

Módulo 7 – Programación Web con Java

Índice

Java Web

Arquitectura REST y Métodos HTTP 2

JSP (Java Server Pages)

JSP (Java Server Pages) 4

Servlets

Servlets..... 6

Sockets

Sockets..... 19

Arquitectura REST y Métodos HTTP

REST (Transferencia de Estado Representacional) o mejor conocido en inglés como representational state transfer, es un estilo de arquitectura de software que define o establece un conjunto de estándares, propiedades y buenas prácticas que se pueden implementar sobre HTTP. Su principal función es la de permitir que un desarrollo web pueda operar con otros mediante sus estándares a través de internet o de una red. La característica que destaca a REST es el hecho de que es “stateless”, es decir, un protocolo que no posee estado.

En base a la arquitectura REST, en el protocolo HTTP existen varios métodos o verbos que pueden ser utilizados para las comunicaciones de las solicitudes, donde cada uno de ellos tiene una finalidad en particular. Sin embargo, dos de los más utilizados son: el método GET (para obtener información del servidor) y el método POST (para brindar información al servidor).

GET

El método o verbo get, es por excelencia utilizado en el desarrollo de aplicaciones web para enviar una solicitud de “obtención de información” a un servidor desde un cliente. Su forma más común de uso es mediante formularios HTML, donde a partir de una serie de datos que se brindan, por ejemplo, una id, un dni, un nombre, etc se pretende lograr como respuesta una serie de registros o datos.

GET es considerada la forma más sencilla de transferir información al servidor y, de hecho, cuando intentamos acceder a una dirección URL en nuestros navegadores, estamos haciendo inconscientemente una solicitud GET.

Algunas de las principales características del método GET son:

- Los parámetros o información a utilizar son enviados generalmente en la **barra de direcciones**.

Por ejemplo: <https://www.misistema.com/buscar?nombre=Luisina&edad=29>. En este caso estamos pasando los parámetros nombre y edad con sus respectivos valores para obtener una respuesta a partir de los datos de dicha solicitud.

- Los navegadores pueden guardar en caché los resultados de una solicitud obtenida con GET para ser utilizada más tarde o ser cargada con mayor rapidez en otra ocasión.
- No es un método apto para pasaje de datos sensibles (como contraseñas o información similar). En caso de que los datos sean pasados en la barra de direcciones se debe tener sumo cuidado, dado que cualquier persona podrá verlos.

¿Cuándo es recomendado utilizar el método GET? Cuando estamos desarrollando una aplicación y el cliente necesita obtener información por parte del servidor. GET nos puede servir para enviar parámetros, como ser filtros para una

tabla, formas de ordenación (ascendente, descendente, etc), parámetros de búsqueda, entre muchos otros.

