



JSP

Java Server Pages

Jakarta Server Pages

Contenidos

Introducción	2
Arquitectura cliente-servidor	4
Etiquetas JSP	5
Servlets	3
¿JSP como servlet o servlet como JSP?	4
Métodos de Servlet	5
Creación de un Servlet	5
Interacción entre un JSP y un Servlet	8
Pasaje/alta de datos mediante POST	8
Recuperar datos desde un Servlet a un JSP mediante GET	12

Introducción

Jakarta Server Pages (JSP; anteriormente **Java Server Pages**) es un conjunto de tecnologías que ayudan a los desarrolladores de software a crear páginas web generadas dinámicamente basadas en HTML , XML , SOAP u otros tipos de documentos. Lanzado en 1999 por Sun Microsystems como Java Server Pages, JSP es similar a PHP y ASP, pero usa el lenguaje de programación Java. La última versión lanzada por la empresa Oracle de **Java Enterprise Edition** es de agosto de 2017. Actualmente el proyecto está en manos de la Eclipse Foundation y por cuestiones legales adoptó el nombre de **Jakarta Enterprise Edition**. JSP forma parte de **Jakarta Enterprise Edition** (**JEE**, lo que anteriormente era *Java Enterprise Edition*). A continuación se presenta un detalle de la evolución de **JEE**.

Platform version	Released	Java SE Support	Important Changes
Jakarta EE 10	2022-09-13	Java SE 17 Java SE 11	Removal of deprecated items in Servlet, Faces, CDI and EJB (Entity Beans and Embeddable Container). CDI-Build Time.
Jakarta EE 9.1	2021-05-25	Java SE 11 Java SE 8	JDK 11 support
Jakarta EE 9	2020-12-08	Java SE 8	API namespace move from <code>javax</code> to <code>jakarta</code>
Jakarta EE 8	2019-09-10	Java SE 8	Full compatibility with Java EE 8
Java EE 8	2017-08-31	Java SE 8	HTTP/2 and CDI based Security
Java EE 7	2013-05-28	Java SE 7	WebSocket , JSON and HTML5 support
Java EE 6	2009-12-10	Java SE 6	CDI managed Beans and REST
Java EE 5	2006-05-11	Java SE 5	Java annotations
J2EE 1.4	2003-11-11	J2SE 1.4	WS-I interoperable web services
J2EE 1.3	2001-09-24	J2SE 1.3	Java connector architecture
J2EE 1.2	1999-12-17	J2SE 1.2	Initial specification release

Evolución de JEE.

Como se ve a partir de la versión Jakarta EE 9 se pasó de los paquetes **javax** a paquetes **jakarta**, esta cuestión se ve reflejada en los *imports* que debemos realizar en nuestras aplicaciones. Más adelante se ejemplifica este asunto con la creación de Servlets.

Para contextualizar brevemente sobre JSP, se presenta la siguiente imagen en donde se puede ver que JSP no es más que una tecnología dentro de JEE (sea Java EE o Jakarta EE) y de la API (Application Programming Interface o Interfaz de Programación de Aplicaciones) de java.



API de Java con la extensión Enterprise Edition.

En la esquina inferior derecha de la imagen también se muestran algunas de las compilaciones de OpenJDK (Oracle, Eclipse Foundation, Microsoft, Redhat, ..) por citar algunas. En el presente curso se está trabajando con la compilación de Oracle para Java SE, pero podría usarse tranquilamente la de la Eclipse Foundation que curiosamente ofrece por ejemplo versión de 32 bits para windows. Note que Jakarta EE extiende a la API standard edition de Java, tal como lo hacía Java EE en su momento.

Trabajar con Jakarta no debería presentar demasiados problemas debido a que los mismos IDEs (Netbeans, Eclipse, etc.) hacen y sugieren el cambio de paquetes necesarios para trabajar con la nueva tecnología. Así que no nos debemos

preocupar demasiado al respecto. Es decir, podemos buscar información y ejemplos sobre Java Server Pages y luego el IDE se encargará de sugerir los cambios necesarios.

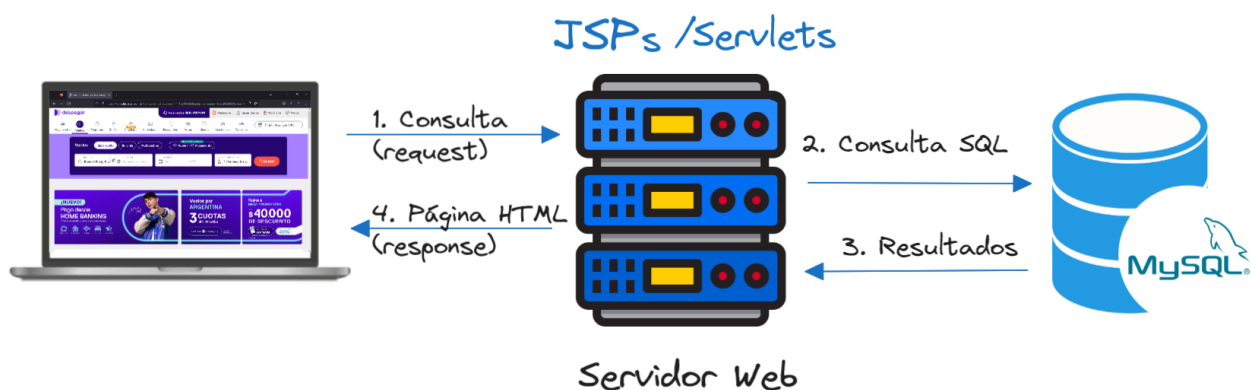
Mucha de la documentación actual se refiere a Java Server Pages, ya que la última versión lanzada por Oracle es todavía “nueva”. Sin embargo, esto irá cambiando poco a poco para finalmente ser totalmente reemplazado en algún momento por Jakarta Server Pages.

