTags</short-name>
   <tag>
      <name>greet</name>
      <tag-class>com.example.tags.HelloTag</tag-class>
      <body-content>empty</body-content>
      <attribute>
         <name>name</name>
         <required>false</required>
         <rtexprvalue>true</rtexprvalue>
      </attribute>
   </tag>
</taglib>
```

JSP Page

```
<%@ taglib prefix="ex" uri="/WEB-INF/custom.tld" %>
<html>
<body>
   <h1>Custom Tag Demo</h1>
      <!-- Using the tag with an attribute -->
   <ex:greet name="Sachin" />
      <!-- Using the tag without an attribute (uses default "Guest") -->
   <ex:greet />
</body>
</html>
```

Example: Capitalizing Body Content

- This example shows how to capture the body content, modify it (convert to uppercase), and then write it back to the JSP page.

Tag Handler Class

```
public class CapitalizeTag extends SimpleTagSupport {
   @Override
   public void doTag() throws JspException, IOException {
      // 1. Get the body content fragment
      JspFragment body = getJspBody();
            if (body != null) {
         // 2. Capture the body into a StringWriter
         StringWriter sw = new StringWriter();
         body.invoke(sw);
                  // 3. Manipulate the content
         String content = sw.toString().toUpperCase();
                  // 4. Write it to the page
         getJspContext().getOut().print(content);
      }
   }
}
```

TLD File

```
<tag>
   <name>capitalize</name>
   <tag-class>com.example.CapitalizeTag</tag-class>
   <body-content>scriptless</body-content>
</tag>
```

JSP

```
<%@ taglib uri="/WEB-INF/custom.tld" prefix="my" %>
<my:capitalize>
   this text will be capitalized.
</my:capitalize>
```

Tag Files

- Tag Files (.tag) provide a simplified way to create custom tags using standard JSP syntax instead of Java classes.

- They are ideal for presentation-centric tasks and allow developers who may not know Java to build reusable UI components.

Example File: /WEB-INF/tags/welcome.tag

```
<%-- Declare attributes using the directive --%>
<%@ attribute name="user" required="true" %>
<%@ attribute name="color" required="false" %>

<%-- Default color if not provided --%>
<div style="color: ${empty color ? 'black' : color}">
    Welcome to the site, ${user}!
</div>
```

Use the Tag in a JSP

```
<%@ taglib prefix="my" tagdir="/WEB-INF/tags" %>
<html>
<body>
   <%-- Call the tag with its filename as the tag name --%>
   <my:welcome user="John Doe" color="blue" />
</body>
</html>
```

JSTL

- Jakarta Standard Tag Library, formerly JavaServer Pages Standard Tag Library is a collection of standardized custom tags used to manage common web development tasks in JSP pages.

- It eliminates the need for Java scriptlets (<% ... %>), leading to cleaner, more maintainable code.

- As of 2026, the latest major version is JSTL 3.0 (part of Jakarta EE), which uses the jakarta.servlet.jsp.jstl namespace.

Key Tag Categories

- Core Tags (c): Used for fundamental tasks like conditional logic, loops, and variable management.
  - <c:if>: Simple conditional checks.
  - <c:forEach>: Iterates over collections or arrays.
  - <c:set> and <c:remove>: Manages scoped variables.

- Formatting Tags (fmt): Supports internationalization (i18n) by formatting numbers, dates, and localized messages based on the user's locale.
  - <fmt:formatDate> and <fmt:formatNumber>

- Function Tags (fn): Provides standard string manipulation and collection length utilities.
  - fn:length(), fn:contains(), fn:toLowerCase().

- XML Tags (x): Used for parsing and transforming XML documents using XPath expressions.

- SQL Tags (sql): Allows executing database queries directly from the JSP (primarily used for rapid prototyping).

Core Tags (c)

```
<%@ taglib uri="jakarta.tags.core" prefix="c" %>
<%-- 1. Setting and displaying a variable --%>
<c:set var="userRole" value="admin" />
<p>User Role: <c:out value="${userRole}" /></p>

<%-- 2. Conditional Logic --%>
<c:choose>
    <c:when test="${userRole == 'admin'}">
        <strong>Welcome, Administrator!</strong>
    </c:when>
    <c:otherwise>
        Welcome, User.
    </c:otherwise>
</c:choose>

<%-- 3. Looping through a collection --%>
<ul>
    <c:forEach var="item" items="${productList}">
        <li>${item.name} - $${item.price}</li>
    </c:forEach>
</ul>
```

Formatting Tags (fmt)