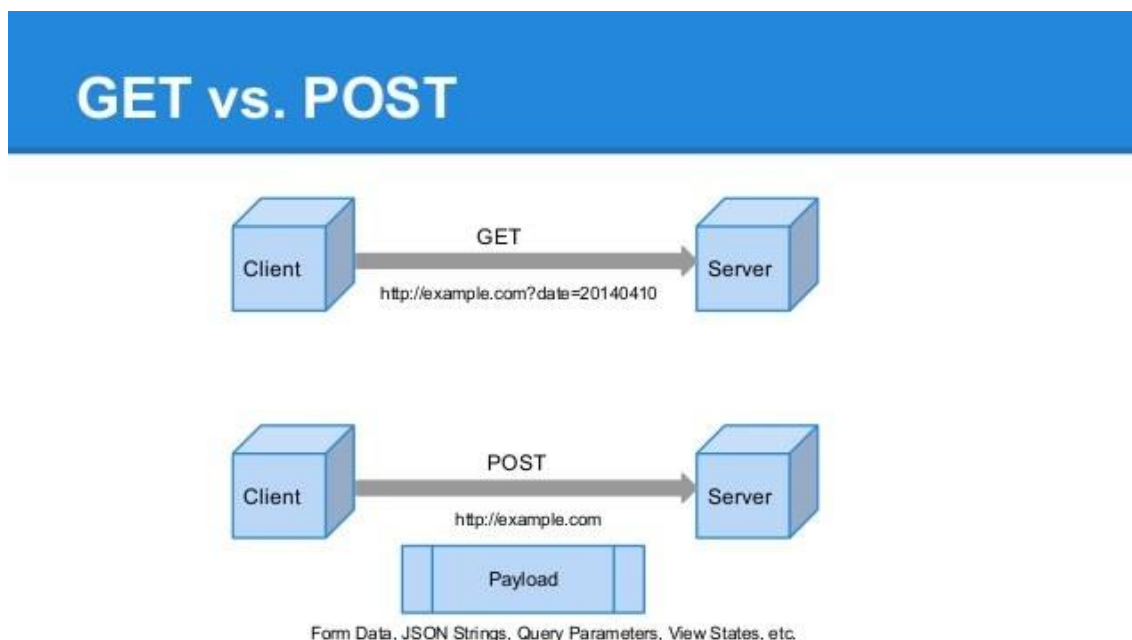
POST

POST es un verbo o método que es utilizado en el protocolo HTTP para poder proporcionar información al servidor desde un cliente.

A diferencia del método GET, en POST la información es enviada a través de las cabeceras de los paquetes o, así también, dentro del cuerpo del mensaje, haciendo de esta manera que la información sea invisible para el usuario. Es sin dudas un método recomendado para manejar gran cantidad de datos, como por ejemplo un formulario que sea enviado desde un cliente, como así también, para datos que necesiten mayor seguridad, como el caso de las contraseñas.

Entre las principales ventajas de POST se encuentra la simplicidad de las URL, donde no es necesario el envío de parámetros en ella (como es el caso de GET); por otro lado, es importante saber también, que a diferencia de las solicitudes GET, las solicitudes POST no son guardadas o almacenadas en caché. Es un método pensado para ser utilizado principalmente cuando es necesario brindar información que producirá cambios en el servidor, como por ejemplo, el alta de un nuevo registro en una base de datos.

Una pequeña [comparativa](#) entre GET y POST:



Otros métodos

GET y POST no son los únicos métodos con los que cuenta el Protocolo HTTP, sin embargo, son dos de los más utilizados. Fuera de ello, existen otros métodos que, dependiendo de la situación, cumplen con una determinada función particular.

Algunos de ellos [los citamos](#) a continuación:

Método	Funcionalidad
GET	El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
POST	El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
HEAD	El método HEAD pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
PUT	El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
DELETE	El método DELETE borra un recurso en específico.
PATCH	El método PATCH es utilizado para aplicar modificaciones parciales a un recurso.

JSP (Java Server Pages)

Java Server Pages, o mejor conocida por sus siglas JSP, es una tecnología cuya principal característica es la de poder incluir código Java en páginas o aplicaciones Web del lado del cliente en conjunto con etiquetas HTML y CSS. Si bien Java es un lenguaje pensado principalmente para ser ejecutado del lado del backend o del servidor, JSP permite hacer una excepción a esto e incluir de cierta manera código Java del lado del cliente.