Links hacia la documentación oficial de las distintas APIs de Java:

- Java SE: <https://eclipse.dev/openj9/docs/api-jdk17/>
- Java EE: <https://docs.oracle.com/javaee/7/api/>
- Jakarta EE: <https://jakarta.ee/specifications/platform/9/apidocs/>

Arquitectura Cliente-Servidor

Una aplicación **JEE** normalmente está compuesta por un cliente, un servidor y una base de datos, como se muestra en la siguiente figura.



Aplicación JEE convencional.

Como se ve el cliente, un navegador web como firefox, hace una consulta (request¹) y el servidor web (puede ser apache Tomcat) la procesa para este ejemplo con JSP. Luego se manda la consulta a la Base de Datos y ésta devuelve los datos solicitados. Finalmente, el servidor devuelve una página HTML al navegador. Note que el código JSP no viaja hacia la página sino que lo que llega al navegador es simple HTML.

Para implementar y ejecutar JSP, se requiere un servidor web compatible con un contenedor de servlets , como Apache Tomcat.

Etiquetas JSP

Cada JSP contendrá mayoritariamente etiquetas HTML y CSS, como todo frontend, sin embargo, podrá utilizar *etiquetas especiales* para especificar porciones de código Java en donde sea necesario. Por ejemplo:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Página JSP</title>
  </head>
  <body>
    <h1>Hola desde HTML!</h1>
    <% String variable = "Hola desde Java"; %>
    <%= variable %>
  </body>
</html>
```

JSP Hola Mundo

Para poder ejecutar una aplicación o página JSP es necesario que la misma sea ejecutada con un servidor Web, como por ejemplo **Tomcat** o Glassfish.

¹ Como se vio al comienzo de este capítulo una request puede hacerse mediante distintos métodos: get, post, put, delete, patch, etc.

En el ejemplo de la ilustración anterior se pueden observar dos tipos de “Hola”, uno llevado a cabo directamente en una etiqueta HTML `<h1>`, y otro a partir de una *variable* Java que fue declarada e inicializada con el mensaje “Hola desde Java”; sin embargo, no es posible hacer esto sin especificar que se trata de código Java, para poder hacerlo se utilizan una serie de etiquetas, entre las cuales las principales y más utilizadas se especifican a continuación:

Etiqueta	Uso/Descripción
<code><%-- --%></code>	Apertura y cierre para realizar comentarios. Por ejemplo: <code><%-- esto es un comentario --%></code>
<code><%@ %></code>	Apertura y cierre para directivas/atributos de configuración de JSP. Por ejemplo: <code><%@ page language='java' contentType='text/html' %></code>
<code><% %></code>	Apertura y cierre para inclusión de sentencias o código Java en general. Esto no es visto/percebido por el usuario. Por ejemplo: <code><% if (numero > numero2) {...} %></code>
<code><%= %></code>	Apertura y cierre para mostrar el resultado de una expresión o contenido de una variable. Lo que se indique aquí será visualizado por el usuario en el apartado de HTML dentro del JSP que se indique. Por ejemplo: <code><%= nombre %></code>
<code><%! %></code>	Apertura y cierre para hacer uso exclusivo de declaración de variables y métodos de instancia. Que se compartirán entre varios JSP asociados al mismo servlet. Nota: Para declarar variables locales se usar <code><% %></code> .

Etiquetas JSP

Las páginas JSP utilizan varios delimitadores para funciones de secuencias de comandos . El más básico es `<% ... %>`, que incluye un **scriptlet** JSP. Un scriptlet es un fragmento de código Java que se ejecuta cuando el usuario solicita la página.

Otros delimitadores comunes incluyen `<%= ... %>` para **expresiones**, donde el scriptlet y los delimitadores se reemplazan con el resultado de evaluar la expresión, y **directivas** indicadas con `<%@ ... %>`.

Ejemplo

Lo siguiente sería un bucle for válido en una página JSP:

```
<p> Contar hasta tres: </p>
<% for ( int i = 1 ; i < 4 ; i ++ ) { %>
<p> Este número es <%= i %> . </p>
<% } %>
<p> Bien. </p>
```

El resultado que se muestra en el navegador web del usuario sería:

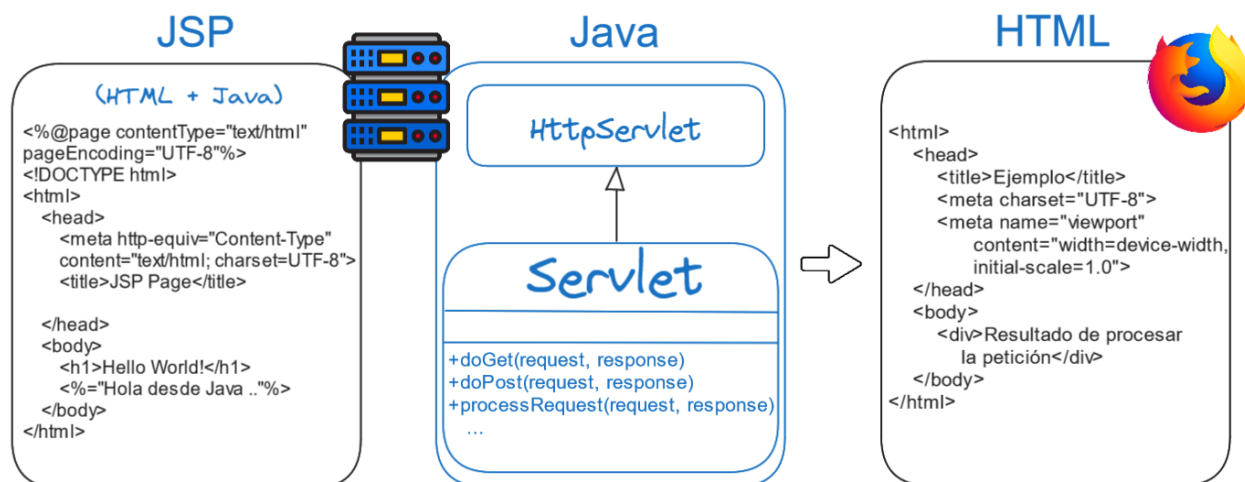
```
Contar hasta tres:
Este número es 1.
Este número es 2.
Este número es 3.
Bien.
```

Por otra parte, para cumplir correctamente con el modelo de capas y el traspaso de información desde un cliente a un servidor, JSP se vale de una ayuda muy importante, que son los Servlets.

Servlets

La definición más común de un servlet es que se trata de una clase Java que se utiliza para poder ampliar las capacidades de un determinado servidor. Su principal característica es la de ser un punto intermedio entre una página JSP y el servidor web (donde generalmente se encuentra el servicio o la lógica de negocio de una aplicación). Como se ve en la siguiente imagen un jsp tiene código Java incrustado dentro de un HTML, mientras que un Servlet es *una clase Java que descende de la clase*

HttpServlet y normalmente devuelve una página HTML. Tanto el jsp como el servlet java permanecen en el servidor, mientras que el html viaja por la red para ser interpretado por un navegador del lado del cliente.



Diferencia entre páginas JSP y un Servlet

Un servlet descende de la clase **HttpServlet** y una cuestión que actualmente hay que tener en cuenta es que HttpServlet pertenecía al paquete:

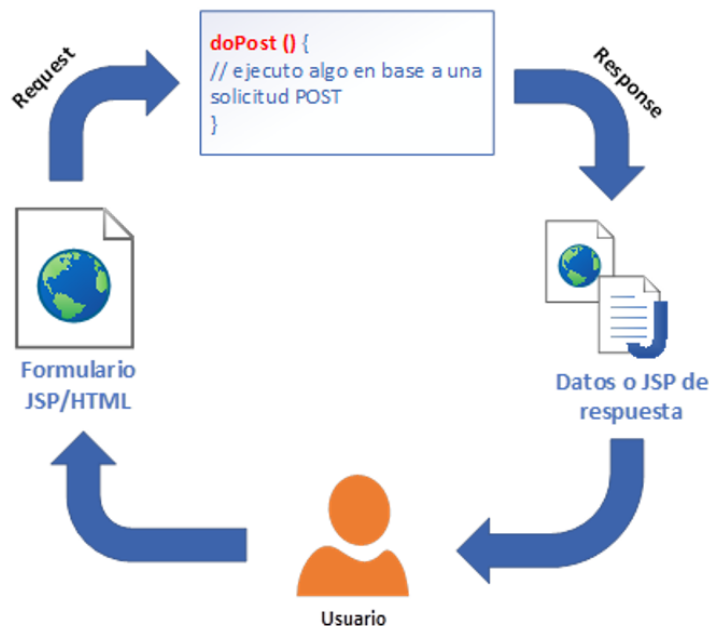
javax.servlet.http.*

Sin embargo, lo más probable es que su IDE le pida el cambio de paquete a

jakarta.servlet.http.*

de la API de **Jakarta Enterprise Edition**.

