Ampliación de conocimientos de JSP

Uso de clases personalizadas en un proyecto JSP

Para utilizar una clase Java personalizada en un archivo JSP dentro de un proyecto Jakarta EE en IntelliJ IDEA Ultimate, puedes seguir estos pasos:

1. Colocar la Clase en el Directorio `WEB-INF/classes` (Método Sencillo para Desarrollo)

Este es el método más rápido para probar durante el desarrollo, pero no es el más recomendado para despliegues de producción.

- **Paso 1: Crea tu clase Java.**
  Por ejemplo, crea un archivo `MiClase.java` en tu proyecto.

```java
// src/main/java/com/tuempresa/miproyecto/MiClase.java (o donde sea tu paquete)
package com.tuempresa.miproyecto;

public class MiClase {
    private String mensaje;

    public MiClase(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String saludar() {
        return "Hola desde MiClase: " + mensaje;
    }
}
```

- **Paso 2: Compila tu clase.**
  IntelliJ IDEA normalmente compila tus clases Java automáticamente. Si no, puedes ir a `Build > Build Project` o `Build > Rebuild Project`. Asegúrate de que el archivo `.class` generado se coloque en la ruta de salida de tu módulo (generalmente `target/classes` si usas Maven/Gradle, y luego IntelliJ lo copia a `WEB-INF/classes` cuando despliegas).
- **Paso 3: Accede a la clase en tu JSP.**

```
<%-- webapp/miPagina.jsp --%>
<%@ page import="com.tuempresa.miproyecto.MiClase" %>
<!DOCTYPE html>
<html>
<head>
    <title>Uso de MiClase en JSP</title>
</head>
<body>
    <h1>Ejemplo de Uso de Clase Propia</h1>

    <%
        // Crear una instancia de MiClase
        MiClase miObjeto = new MiClase("¡Este es un mensaje desde JSP!");

        // Llamar a un método de la clase
        String saludo = miObjeto.saludar();
    %>

    <p><%= saludo %></p>

    <%
        // También puedes guardar el objeto en un ámbito (request, session, applica
        request.setAttribute("miObjeto", miObjeto);
    %>

    <p>Mensaje desde el atributo de request: <%= ((MiClase)request.getAttribute("mi

</body>
</html>
```

2. Usar JavaBeans con `jsp:useBean` (Recomendado para Modelos Simples)

Esta es la forma más idiomática y recomendada para usar objetos Java en JSP, especialmente si tu clase sigue la convención de JavaBeans (constructor sin argumentos y métodos `get` / `set` para sus propiedades).

- **Paso 1: Asegúrate de que tu clase sea un JavaBean.**
  Esto significa que debe tener un constructor público sin argumentos (incluso si tiene otros constructores) y métodos `getPropiedad()` y `setPropiedad()` para cada propiedad que quieras exponer.

```java
// src/main/java/com/tuempresa/miproyecto/Usuario.java
package com.tuempresa.miproyecto;

import java.io.Serializable; // Buena práctica para JavaBeans

public class Usuario implements Serializable {
    private String nombre;
    private int edad;

    // Constructor sin argumentos (obligatorio para JavaBeans)
    public Usuario() {
    }

    // Constructor con argumentos (opcional, pero útil)
    public Usuario(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    public String obtenerSaludoPersonalizado() {
        return "Hola, " + nombre + ". Tienes " + edad + " años.";
    }
}
```

- **Paso 2: Accede a la clase en tu JSP usando `jsp:useBean`.**

```
<%-- webapp/perfilUsuario.jsp --%>
<!DOCTYPE html>
<html>
<head>
    <title>Perfil de Usuario</title>
</head>
<body>
    <h1>Perfil del Usuario</h1>

    <%--
        jsp:useBean busca un objeto de la clase especificada en el ámbito especificado.
        Si no lo encuentra, lo crea e inicializa.
        id: el nombre de la variable que usarás en el JSP
        class: el nombre completo de la clase (incluyendo el paquete)
        scope: el ámbito donde se buscará/almacenará el bean (page, request, session, a
    --%>
    <jsp:useBean id="usuarioBean" class="com.tuempresa.miproyecto.Usuario" scope="reque

    <%--
        Puedes establecer propiedades usando jsp:setProperty,
        ya sea manualmente o automáticamente desde parámetros de request.
    --%>
    <%-- Si viniera de un formulario: <jsp:setProperty name="usuarioBean" property="*"
    <%
        // Para este ejemplo, establecemos las propiedades manualmente
        usuarioBean.setNombre("Ana López");
        usuarioBean.setEdad(30);
    %>

    <p>Nombre: <jsp:getProperty name="usuarioBean" property="nombre" /></p>
    <p>Edad: <jsp:getProperty name="usuarioBean" property="edad" /></p>
    <p>Saludo: <%= usuarioBean.obtenerSaludoPersonalizado() %></p>

</body>
</html>
```

