**Assignment 5: Enhancing the Understanding of Java Servlets**

**1. Understanding CGI vs Servlets (Theory & Practical Implementation)**

**What is CGI (Common Gateway Interface)?**

CGI is a standard for running **server-side scripts** (e.g., Python, Perl) to generate dynamic web pages.

**Disadvantages of CGI:**

Creates a new process for each request → **High CPU & memory usage**.  
 Slower compared to Servlets.  
 Poor scalability for large applications.

**What are Servlets?**

Servlets are **Java-based web components** that run on a server and generate dynamic content.

**Advantages of Servlets Over CGI:**

**Efficient** – No need to create a new process for each request.  
 **Platform-independent** – Runs on any Java-supported server.  
 **Secure & Scalable** – Uses Java’s built-in security and threading model.

**Practical Implementation: CGI vs Servlet**

**Step 1: Implement a CGI Script (Python) to Handle Form Submission**

python

#!/usr/bin/python3

import cgi

print("Content-Type: text/html\n")

form = cgi.FieldStorage()

name = form.getvalue("name")

print(f"<html><body><h2>Hello, {name}!</h2></body></html>")

This script reads form data and outputs an HTML response.

**Step 2: Implement the Same Functionality Using Java Servlets**

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

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

String name = request.getParameter("name");

response.setContentType("text/html");

response.getWriter().println("<html><body><h2>Hello, " + name + "!</h2></body></html>");

}

}

Uses HttpServlet to handle requests efficiently.  
 Faster than CGI because it does **not create a new process** for each request.

**2. Servlet API & Basic Servlet Implementation**

**Understanding Servlet API**

The **Servlet API** provides classes and interfaces to handle HTTP requests and responses in Java web applications. The key components are:

1. **HttpServlet** – A base class for servlets handling HTTP requests.
2. **HttpServletRequest** – Represents the incoming request (retrieves query parameters, headers, cookies, etc.).
3. **HttpServletResponse** – Represents the outgoing response (sends HTML, JSON, etc.).

**Step 1: Creating a Basic Java Servlet**

A **Servlet** is a Java class that extends HttpServlet. It overrides doGet() or doPost() to handle client requests.

**Basic Servlet to Handle a GET Request**

java

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/welcome") // URL mapping

public class WelcomeServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.setContentType("text/html");

response.getWriter().println("<h2>Welcome to Java Servlets!</h2>");

}

}

The @WebServlet("/welcome") annotation maps the servlet to /welcome.  
 The doGet() method sends an **HTML response** to the client.

**Step 2: Handling GET & POST Requests**

* **GET Request** – Used to retrieve data from the server.
* **POST Request** – Used to send data to the server (e.g., form submission).

**Servlet Handling GET & POST Requests**

java

@WebServlet("/user")

public class UserServlet extends HttpServlet {

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

response.getWriter().println("This is a GET request.");

}

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

String username = request.getParameter("username");

response.getWriter().println("Hello, " + username + "!");

}

}

The doGet() method handles **URL requests**.  
 The doPost() method processes **form submissions**.

**Step 3: Using HttpServletRequest & HttpServletResponse**

**Retrieve Query Parameters from URL**

java

@WebServlet("/greet")

public class GreetServlet extends HttpServlet {

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

String name = request.getParameter("name"); // Extracts ?name=John

response.getWriter().println("Hello, " + name + "!");

}

}

Example URL: http://localhost:8080/greet?name=John  
 Output: Hello, John!

**Displaying Client Request Details**

The **HttpServletRequest** object provides details like:  
 **Client IP Address**  
 **User-Agent (browser info)**  
 **Request Method (GET/POST)**

java

@WebServlet("/request-info")

public class RequestInfoServlet extends HttpServlet {

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

String clientIP = request.getRemoteAddr();

String userAgent = request.getHeader("User-Agent");

response.getWriter().println("Client IP: " + clientIP);

response.getWriter().println("User-Agent: " + userAgent);

}

}

Helps in logging and analytics.