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. Te mostramos un ejemplo:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Esto es una página JSP</title>
  </head>
  <body>
    <h1>Hola Mundo desde HTML!</h1>
    <% String hola = "Hola mundo desde Java"; %>
    <%= hola %>
  </body>
</html>
```

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](#).

En el ejemplo de la ilustración anterior se pueden observar dos tipos de “Hola mundo”, 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 mundo 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/percibido 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> .

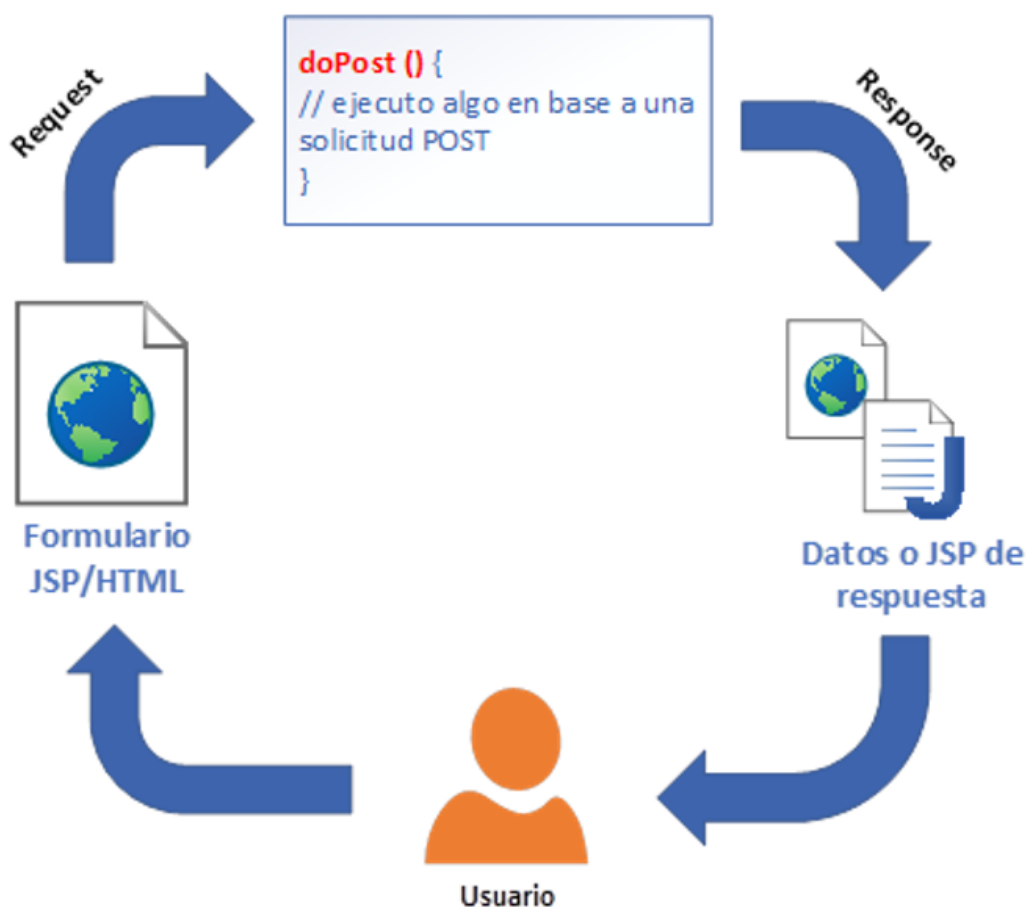
La principal ventaja que tiene JSP frente a otros tipos de implementaciones, es el hecho de que Java es un lenguaje que va más allá del mundo Web. Esto hace que JSP tenga las características de portabilidad (ejecución en diferentes sistemas operativos) que posee Java.

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).

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. Un ejemplo con el método POST puede visualizarse a continuación:



¿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.

Un ejemplo de un servlet transformándose en JSP:

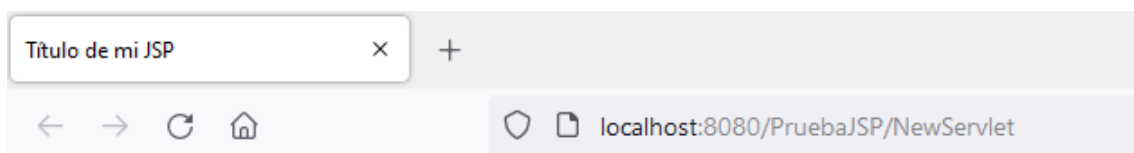
```
@WebServlet(urlPatterns = {"/NewServlet"})
public class NewServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Titulo de mi JSP</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1> Esto es un nuevo servlet en: " + request.getContextPath() + "</h1>");
            out.println("</body>");
            out.println("</html>");