Un servlet se encarga de recibir peticiones o request desde un cliente, tratarlas y analizar si es necesario realizar alguna solicitud en particular o brindar una determinada respuesta o response. Para poder tratar cada una de las peticiones, utiliza una serie de métodos donde dependiendo del verbo por el cual se reciba la petición (GET, POST, PUT, DELETE, etc) se ejecutará la especificación del que corresponda y luego la mayoría de las veces se devolverá un HTML para que lo visualice el navegador. Un ejemplo con el método POST puede visualizarse en la siguiente ilustración:



Método doPost.

¿JSP como Servlet o Servlet como JSP?

Aunque JSP y servlets parecen a primera vista tecnologías distintas, en realidad el servidor web traduce internamente el JSP a un servlet, lo compila y finalmente lo ejecuta cada vez que el cliente solicita la página JSP. Por ello, en principio, JSPs y servlets ofrecen la misma funcionalidad, aunque sus características los hacen apropiados para distintos tipos de tareas.

Los JSP son mejores para generar páginas con gran parte de contenido estático. Un servlet que realice la misma función debe incluir gran cantidad de sentencias del tipo `out.println()` para producir el HTML. Por el contrario, los servlets son mejores en tareas que generen poca salida, datos binarios o páginas con gran parte de contenido variable.

En proyectos más complejos, lo recomendable es combinar ambas tecnologías: los servlets para el procesamiento de información y los JSP para presentar los datos al cliente. Esta última opción es la que veremos en los ejemplos que se desarrollarán.

Métodos de Servlet

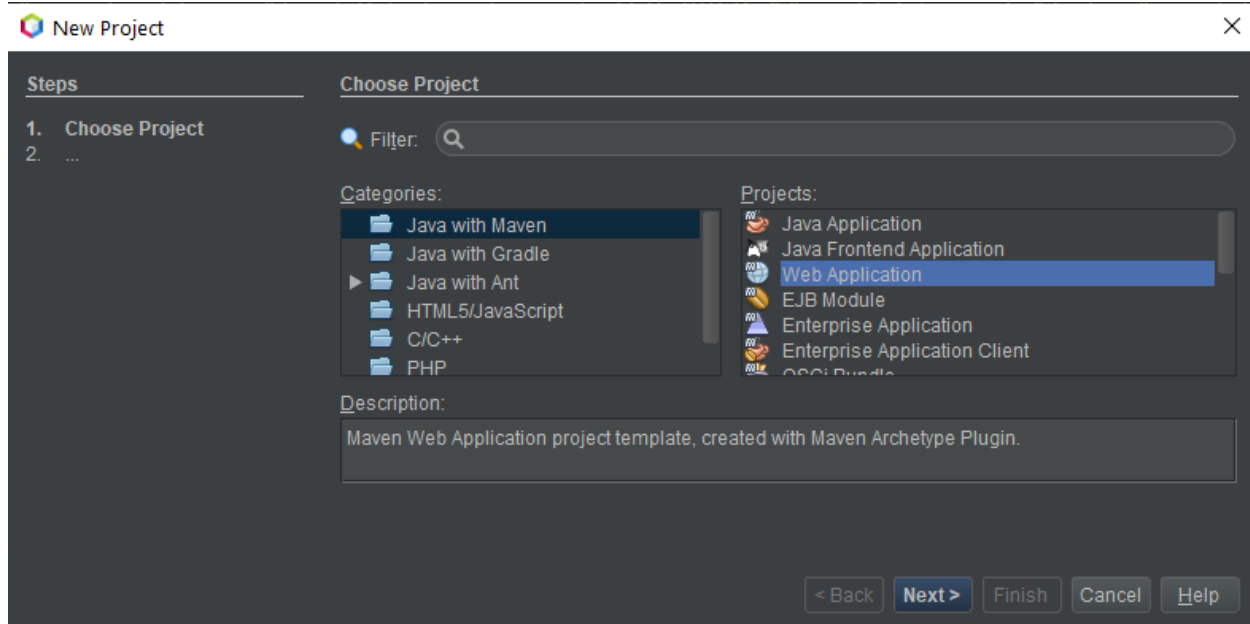
Los servlets tienen diferentes métodos que pueden ser utilizados dependiendo del tipo de solicitud que reciban por parte del cliente, sin embargo, los dos más utilizados son el método `doGet()`, para recibir peticiones mediante GET y `doPost()`, para recibir peticiones mediante POST.