**3. ServletConfig & ServletContext**

**Step 1: Understanding ServletConfig**

ServletConfig is used to pass **initialization parameters** to a specific servlet via web.xml.

**How ServletConfig Works:**

* Stores **configuration settings** for a single servlet.
* Accessed using getServletConfig().getInitParameter("paramName").
* Defined in web.xml.

**Example: Using ServletConfig to Read Configuration Parameters**

**Step 1: Define Parameters in web.xml**

xml

<servlet>

<servlet-name>ConfigServlet</servlet-name>

<servlet-class>com.example.ConfigServlet</servlet-class>

<init-param>

<param-name>dbURL</param-name>

<param-value>jdbc:mysql://localhost:3306/mydb</param-value>

</init-param>

</servlet>

**Step 2: Retrieve Parameters in the Servlet**

java

import java.io.IOException;

import javax.servlet.ServletConfig;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/config")

public class ConfigServlet extends HttpServlet {

private String dbURL;

public void init(ServletConfig config) throws ServletException {

this.dbURL = config.getInitParameter("dbURL"); // Fetch parameter from web.xml

}

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

response.getWriter().println("Database URL: " + dbURL);

}

}

**Stores database URL inside ServletConfig.**  
 **Only accessible within ConfigServlet.**

**Step 2: Understanding ServletContext**

ServletContext is used to **share data across the entire application**.

**Key Features of ServletContext:**

Stores **global parameters** that multiple servlets can access.  
 Used for **application-wide** settings (e.g., app name, version).  
 Accessed using getServletContext().getInitParameter("paramName").

**Example: Using ServletContext to Share Global Data**

**Step 1: Define Global Parameters in web.xml**

xml

<context-param>

<param-name>appName</param-name>

<param-value>My Web App</param-value>

</context-param>

**Step 2: Access the Context Parameter in a Servlet**

java

@WebServlet("/context")

public class ContextServlet extends HttpServlet {

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

String appName = getServletContext().getInitParameter("appName");

response.getWriter().println("Application Name: " + appName);

}

}

**Any servlet can access appName using ServletContext.**

**Step 3: Sharing Application-Wide Data Using ServletContext**

ServletContext can store and retrieve data dynamically using **setAttribute()** and **getAttribute()**.

**Example: Storing & Retrieving Data**

java

@WebServlet("/setAppData")

public class SetAppDataServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) {

getServletContext().setAttribute("activeUsers", 100);

response.getWriter().println("Active users set to 100.");

}

}

@WebServlet("/getAppData")

public class GetAppDataServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response) {

Integer activeUsers = (Integer) getServletContext().getAttribute("activeUsers");

response.getWriter().println("Current Active Users: " + activeUsers);

}

}

setAppData servlet stores data, which getAppData servlet retrieves.

**4. Session Management in Servlets**

**1. Session Tracking with Cookies**

**Cookies** are small pieces of data stored on the client’s browser. They can be used to store user-specific information, such as a username, and pass it between requests.

**Example: Storing and Retrieving User Data Using Cookies**

**Step 1: Storing Data in a Cookie**

java

@WebServlet("/setCookie")

public class SetCookieServlet extends HttpServlet {

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

String username = request.getParameter("username");

// Create a cookie with the username

Cookie cookie = new Cookie("username", username);

cookie.setMaxAge(60 \* 60 \* 24); // 1 day expiry

response.addCookie(cookie);

response.getWriter().println("Cookie Set: Username = " + username);

}

}

**Step 2: Retrieving Data from the Cookie**

java

@WebServlet("/getCookie")

public class GetCookieServlet extends HttpServlet {

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

Cookie[] cookies = request.getCookies();

if (cookies != null) {

for (Cookie cookie : cookies) {

if ("username".equals(cookie.getName())) {

response.getWriter().println("Username from Cookie: " + cookie.getValue());

return;

}

}

}

response.getWriter().println("No username cookie found.");

}

}

**Set the cookie** with the username when the user submits the form.  
 **Retrieve the cookie** on another page to personalize the content.