        }
    }
}
```

El ejemplo a nivel código, se vería en el navegador tal y como se muestra a continuación:



Esto es un nuevo servlet en: /PruebaJSP

En proyectos más complejos, lo recomendable es combinar ambas tecnologías: los servlets para el procesamiento de información y los JSP para presentar los datos

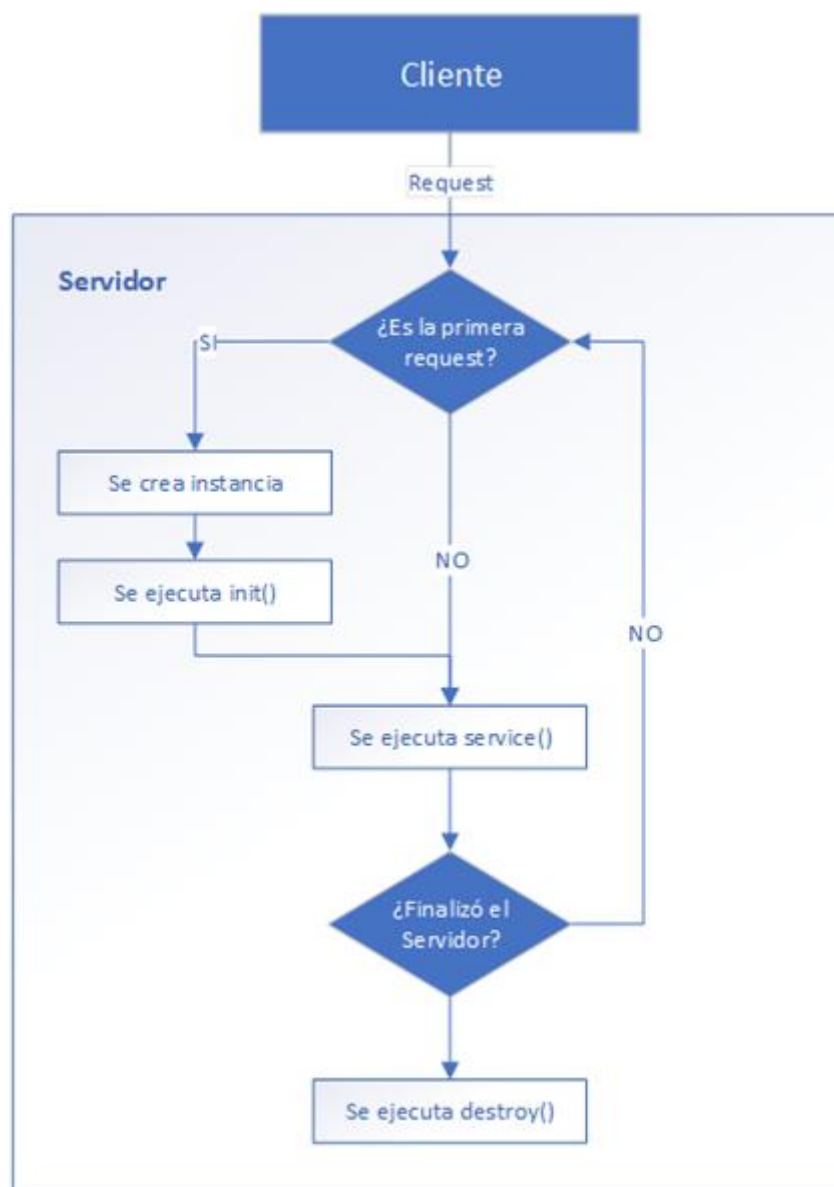
al cliente. Esta última opción es la que veremos en los ejemplos que se desarrollarán.

Ciclo de vida de un Servlet

Todo servlet tiene un ciclo de vida dentro de una aplicación web. Cada etapa del ciclo de vida dependerá estrechamente de las operaciones que esté haciendo el servidor web donde se encuentre alojado y se puede simplificar en 3 grandes pasos o momentos:

- **Inicio del servidor:** Al comenzar la ejecución de un servidor se producen dos acciones en un servlet:
 - o Se crea una instancia del Servlet
 - o Se inicia el Servlet mediante el método **init ()**
- **Llegada de una request o solicitud:**
 - o Se hace llamado al método **service()**, el cual implementa un nuevo thread (hilo) para atender a dicha solicitud.
- **Cierre o apagado del servidor:**
 - o Se hace llamado al método **destroy()**, el cual se encarga de destruir el servlet.

Un resumen de este ciclo de vida puede visualizarse en el siguiente diagrama de flujo:



Métodos de Servlets

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

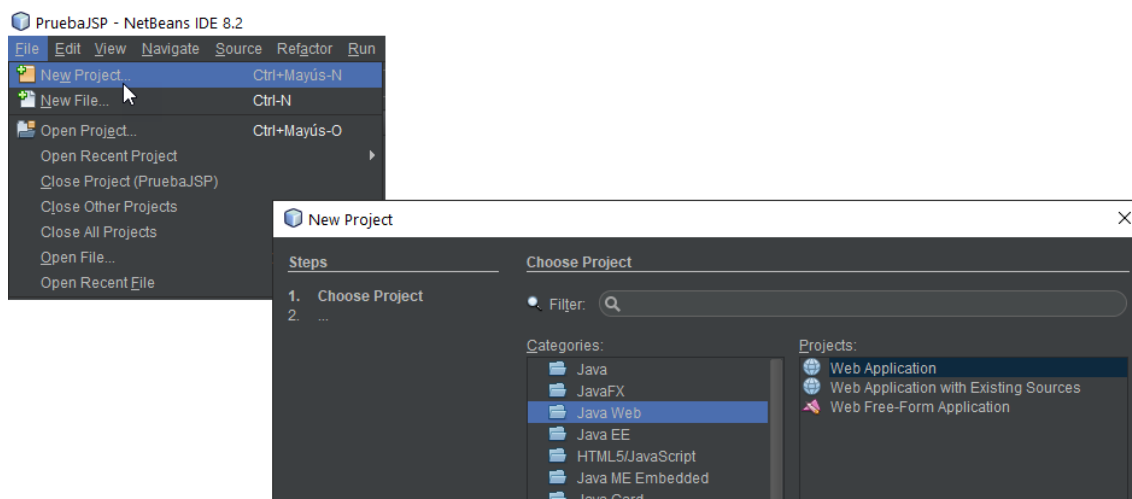