- ☐ **`doGet()`**: Es el método encargado de recibir las solicitudes que provienen mediante GET. Generalmente recibe los parámetros desde la URL de la petición y su principal función es la de solicitar datos del servidor y devolverlos al cliente.
- ☐ **`doPost()`**: Es el método encargado de recibir las solicitudes que provienen mediante POST. Los parámetros, objetos o datos pueden provenir en el header o body de una solicitud, a partir del envío (submit) de un formulario HTML desde el JSP. Su principal función es la de obtener datos desde el cliente para generar cambios en el servidor.

Creación de un Servlet

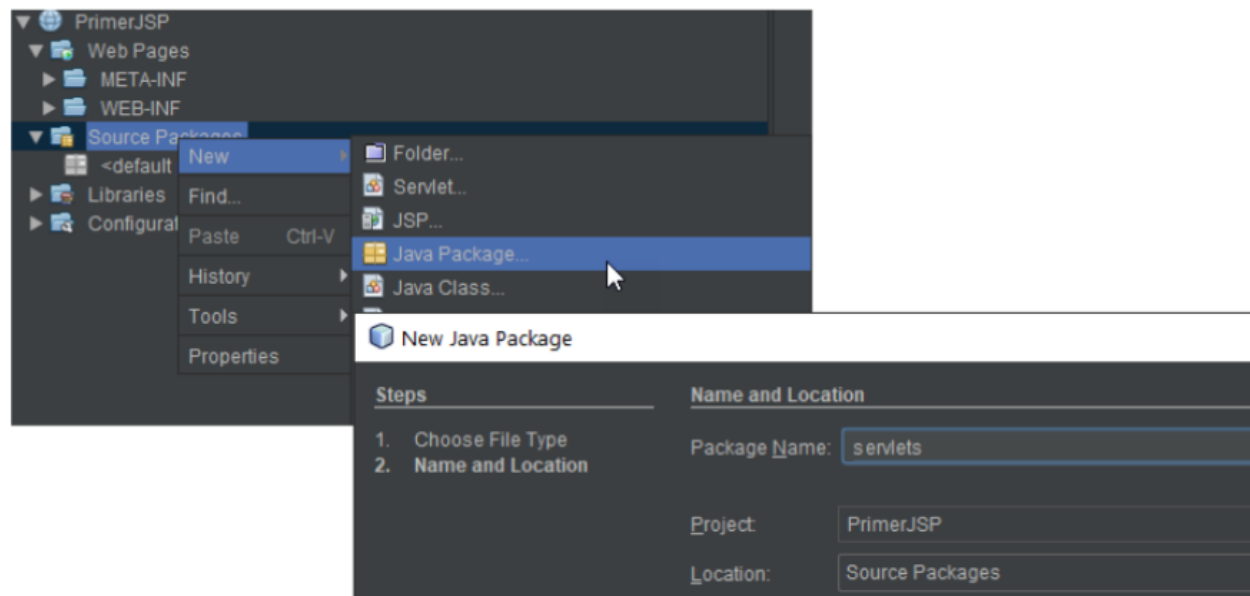
Dependiendo de cada IDE, la creación de servlets es relativamente sencilla. Tomaremos como ejemplo el IDE Netbeans.

PASO 1. Crear un nuevo proyecto Java Web. Lo más correcto para este curso es crear un proyecto **Java With Maven** y dentro de esa opción seleccionar el arquetipo “**Web Application**”):



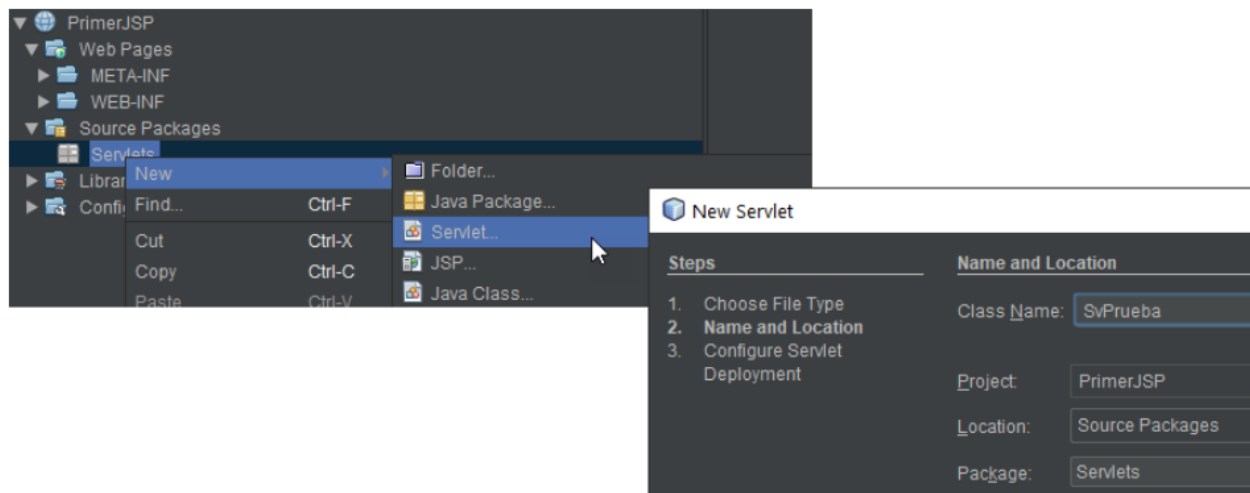
Creación de un nuevo proyecto con Netbeans: Web Application

PASO 2. Crear un nuevo paquete para el guardado de los servlets:



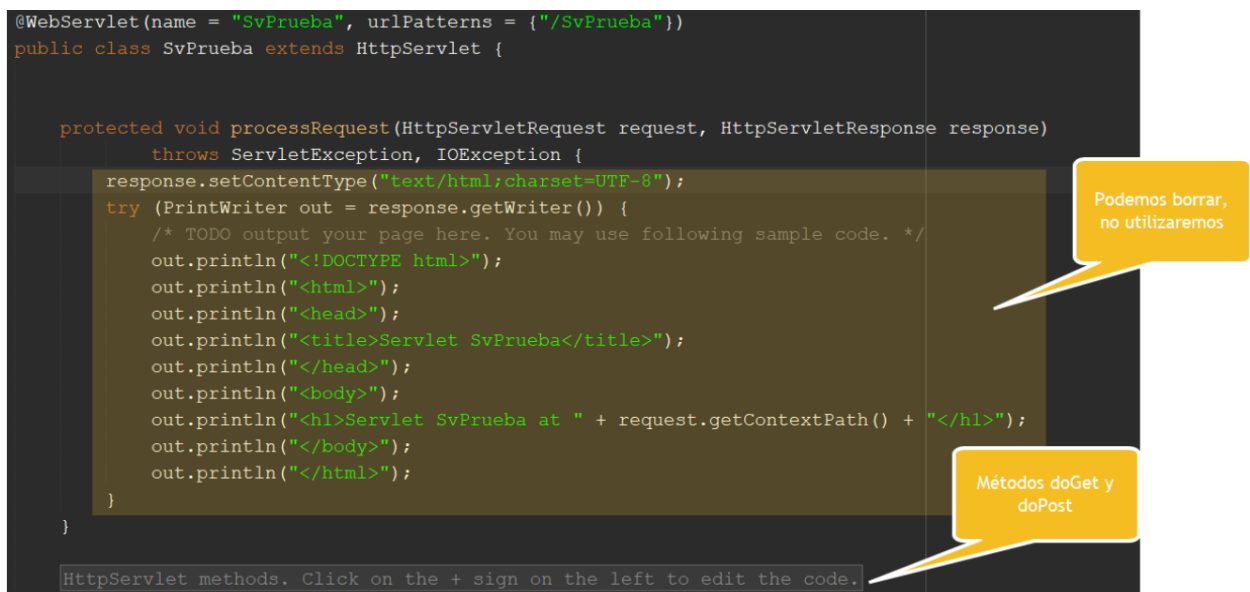
Creación de un Java Package para el almacenamiento de servlets

PASO 3. Click derecho sobre el nuevo paquete creado, luego en *new* y elegimos *Servlet*:



Creación de un servlet

PASO 4. El IDE nos creará de forma automática los métodos `doGet()` y `doPost()`, como así también un apartado por si es necesario transformar el servlet en JSP. Cómo utilizaremos los JSP y servlets de forma separada, este apartado podemos eliminarlo:



Eliminación de código innecesario.

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

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}
```