**2. Session Tracking with URL Rewriting**

When cookies are disabled, we can track sessions by appending a **session ID** to the URL. This method is less secure than cookies, as the session ID can be seen in the browser's address bar.

**Example: URL Rewriting to Pass Session ID**

java

@WebServlet("/sessionExample")

public class SessionServlet extends HttpServlet {

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

String sessionID = request.getSession().getId();

// Append the session ID to a URL for further requests

String sessionURL = response.encodeURL("nextPage.jsp?sessionID=" + sessionID);

response.getWriter().println("Session ID: " + sessionID);

response.getWriter().println("<a href='" + sessionURL + "'>Go to Next Page</a>");

}

}

**Use encodeURL() to append the session ID** to the URL.  
 **Pass the session ID** to other pages for tracking.

**3. Session Tracking with HttpSession API**

HttpSession is the most common and reliable method for managing user sessions. It stores session data on the server-side and uses a session ID to link the client to the data.

**Creating and Managing Sessions with HttpSession**

**Step 1: Creating a Session and Storing Data**

java

@WebServlet("/startSession")

public class StartSessionServlet extends HttpServlet {

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

HttpSession session = request.getSession();

session.setAttribute("username", request.getParameter("username")); // Storing data

response.getWriter().println("Session started. Username stored.");

}

}

**Step 2: Retrieving Session Data**

java

@WebServlet("/getSession")

public class GetSessionServlet extends HttpServlet {

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

HttpSession session = request.getSession(false); // Check if session exists

if (session != null) {

String username = (String) session.getAttribute("username");

response.getWriter().println("Username from session: " + username);

} else {

response.getWriter().println("No active session.");

}

}

}

**Use getSession()** to create or retrieve a session.  
 **Store and retrieve session data** using setAttribute() and getAttribute().

**5. RequestDispatcher & Servlet Communication**

Servlets often need to work together to handle complex requests. **RequestDispatcher** helps in forwarding requests from one servlet to another or to a **JSP page**. It also allows a servlet to include the output of another resource (another servlet or JSP) in the response. This is essential for building **modular applications** and improving the maintainability of your code.

**RequestDispatcher Overview**

* **Forwarding a Request**: One servlet can forward a request to another servlet or JSP, allowing the second resource to handle the request.
* **Including Content**: A servlet can include the output of another servlet or JSP into its own response.

**1. Forwarding a Request Using RequestDispatcher**

**Scenario: Login System with Forwarding**

In this example, a **LoginServlet** will validate the user credentials. If they are correct, the request will be forwarded to a **WelcomeServlet**. If the credentials are incorrect, the request will be forwarded back to the login page with an error message.

**Step 1: LoginServlet to Validate User Credentials**

java

@WebServlet("/login")

public class LoginServlet extends HttpServlet {

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

String username = request.getParameter("username");

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

// Dummy validation logic (In real life, you would query a database)

if ("admin".equals(username) && "password123".equals(password)) {

// Credentials are valid, forward the request to WelcomeServlet

RequestDispatcher dispatcher = request.getRequestDispatcher("/welcome");

dispatcher.forward(request, response);

} else {

// Invalid credentials, forward the request to login with an error message

request.setAttribute("errorMessage", "Invalid username or password");

RequestDispatcher dispatcher = request.getRequestDispatcher("/login.jsp");

dispatcher.forward(request, response);

}

}

}

* **RequestDispatcher.forward(request, response)** forwards the request to another servlet or JSP for further processing.
* If the login is successful, the request is forwarded to the **WelcomeServlet**.
* If the login fails, the request is forwarded back to the login page with an **error message**.