JSP. Su principal función es la de obtener datos desde el cliente para generar cambios en el servidor.

Creación de un Servlet

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

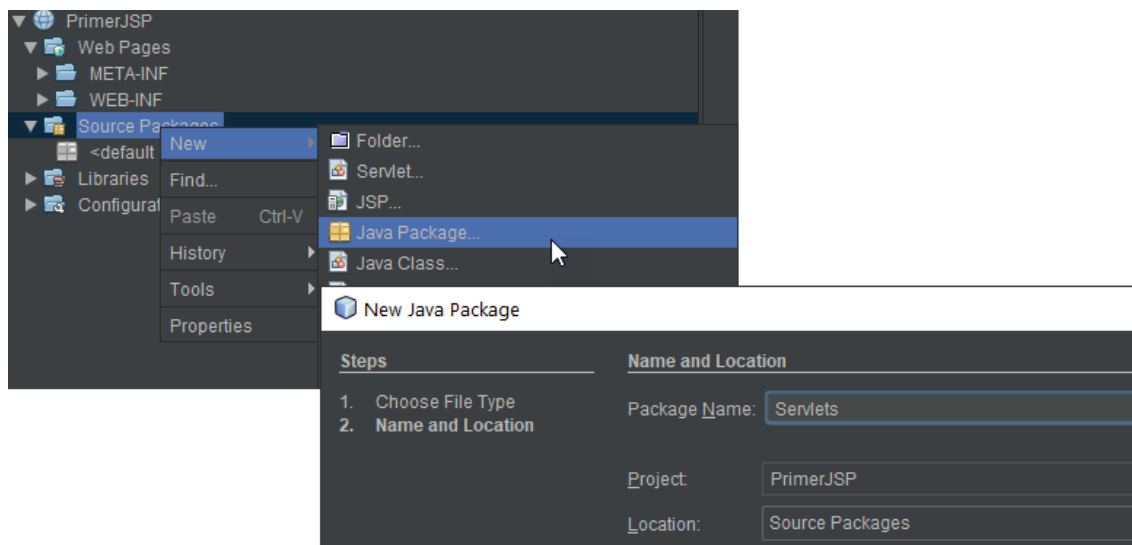
PASO 1

Crear un nuevo proyecto Java Web:



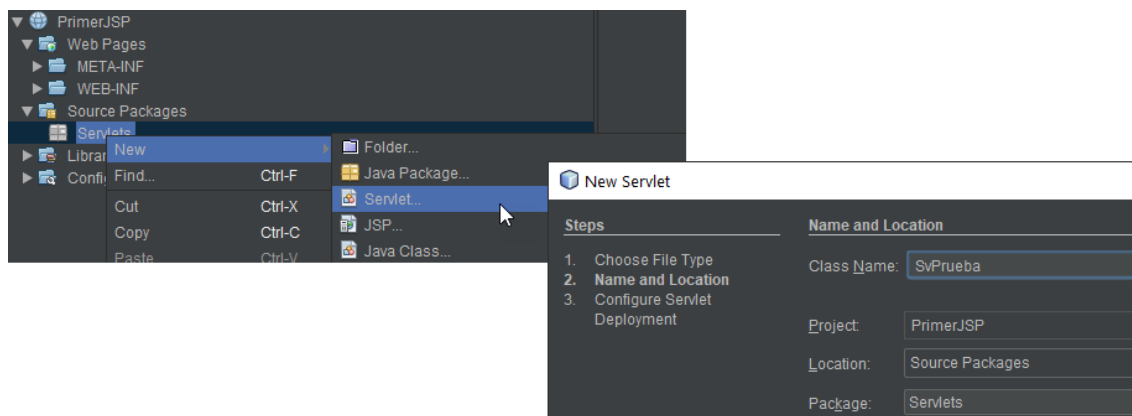
PASO 2

Crear un nuevo paquete para el guardado de los servlets



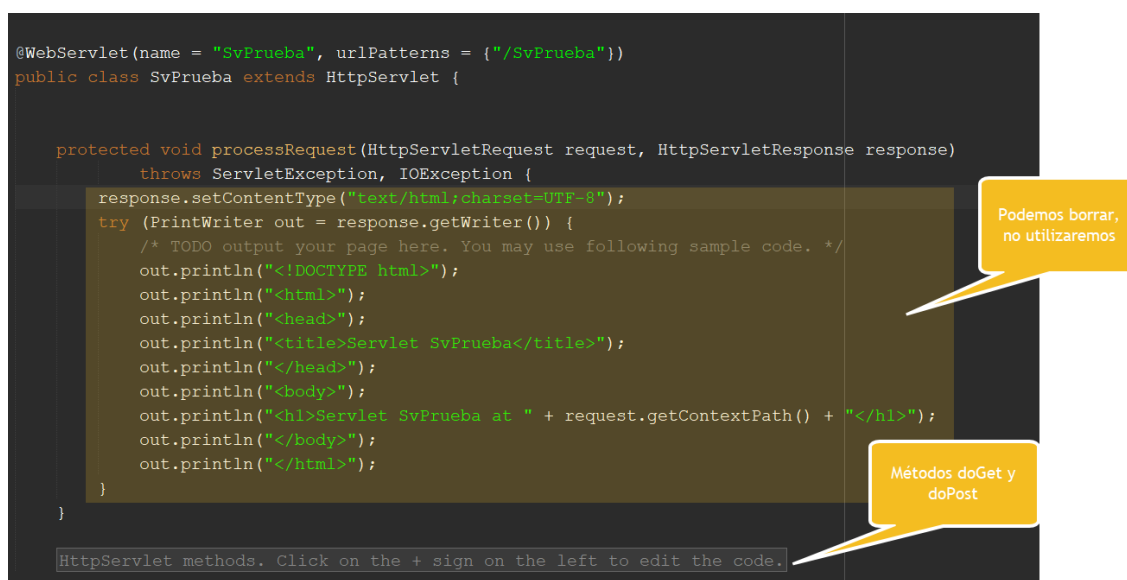
PASO 3

Click derecho sobre el nuevo paquete creado, luego en new y elegimos Java 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. Como utilizaremos los JSP y servlets de forma separada, este apartado podemos eliminarlo:



Borrado de apartado JSP en el servlet