Métodos doGet y doPost del servlet creado.

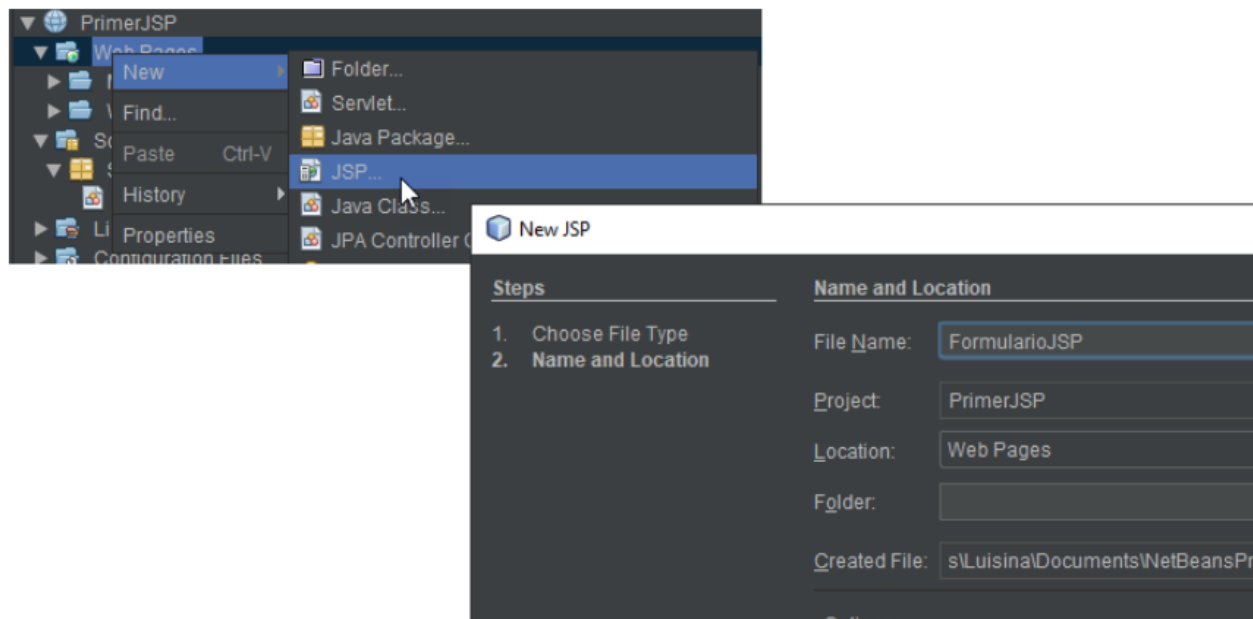
Interacción entre un JSP y un Servlet

Como se mencionó anteriormente, un JSP puede ejecutarse por separado y llamar a un servlet, como así también, ejecutarse por sí mismo dentro de un servlet. Cuando trabajamos con un modelo de capas, donde se ve a un JSP como una entidad separada del servlet (por más que trabajen en conjunto), tenemos que configurar esta conexión e intercambio de información. Para ello, tenemos que considerar si la operación que se hará desde el JSP es de obtención (GET) o creación de nuevos datos (POST). A continuación se citarán los paso a paso de un ejemplo de cada uno.

Pasaje/alta de datos mediante POST

Supongamos un formulario para el alta de un nuevo cliente a un sistema, donde se solicita dni, nombre, apellido y número de teléfono y donde pasaremos, estos datos, desde el JSP al servlet para su posterior tratamiento (por ejemplo un alta en una base de datos). Para ello, seguiremos los siguientes pasos:

PASO 1. A partir de un proyecto Java Web ya creado, se debe crear un nuevo archivo JSP:



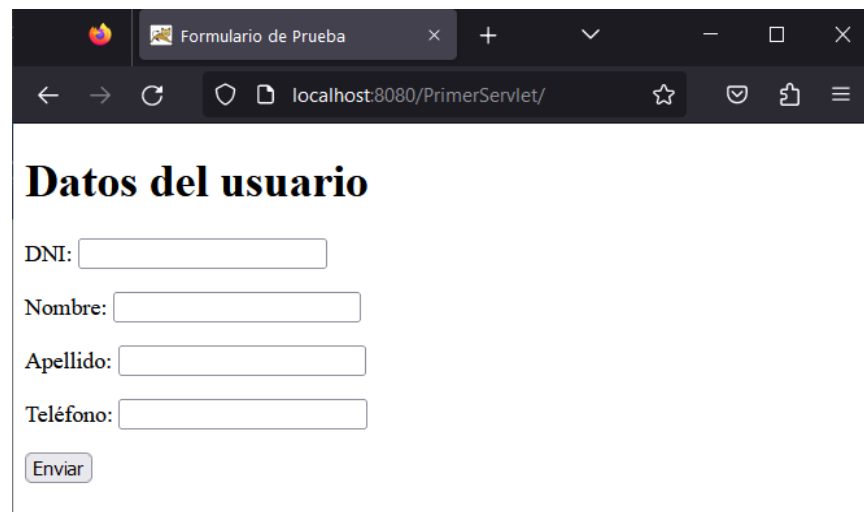
Creación de un FormularioJSP

PASO 2. Una vez creado el JSP, armaremos en él un pequeño formulario HTML para proporcionar los datos desde el lado del cliente. En la siguiente figura se pueden observar las etiquetas HTML dentro del JSP, donde en el formulario, en el atributo *action* de la etiqueta *form*, se hace referencia al servlet que creamos anteriormente “*SvUsuarios*” para hacer la redirección y enviar los datos del formulario al mencionado; al mismo tiempo agregamos el método por el cual será enviado, en este caso el POST. Note además, que para este caso, todo este archivo carece de código Java por lo que tranquilamente podría ser un HTML.

```
1 <%@page contentType="text/html" pageEncoding="UTF-8"%>
2 <!DOCTYPE html>
3 <html>
4 <head>
5     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
6     <title>Formulario de Prueba</title>
7 </head>
8 <body>
9     <h1>Datos del usuario</h1>
10    <form action="SvUsuarios" method="POST">
11        <p><label>DNI: </label><input type="text" name="dni"></p>
12        <p><label>Nombre: </label><input type="text" name="nombre"></p>
13        <p><label>Apellido: </label><input type="text" name="apellido"></p>
14        <p><label>Teléfono: </label><input type="text" name="telefono"></p>
15        <button type="submit">Enviar</button>
16    </form>
17 </body>
18 </html>
```

Formulario.jsp

En la siguiente imagen se especifica cómo se verá en un navegador el formulario dentro del JSP una vez que sea cargado.