**Step 2: WelcomeServlet to Display a Welcome Message**

java

@WebServlet("/welcome")

public class WelcomeServlet extends HttpServlet {

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

// This will only be reached if login was successful

response.getWriter().println("Welcome to the system!");

}

}

* The **WelcomeServlet** is invoked when the credentials are valid, and it displays a **welcome message**.

**2. Including Content with RequestDispatcher**

RequestDispatcher can also be used to include the output of another servlet or JSP in the response. This is useful for displaying shared content (like headers or footers) in multiple servlets or JSPs.

**Example: Including a Header in Multiple Pages**

**Step 1: Including a Header Servlet in the Main Servlet**

java

@WebServlet("/home")

public class HomeServlet extends HttpServlet {

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

// Include header content

RequestDispatcher dispatcher = request.getRequestDispatcher("/header.jsp");

dispatcher.include(request, response);

// Home page content

response.getWriter().println("<h2>Welcome to the Home Page!</h2>");

}

}

**Step 2: The header.jsp File**

jsp

<!-- header.jsp -->

<html>

<head><title>My Web Application</title></head>

<body>

<h1>Welcome to My Web App</h1>

</body>

</html>

* The **RequestDispatcher.include(request, response)** includes the content of header.jsp in the response.
* This way, you can reuse the header content across different pages of the application.

**6. Implementing Servlet Filters**

Servlet Filters are used to perform preprocessing or postprocessing on requests and responses in a web application. Filters are powerful tools that can handle tasks like **logging**, **authentication**, **input validation**, and **response manipulation**. Filters are defined in the web.xml file or annotated using @WebFilter.

**What are Servlet Filters?**

Filters act on the request before it reaches the target servlet and can modify the request or response. They can also perform postprocessing on the response after the target servlet has processed it. Filters are configured in the web.xml or through annotations.

**Key Points about Filters:**

* **Preprocessing**: Filters are used to intercept and modify requests before they reach a servlet.
* **Postprocessing**: Filters can also modify responses after the servlet has processed them.
* **Chaining**: Multiple filters can be used together in a chain to perform various tasks.

**1. Logging Filter**

A **logging filter** logs each incoming request to the server, such as the IP address, time of request, and the URL requested. This can be useful for debugging, monitoring, or auditing requests to your application.

**Example: Logging Filter**

java

@WebFilter("/\*") // This will intercept all requests to the web application

public class LoggingFilter implements Filter {

public void init(FilterConfig filterConfig) throws ServletException {

// Initialize filter if needed (optional)

}

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

// Log request details

String ipAddress = request.getRemoteAddr();

String requestedURL = ((HttpServletRequest) request).getRequestURL().toString();

long timestamp = System.currentTimeMillis();

System.out.println("Request from IP: " + ipAddress + " at " + timestamp + " for URL: " + requestedURL);

// Pass the request and response to the next filter or servlet in the chain

chain.doFilter(request, response);

}

public void destroy() {

// Clean up resources if necessary (optional)

}

}

* The LoggingFilter intercepts every request to the web application (because of the /\* mapping).
* It logs the **IP address**, **timestamp**, and **requested URL** before passing the request to the next filter or servlet.
* chain.doFilter(request, response) ensures that the request continues to the next filter or servlet.

**2. Authentication Filter**

An **authentication filter** is often used to check whether the user is logged in before accessing certain protected resources. If the user is not authenticated, the filter can redirect them to a login page or return an error.