3. Usar JSTL (JSP Standard Tag Library) con EL (Expression Language) y Modelos (Recomendado para Aplicaciones Robustas)

Esta es la forma más moderna, limpia y preferida de trabajar con JSP y Java, ya que separa mejor la lógica de presentación del código Java. Necesitarás agregar las dependencias de JSTL a tu proyecto.

- **Paso 1: Agrega las dependencias de JSTL.**
  Si usas Maven, agrega esto a tu `pom.xml`:

```
<dependencies>
    <dependency>
        <groupId>jakarta.platform</groupId>
        <artifactId>jakarta.jakartaee-api</artifactId>
        <version>10.0</version> <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>jakarta.servlet.jsp.jstl</groupId>
        <artifactId>jakarta.servlet.jsp.jstl-api</artifactId>
        <version>3.0.0</version> </dependency>
    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>jakarta.servlet.jsp.jstl</artifactId>
        <version>3.0.1</version> </dependency>

    </dependencies>
```

  Si usas Gradle, agrega a `build.gradle`:

```
dependencies {
    // Jakarta EE API (depende de tu configuración específica)
    providedCompile 'jakarta.platform:jakarta.jakartaee-api:10.0' // O tu versión

    // JSTL para Jakarta EE
    implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api:3.0.0'
    runtimeOnly 'org.glassfish.web:jakarta.servlet.jsp.jstl:3.0.1'

    // Para Java EE 8 o anterior
    // implementation 'javax.servlet:jstl:1.2'
    // runtimeOnly 'taglibs:standard:1.1.2'
}
```

- **Paso 2: Crea tus clases de modelo (POJOs/JavaBeans).**
  Utiliza la misma clase `Usuario` del ejemplo anterior (que es un JavaBean).
- **Paso 3: Prepara tus datos en un Servlet o Controlador.**
  En una aplicación real, no harías la lógica de negocio directamente en el JSP. Usarías un Servlet o un controlador (por ejemplo, con Spring MVC, JSF, etc.) para preparar los datos y luego pasarlos al JSP.

```java
// src/main/java/com/tuempresa/miproyecto/UsuarioServlet.java
package com.tuempresa.miproyecto;

import jakarta.servlet.ServletException;
import jakarta.servlet.annotation.WebServlet;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

@WebServlet("/usuarios")
public class UsuarioServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        // Lógica de negocio para obtener datos
        Usuario usuario1 = new Usuario("Carlos Ruiz", 25);
        Usuario usuario2 = new Usuario("María García", 40);

        List<Usuario> listaUsuarios = new ArrayList<>();
        listaUsuarios.add(usuario1);
        listaUsuarios.add(usuario2);

        // Poner los objetos en el ámbito de la solicitud para que el JSP los vea
        request.setAttribute("usuarioPrincipal", usuario1);
        request.setAttribute("usuariosRegistrados", listaUsuarios);

        // Redirigir al JSP para la vista
        request.getRequestDispatcher("/WEB-INF/jsp/mostrarUsuarios.jsp").forward(re
    }
}
```