Resultado en un navegador del JSP

Una vez que un usuario cargue los datos y haga clic en enviar, los mismos se dirigirán desde el JSP al servlet mediante el método POST, tal y como lo especificamos en las etiquetas HTML del formulario.

PASO 3. En el servlet es necesario configurar el método doPost() para la recepción de cada uno de los datos que vendrán desde el JSP mediante la request, para esto será necesario utilizar el método getParameter(), tal y como puede verse a continuación.

```
14  @WebServlet(name = "SvUsuarios", urlPatterns = {"/SvUsuarios"})
15  public class SvUsuarios extends HttpServlet {
16
17      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
18          throws ServletException, IOException {
19      }
20
21      @Override
22      protected void doGet(HttpServletRequest request, HttpServletResponse response)
23          throws ServletException, IOException {
24          List<Usuario> listaUsuarios = new ArrayList<>();
25          listaUsuarios.add(new Usuario("123", "Luis", "Z", "123"));
26          listaUsuarios.add(new Usuario("222", "Alejandra", "Q", "222"));
27          listaUsuarios.add(new Usuario("333", "Pepé", "Argento", "333"));
28          HttpSession miSesion = request.getSession();
29          miSesion.setAttribute("listaUsuarios", listaUsuarios);
30          response.sendRedirect("mostrarUsuarios.jsp");
31      }
32
33      @Override
34      protected void doPost(HttpServletRequest request, HttpServletResponse response)
35          throws ServletException, IOException {
36          String dni = request.getParameter("dni");
37          String nombre = request.getParameter("nombre");
38          String apellido = request.getParameter("apellido");
39          String telefono = request.getParameter("telefono");
40
41          System.out.println("Dni: " + dni);
42          System.out.println("Nombre: " + nombre);
43          System.out.println("Apellido: " + apellido);
44          System.out.println("Teléfono: " + telefono);
45      }
46
47      @Override
48      public String getServletInfo() {
49          return "Short description";
50      }
51  }
```

Ejemplo de método doPost

Una vez recibidos los datos por el método doPost, podemos pasarlos a la lógica de negocio para realizar operaciones, o guardarlas en una base de datos, o cualquier función que queramos darles.

Una vez realizado esto, tenemos todos los pasos necesarios para levantar una página JSP, recibir datos mediante un formulario y transmitirlos al servlet por el método POST para el posterior tratamiento de los mismos.

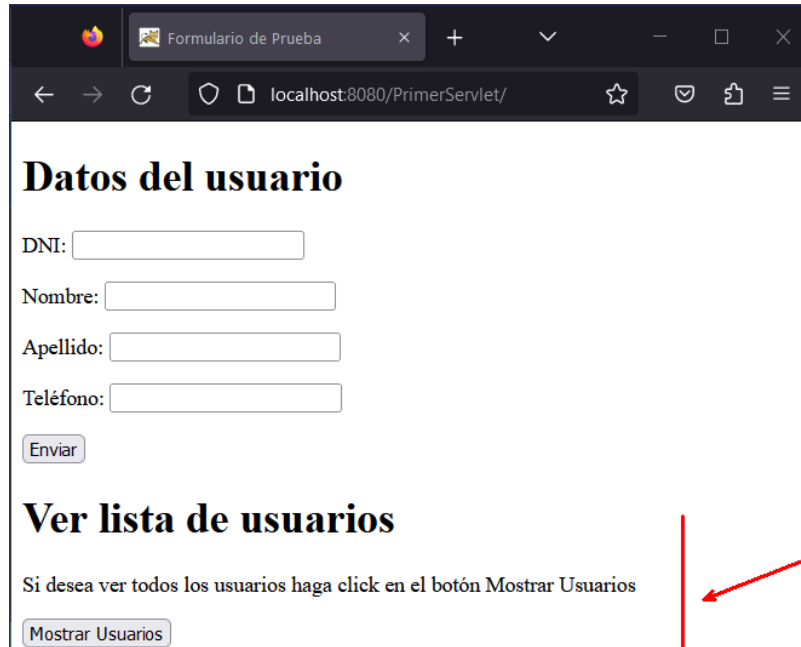
Recuperar datos desde un Servlet a un JSP mediante GET

Supongamos que tenemos en un servlet una lista de clientes que trajimos desde una base de datos y queremos mostrarla en un JSP. Para realizar la solicitud desde el servlet, podemos utilizar un formulario GET que llame al servlet y nos redirija a otra página donde se nos muestran los datos de los clientes almacenados en la lista. A continuación veremos el paso a paso para realizar esto.

PASO 1. A partir de un proyecto Java Web ya creado, agregaremos un nuevo apartado en el JSP “FormularioJSP” para contemplar la solicitud de los usuarios de la lista además del alta de un nuevo usuario.

```
8  <body>
9    <h1>Datos del usuario</h1>
10   <form action="SvUsuarios" method="POST">
11     <p><label>DNI: </label><input type="text" name="dni"></p>
12     <p><label>Nombre: </label><input type="text" name="nombre"></p>
13     <p><label>Apellido: </label><input type="text" name="apellido"></p>
14     <p><label>Teléfono: </label><input type="text" name="telefono"></p>
15     <button type="submit">Enviar</button>
16   </form>
17   <h1>Ver lista de usuarios</h1>
18   <p>Si desea ver todos los usuarios haga click en el botón Mostrar Usuarios</p>
19   <form action="SvUsuarios" method="GET">
20     <button type="submit">Mostrar Usuarios</button>
21   </form>
22 </body>
```