**Example: Authentication Filter**

java

@WebFilter("/protected/\*") // This will intercept all requests to URLs starting with /protected

public class AuthenticationFilter implements Filter {

public void init(FilterConfig filterConfig) throws ServletException {

// Optional initialization logic

}

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

throws IOException, ServletException {

HttpServletRequest httpRequest = (HttpServletRequest) request;

HttpServletResponse httpResponse = (HttpServletResponse) response;

// Check if the user is logged in (e.g., session attribute "loggedIn")

HttpSession session = httpRequest.getSession(false);

if (session == null || session.getAttribute("loggedIn") == null) {

// User is not logged in, redirect to the login page

httpResponse.sendRedirect("login.jsp");

} else {

// User is logged in, continue with the request

chain.doFilter(request, response);

}

}

public void destroy() {

// Clean up resources if necessary (optional)

}

}

* The AuthenticationFilter checks if the session has a loggedIn attribute.
* If the user is not logged in, they are redirected to the **login page**.
* If the user is authenticated, the request continues to the servlet or next filter in the chain.

**Advance topics**

**Implement a database connection (JDBC) to validate users in the login system.**

**Create a servlet-based MVC framework where Servlets handle logic, JSP handles presentation, and a database stores user data.**

**Implement an AJAX-based form submission where the servlet returns a JSON response.**

**1. Implementing a Database Connection (JDBC) to Validate Users in the Login System**

In a real-world scenario, user credentials are typically stored in a **database**. Instead of hardcoding the credentials, we use **JDBC (Java Database Connectivity)** to interact with a relational database to validate users.

**Step 1: Set Up JDBC Connection**

You will need to configure the connection to the database, typically using a **JDBC driver** (such as MySQL or PostgreSQL). Ensure that you have a **database** and **table** for storing user credentials.

**Example Database Schema**

sql

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT,

username VARCHAR(50),

password VARCHAR(50)

);

* The **users table** contains id, username, and password fields.
* For security, passwords should be hashed, but for simplicity, we'll use plaintext in this example.

**Step 2: JDBC Code to Validate Users**

**JDBC Utility Class to Connect to the Database**

java

import java.sql.\*;

public class DBUtil {

private static final String URL = "jdbc:mysql://localhost:3306/mydb";

private static final String USER = "root";

private static final String PASSWORD = "password";

public static Connection getConnection() throws SQLException {

return DriverManager.getConnection(URL, USER, PASSWORD);

}

}

* DBUtil.getConnection() establishes a connection to the MySQL database.

**Step 3: Modify the LoginServlet to Use JDBC**

Now, modify the LoginServlet to validate the user credentials by querying the database instead of hardcoding them.

java

@WebServlet("/login")

public class LoginServlet extends HttpServlet {

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

String username = request.getParameter("username");

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

try (Connection conn = DBUtil.getConnection()) {

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement statement = conn.prepareStatement(sql);

statement.setString(1, username);

statement.setString(2, password); // In practice, you should hash the password!

ResultSet resultSet = statement.executeQuery();

if (resultSet.next()) {

// User found, valid credentials

RequestDispatcher dispatcher = request.getRequestDispatcher("/welcome");

dispatcher.forward(request, response);

} else {

// Invalid credentials

request.setAttribute("errorMessage", "Invalid username or password");

RequestDispatcher dispatcher = request.getRequestDispatcher("/login.jsp");

dispatcher.forward(request, response);

}

} catch (SQLException e) {

e.printStackTrace();

response.sendError(HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR, "Database error");

}

}

}

* The **LoginServlet** now validates the username and password by querying the database.
* If the credentials are valid, it forwards the request to the **WelcomeServlet**. Otherwise, it displays an error message.

**2. Create a Servlet-Based MVC Framework (Model-View-Controller)**

The **MVC (Model-View-Controller)** pattern is widely used in web applications. It separates the application's logic into three components:

* **Model**: Represents the data and business logic (e.g., interacting with the database).
* **View**: Represents the user interface (e.g., JSP pages).
* **Controller**: Handles user requests and manages communication between the Model and View (e.g., Servlets).