```
<%@ taglib prefix="fmt" uri="jakarta.tags.fmt" %>

<%-- 1. Formatting a Date --%>
<c:set var="now" value="<%= new java.util.Date() %>" />
<p>Formatted Date: <fmt:formatDate value="${now}" pattern="yyyy-MM-dd HH:mm:ss" /></p>

<%-- 2. Formatting Currency --%>
<p>Total: <fmt:formatNumber value="1250.5" type="currency" currencySymbol="$" /></p>
```

Function Tags (fn)

```
<%@ taglib prefix="fn" uri="jakarta.tags.functions" %>

<c:set var="message" value="Hello Jakarta EE 2026!" />

<%-- 1. Get length of string --%>
Length: ${fn:length(message)}

<%-- 2. Check if string contains a word --%>
<c:if test="${fn:contains(message, 'Jakarta')}">
    <p>Message mentions Jakarta.</p>
</c:if>

<%-- 3. Convert to Uppercase --%>
Upper: ${fn:toUpperCase(message)}
```

Asynchronous web applications

- Asynchronous processing in Jakarta Servlet 6.1+ remains a critical feature for building scalable web applications.

- It allows the server to handle long-running tasks such as waiting for external data or processing a complex calculation without blocking the initial request-handling thread.

Core Concept: AsyncContext

- Traditional servlets use a "thread-per-request" model where a thread is tied up until the response is completed.
- In an asynchronous setup, the servlet signals the container to free up the request-handling thread while the connection remains open for a different thread to finish the work.
- request.startAsync(): Signals that processing should continue asynchronously.
- AsyncContext: An object that provides access to the request and response after the initial servlet thread has exited.
- asyncContext.complete(): Notifies the container that the asynchronous response is finished.

```
@WebServlet(urlPatterns = "/asyncProcess", asyncSupported = true)
public class ModernAsyncServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
                // Start async mode
        final
                // Pass task to a background thread (preferably from a managed thread pool)
        asyncContext.start(() -> {
            try {
                // Simulate long-running process
                Thread.sleep(3000);
                    ServletResponse resp = asyncContext.getResponse();
                resp.getWriter().println("Asynchronous task completed!");
                            // Finalize the response
                asyncContext.complete();
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
}
```

Example: Server-Sent Events (SSE) with Asynchronous Servlet

- This example uses an asynchronous servlet to push real-time updates to a JSP page without closing the connection.

```
@WebServlet(urlPatterns = "/sse-updates", asyncSupported = true)
public class ModernSSEServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
                // Essential headers for SSE
        response.setContentType("text/event-stream");
        response.setCharacterEncoding("UTF-8");
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Connection", "keep-alive");
        final AsyncContext asyncContext = request.startAsync();
                // Use a background thread to push data
        asyncContext.start(() -> {
            try {
                PrintWriter writer = asyncContext.getResponse().getWriter();
                for (int i = 1; i <= 5; i++) {
                    // SSE format: "data: <message>\n\n"
                    writer.write("data: Update #" + i + " at 2026-time\n\n");
                    writer.flush();
                    Thread.sleep(2000); // Wait 2 seconds between updates
                }
                asyncContext.complete(); // End stream
            } catch (Exception e) {
                asyncContext.complete();
            }
        });
    }
}
```

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>2026 Real-Time Updates</title>
</head>
<body>
    <h2>Live Server Updates</h2>
    <div id="updates">Waiting for data...</div>
    <script>
        // Connect to the SSE servlet
        const eventSource = new EventSource("sse-updates");
        eventSource.onmessage = function(event) {
            const newElement = document.createElement("p");
            newElement.textContent = "Received: " + event.data;
            document.getElementById("updates").appendChild(newElement);
        };
        eventSource.onerror = function() {
            console.log("Connection closed or failed.");
            eventSource.close();
        };
    </script>
</body>
</html>
```

Web Application Security

- Web application security in Jakarta Servlet is built on a "Defense in Depth" strategy.

- It leverages both container-managed security and modern programmatic APIs to protect against unauthorized access and data breaches.

- Security is primarily categorized into **Declarative** and **Programmatic** approaches.