Agregado en JSP para mostrar usuarios



Agregado JSP visto desde un navegador

PASO 2. En el servlet, se debe configurar el método `doGet()` para recibir la request, preparar la lista de usuarios y reenviarla a otro JSP para que sea mostrada. Para poder pasar la lista a otro JSP, es necesario capturar la sesión de la request y setear a la lista como parámetro. De esta manera la lista podrá ser utilizada en cualquier otro JSP mientras dure la sesión.

```
20      @Override
21      protected void doGet(HttpServletRequest request,
22                          HttpServletResponse response) throws ServletException, IOException {
23          List<Usuario> listaUsuarios = new ArrayList<>();
24          listaUsuarios.add(new Usuario("123", "Luis", "2", "123"));
25          listaUsuarios.add(new Usuario("222", "Gerardo", "Q", "222"));
26          listaUsuarios.add(new Usuario("333", "Pepe", "Argento", "333"));
27
28          // seteamos la lista de usuarios como un parámetro
29          // para poder utilizar en cualquier JSP
30          // para ello traemos la sesión de la request
31          HttpSession miSesion = request.getSession();
32          miSesion.setAttribute("listaUsuarios", listaUsuarios);
33
34          // redireccionamos a otro JSP
35          response.sendRedirect("mostrarUsuarios.jsp");
36      }
```

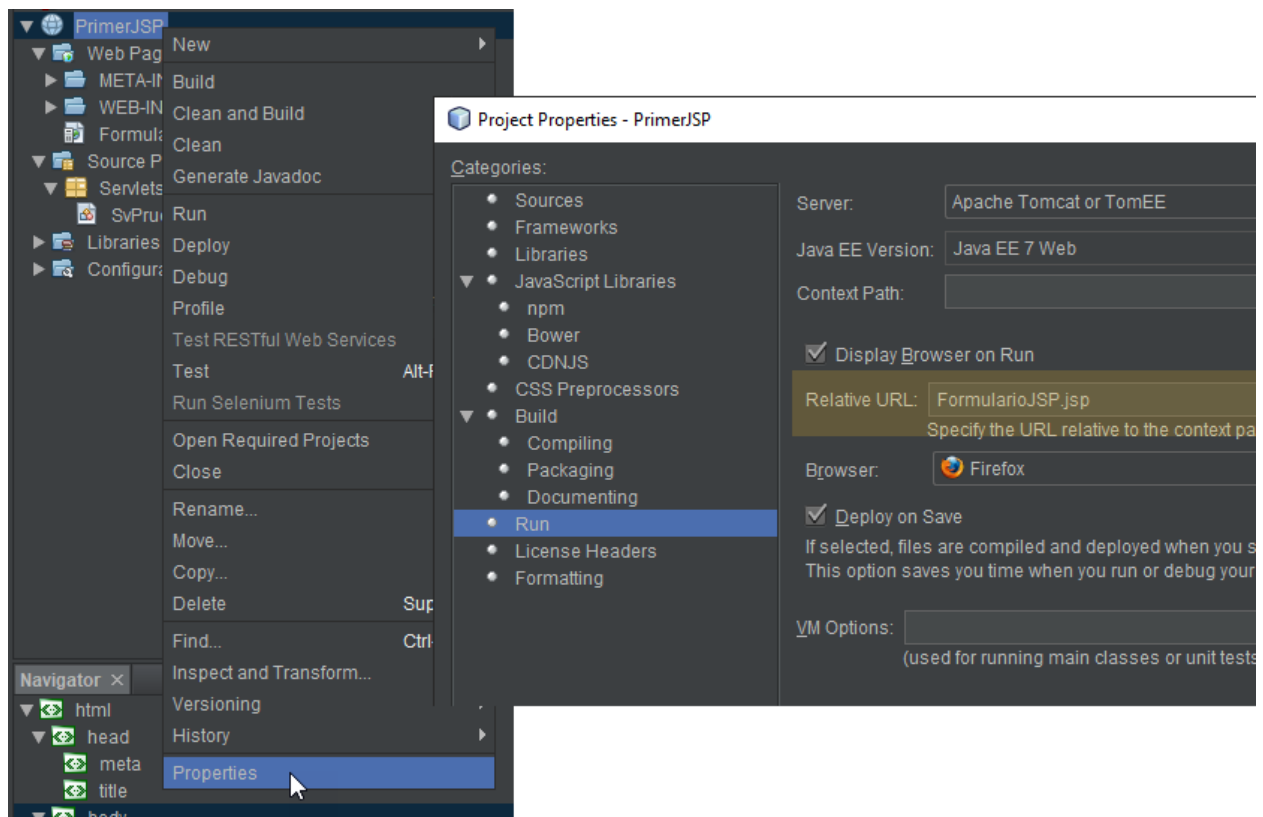
Configuración del método doGet y redirección a otro JSP

PASO 3. Creamos el nuevo JSP “MostrarJSP”, para recibir el parámetro enviado desde el servlet y mostrarlo gráficamente al usuario:

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Mostrar usuarios</title>
</head>
<body>
<h1>Lista de usuarios registrados</h1>
<%
    List<Usuario> lista = (List)request.getSession().getAttribute("listaUsuarios");
    int cont = 1;
    for(Usuario usu: lista){
        %>
        <p>Usuario N°<%=cont%></p>
        <p>DNI: <%=usu.getDni() %></p>
        <p>Nombre: <%=usu.getNombre() %></p>
        <p>Telefono<%=usu.getTelefono() %></p>
        <% cont++; %>
    }%>
</body>
</html>
```

mostrarUsuarios.jsp

PASO 4. Antes de ejecutar nuestra aplicación para probarla, como ahora existen dos archivos JSP, es necesario especificar cuál se ejecutará al iniciar la ejecución de la aplicación Web. Los pasos para realizar esta configuración están en la siguiente Ilustración:



Configurar JSP de inicio

Una vez realizados todos estos pasos, al ejecutar la aplicación y al hacer click en “Mostrar Usuarios”, deberíamos poder visualizar toda la lista de usuarios en un nuevo JSP.