**Step 1: Set Up the Model (Database Interaction)**

The **Model** component contains the business logic, such as interacting with the database. You already have the DBUtil class to manage database connections. Let's now create a **UserModel** to interact with the users table.

java

public class UserModel {

public boolean validateUser(String username, String password) {

try (Connection conn = DBUtil.getConnection()) {

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";

PreparedStatement statement = conn.prepareStatement(sql);

statement.setString(1, username);

statement.setString(2, password);

ResultSet resultSet = statement.executeQuery();

return resultSet.next(); // Return true if user exists

} catch (SQLException e) {

e.printStackTrace();

return false;

}

}

}

* **UserModel** handles the interaction with the database to validate users.

**Step 2: Create the Controller (Servlet)**

The **Controller** will handle requests and pass data between the Model and View. You can reuse the LoginServlet for this purpose but refactor it to use the UserModel.

java

@WebServlet("/login")

public class LoginController extends HttpServlet {

private UserModel userModel = new UserModel();

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

String username = request.getParameter("username");

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

if (userModel.validateUser(username, password)) {

RequestDispatcher dispatcher = request.getRequestDispatcher("/welcome.jsp");

dispatcher.forward(request, response);

} else {

request.setAttribute("errorMessage", "Invalid username or password");

RequestDispatcher dispatcher = request.getRequestDispatcher("/login.jsp");

dispatcher.forward(request, response);

}

}

}

* The **LoginController** handles the logic of validating user credentials by calling the **UserModel**. If validation is successful, it forwards the request to a **JSP page** for the view.

**Step 3: Create the View (JSP Pages)**

Create simple JSP pages to serve as views for the login form and the welcome page.

**login.jsp (Login Form)**

jsp

<form action="login" method="post">

<input type="text" name="username" placeholder="Username" required />

<input type="password" name="password" placeholder="Password" required />

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

</form>

<% String errorMessage = (String) request.getAttribute("errorMessage"); %>

<% if (errorMessage != null) { %>

<div class="error"><%= errorMessage %></div>

<% } %>

**welcome.jsp (Welcome Page)**

jsp

<h1>Welcome, you are logged in!</h1>

* The **login.jsp** displays the login form and shows error messages if login fails.
* The **welcome.jsp** displays a welcome message upon successful login.

**3. Implementing an AJAX-Based Form Submission**

AJAX (Asynchronous JavaScript and XML) allows you to submit forms without refreshing the page. This improves user experience by making the application more responsive.

**Step 1: Modify the Login Form to Use AJAX**

Use **JavaScript** to send the login form data asynchronously to the server and handle the response.

html

<form id="loginForm">

<input type="text" id="username" placeholder="Username" required />

<input type="password" id="password" placeholder="Password" required />

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

</form>

<div id="errorMessage"></div>

<script>

document.getElementById("loginForm").addEventListener("submit", function(event) {

event.preventDefault(); // Prevent the form from submitting normally

var username = document.getElementById("username").value;

var password = document.getElementById("password").value;

var xhr = new XMLHttpRequest();

xhr.open("POST", "login", true);

xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

// Handle success (e.g., redirect or display message)

document.location.href = "welcome.jsp";

}

};

xhr.send("username=" + username + "&password=" + password);

});

</script>

* The **AJAX** request sends the username and password to the **LoginServlet**.
* If successful, the user is redirected to the **welcome.jsp** without a page refresh.

**Step 2: Modify the LoginServlet to Return JSON**

The **LoginServlet** should return a **JSON response** instead of redirecting or forwarding.

java

@WebServlet("/login")

public class LoginServlet extends HttpServlet {

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

String username = request.getParameter("username");

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

response.setContentType("application/json");

response.setCharacterEncoding("UTF-8");

PrintWriter out = response.getWriter();

if (userModel.validateUser(username, password)) {

out.write("{\"status\":\"success\", \"message\":\"Login successful\"}");

} else {

out.write("{\"status\":\"error\", \"message\":\"Invalid username or password\"}");

}

}

}

* The **LoginServlet** now responds with a **JSON object**, which can be processed by the **AJAX** request.