Declarative Security (Configuration-Based)

- Uses the web.xml deployment descriptor or annotations to define constraints.
- Security Constraints (<security-constraint>): Defines which URL patterns are protected and which roles can access them.
- Authentication Mechanisms (<login-config>)
  - BASIC: Browser-native dialog for username and password.
  - FORM: Custom HTML login page with action j_security_check.
  - DIGEST: Similar to Basic but transmits passwords in a hashed format.
  - CLIENT-CERT: Highly secure, using SSL/TLS client certificates.
- Secure Transport: Using <user-data-constraint> with the value CONFIDENTIAL ensures the resource is only accessible over HTTPS.

Programmatic Security (Code-Based)

- Used when business logic requires dynamic security decisions (e.g., "only allow access during business hours").

- Key methods in HttpServletRequest include:
  - isUserInRole(String role): Checks if the authenticated user has a specific role.
  - getRemoteUser(): Returns the login name of the authenticated user.
  - authenticate(): Triggers the container's authentication process manually from code.
  - login(user, pass) and logout(): Explicitly manage the user session.

The Custom Login Page (login.jsp)

```
<!-- login.jsp -->
<form method="POST" action="j_security_check">
    <label>Username:</label>
    <input type="text" name="j_username" required>

    <label>Password:</label>
    <input type="password" name="j_password" required>

    <button type="submit">Login</button>
</form>
```

The Configuration (web.xml)

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>Protected Area</web-resource-name>
      <url-pattern>/admin/*</url-pattern> <!-- URLs requiring login -->
   </web-resource-collection>
   <auth-constraint>
      <role-name>ADMIN_ROLE</role-name> <!-- Role allowed to enter -->
   </auth-constraint>
</security-constraint>
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/error.jsp</form-error-page> <!-- Page shown on failure -->
   </form-login-config>
</login-config>
<security-role>
   <role-name>ADMIN_ROLE</role-name>
</security-role>
```

How it Works (Flow)

- Access Request: A user tries to visit /admin/dashboard.jsp.
- Redirection: The container detects the security constraint and automatically displays login.jsp instead of the requested page.
- Intercept: When the user submits the form to j_security_check, the container intercepts the data and checks it against its configured Realm (e.g., a database or LDAP).
- Success: If valid, the container redirects the user back to the originally requested /admin/dashboard.jsp.
- Failure: If invalid, the container forwards the user to error.jsp.

Database Table Schema

```
CREATE TABLE users (
  user_name VARCHAR(50) NOT NULL PRIMARY KEY,
  user_pass VARCHAR(100) NOT NULL -- Use at least 64 chars if hashing
);

-- Table mapping users to roles
CREATE TABLE user_roles (
  user_name VARCHAR(50) NOT NULL,
  role_name VARCHAR(50) NOT NULL,
  PRIMARY KEY (user_name, role_name),
  FOREIGN KEY (user_name) REFERENCES users(user_name)
);
```

Define the DataSource (context.xml)

```
<Context>
  <Resource name="jdbc/AuthDB"
            auth="Container"
            type="javax.sql.DataSource"
            driverClassName="com.mysql.cj.jdbc.Driver"
            url="jdbc:mysql://localhost:3306/your_database"
            username="db_user"
            password="db_password"
            maxTotal="20"
            maxIdle="10"
            maxWaitMillis="-1"/>
</Context>
```

Configure the Realm (context.xml)

```
<Context>
  <!-- ... Resource definition from step 2 above ... -->
  <Realm className="org.apache.catalina.realm.DataSourceRealm"
         dataSourceName="jdbc/AuthDB"
         localDataSource="true"
         userTable="users"
         userNameCol="user_name"
         userCredCol="user_pass"
         userRoleTable="user_roles"
         roleNameCol="role_name">

    <!-- Optional: Use a CredentialHandler for hashed passwords (SHA-256) -->
    <CredentialHandler
        className="org.apache.catalina.realm.MessageDigestCredentialHandler"
        algorithm="SHA-256" />
  </Realm>
</Context>
```

Link to web.xml

```
<security-role>
  <role-name>ADMIN_ROLE</role-name>
</security-role>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Admin Pages</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>ADMIN_ROLE</role-name>
  </auth-constraint>
</security-constraint>
```

Key Implementation Details for 2026

- Modern Tomcat Versions: In Tomcat 10 and 11, ensure you use the localDataSource="true" attribute if the DataSource is defined within the application's context.xml rather than globally in server.xml.

- Password Security: Storing passwords in cleartext is highly discouraged. Always use a CredentialHandler (as shown in Step 3) to verify salted and hashed passwords.

Thank You