- **Paso 4: Accede a los datos en tu JSP usando JSTL y EL.**

```jsp
<%-- webapp/WEB-INF/jsp/mostrarUsuarios.jsp --%>
<%@ taglib prefix="c" uri="jakarta.tags.core" %> <%-- O "http://java.sun.com/jsp/jstl/c
<!DOCTYPE html>
<html>
<head>
    <title>Lista de Usuarios</title>
</head>
<body>
    <h1>Detalles del Usuario Principal</h1>
    <p>Nombre: ${usuarioPrincipal.nombre}</p>
    <p>Edad: ${usuarioPrincipal.edad}</p>
    <p>Saludo: ${usuarioPrincipal.obtenerSaludoPersonalizado()}</p>

    <h2>Usuarios Registrados</h2>
    <c:choose>
        <c:when test="${not empty usuariosRegistrados}">
            <ul>
                <c:forEach var="usuario" items="${usuariosRegistrados}">
                    <li>${usuario.nombre} (${usuario.edad} años)</li>
                </c:forEach>
            </ul>
        </c:when>
        <c:otherwise>
            <p>No hay usuarios registrados.</p>
        </c:otherwise>
    </c:choose>

</body>
</html>
```

Nota importante sobre las URIs de las Taglibs:

- Para **Jakarta EE 9+** (JSTL 3.0+), la URI correcta para el core de JSTL es `jakarta.tags.core`.
- Para **Java EE 8 o anterior** (JSTL 1.2), la URI correcta es `http://java.sun.com/jsp/jstl/core`. Asegúrate de usar la URI correcta según la versión de Jakarta EE/Java EE que estés utilizando en tu proyecto de IntelliJ.

Configuración en IntelliJ IDEA Ultimate (Jakarta EE Web Application)

IntelliJ IDEA Ultimate está diseñado para manejar esto de manera muy fluida:

1. **Estructura del Proyecto:** Asegúrate de que tus clases Java estén en la carpeta de fuentes (`src/main/java` si usas Maven/Gradle, o tu carpeta de fuentes configurada).
2. **Artefacto de Despliegue (WAR):**
   - Ve a `File > Project Structure...` (`Ctrl+Alt+Shift+S`).
   - Selecciona `Artifacts` en el panel izquierdo.
   - Asegúrate de tener un artefacto de tipo `Web Application: Exploded` o `Web Application: Archive` (WAR).
   - En la pestaña `Output Layout` , verifica que la salida de tus módulos (clases compiladas) esté incluida. IntelliJ lo hará automáticamente si sigues la estructura de proyecto estándar, colocando tus `.class` files en `WEB-INF/classes` .
3. **Configuración del Servidor de Aplicaciones:**
   - Ve a `Run > Edit Configurations...` .
   - Selecciona tu configuración de `Tomcat Server` (o el servidor de aplicaciones que uses).
   - En la pestaña `Deployment` , asegúrate de que tu artefacto web esté configurado para ser desplegado. La "Application context" será la URL base de tu aplicación (ej., `/miaplicacion`).
4. **Ejecutar/Depurar:**
   - Una vez configurado, puedes ejecutar tu aplicación (`Run > Run 'TuConfiguracionTomcat'`). IntelliJ compilará, empaquetará el WAR y lo desplegará en tu servidor de aplicaciones.

Puntos Clave a Recordar:

- **Compilación:** IntelliJ IDEA compila automáticamente tus clases Java. Asegúrate de que no haya errores de compilación.
- **Classpath:** Cuando se despliega un WAR, el contenedor de servlets (como Tomcat, GlassFish, WildFly) pone las clases de `WEB-INF/classes` y los JARs de `WEB-INF/lib` en el classpath de la aplicación web, lo que permite que tus JSPs las vean.
- **Separación de Responsabilidades (MVC):** Aunque puedes poner lógica en JSP usando scriptlets (`<% %>`), la mejor práctica en aplicaciones Jakarta EE modernas es usar Servlets (o frameworks MVC como Spring MVC/Jakarta Faces/Struts) para manejar la lógica de negocio y pasar los datos a los JSPs (vistas) usando JavaBeans, JSTL y EL. Esto sigue el patrón MVC (Modelo-Vista-Controlador).
- **Rutas de JSP:** Si colocas tus JSPs dentro de `WEB-INF/jsp/` (como en el ejemplo de JSTL), los usuarios no podrán acceder a ellos directamente por la URL, lo cual es una buena práctica de seguridad y un mejor control del flujo de la aplicación a través de Servlets.