```
@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

Interacción entre un JSP y un Servlet

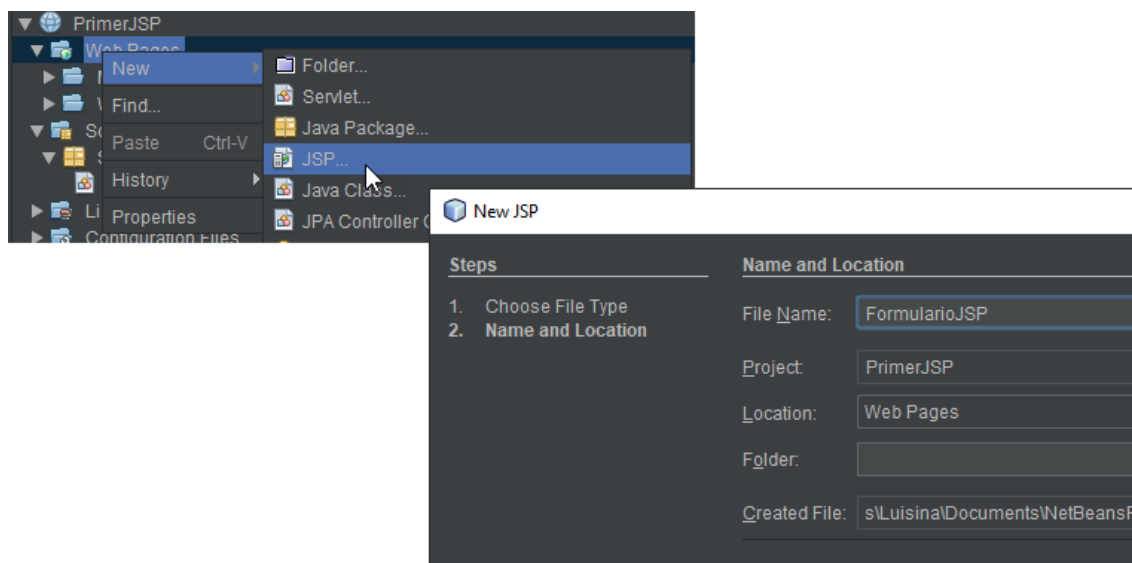
Como se mencionó anteriormente, un JSP puede ejecutarse por separado y llamar a un servlet, como así también, ejecutarse por si mismo dentro de un servlet. Cuando trabajamos con un modelo de capas, donde se ve a un JSP como una entidad separada del servlet (por más que trabajen en conjunto), tenemos 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 (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:

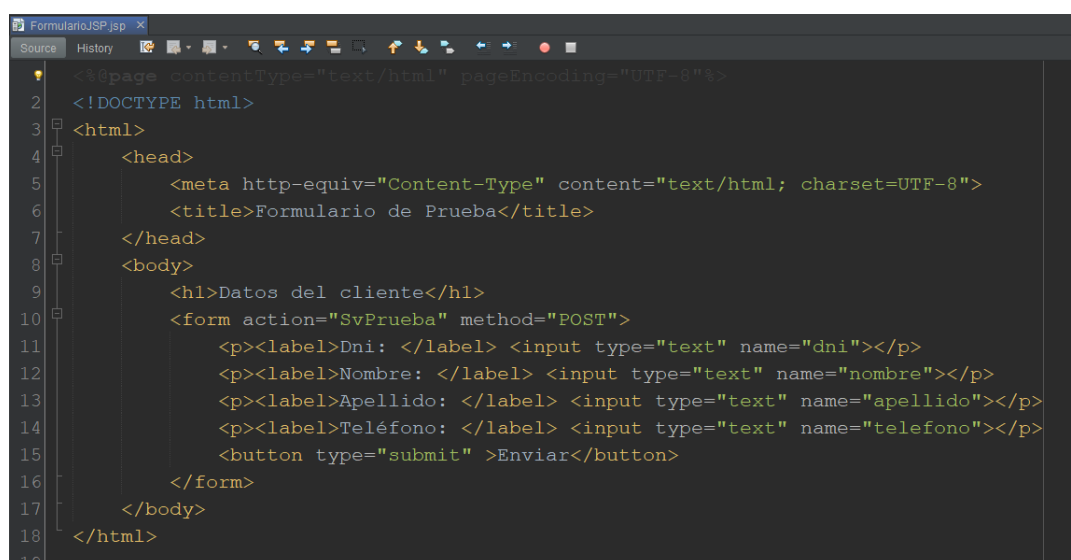


PASO 2

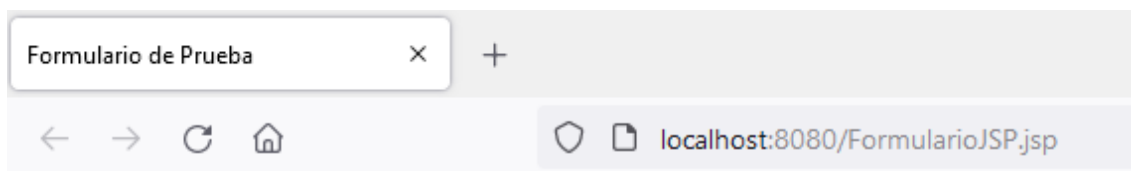
Una vez creado el JSP, armaremos en él un pequeño formulario HTML para proporcionar los datos desde el lado del cliente.

En la ilustración 18 se pueden observar las etiquetas HTML dentro del JSP, donde en el formulario, en el apartado action, se hace referencia al servlet que creamos anteriormente "SvPrueba" 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.

En la siguiente ilustración se especifica cómo se verá en el navegador el formulario dentro del JSP una vez que sea cargado:



HTML dentro del JSP para el formulario de carga de datos



Datos del cliente

Dni:

Nombre:

Apellido:

Teléfono:

Resultado en 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 a continuación:

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

    //traemos los datos enviados en la request
    //los guardamos en variables auxiliares
    //el nombre en getParameter debe ser el mismo que en el input del form
    String dni = request.getParameter("dni");
    String nombre = request.getParameter("nombre");
    String apellido = request.getParameter("apellido");
    String telefono = request.getParameter("telefono");

}
```

Una vez recibidos los datos, podemos pasarlos a la lógica de negocio para realizar operaciones, o guardarlos en una base de datos, o cualquier función que queramos darles

Ejemplo de método doPost()

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.

Traer datos desde un Servlet a un JSP (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 muestren 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 clientes de la lista además del alta de un nuevo cliente:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Formulario de Prueba</title>
</head>
<body>
  <h1>Datos del cliente</h1>
  <form action="SvPrueba" method="POST">
    <p><label>Dni: </label> <input type="text" name="dni"></p>
    <p><label>Nombre: </label> <input type="text" name="nombre"></p>
    <p><label>Apellido: </label> <input type="text" name="apellido"></p>
    <p><label>Teléfono: </label> <input type="text" name="telefono"></p>
    <button type="submit" >Enviar</button>
  </form>

  <h1>Ver lista de clientes</h1>
  <p>Si desea ver todos los clientes haga click en el botón mostrar clientes</p>
  <form action="SvPrueba" method="GET">
    <button type="submit" >Mostrar Clientes</button>
  </form>
</body>
</html>
```

Agregado en JSP para solicitar clientes al Servlet

Formulario de Prueba
+

localhost:8080/FormularioJSP.jsp

Datos del cliente

Dni:

Nombre:

Apellido:

Teléfono:

Ver lista de clientes

Si desea ver todos los clientes haga click en el botón mostrar clientes

Agregado JSP visto desde el navegador

PASO 2

En el servlet, se debe configurar el método `doGet()` para recibir la request, preparar la lista de clientes 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:

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

    //simulamos una lista de clientes
    //esto ya podría venir desde una base de datos
    List<Cliente> listaClientes = new ArrayList<> ();
    listaClientes.add(new Cliente("12345678", "Luisina", "de Paula", "444222357"));
    listaClientes.add(new Cliente("39887451", "Avril", "Lavigne", "777123575"));
    listaClientes.add(new Cliente("36442386", "Gianluigi", "Guidicci", "987441220"));

    //seteamos la lista de clientes como un parámetro
    //para poder utilizar en cualquier JSP
    //para ello traemos la sesión de la request
    HttpSession misession = request.getSession();
    misession.setAttribute("listaClientes", listaClientes);

    //redireccionamos a otro JSP
    response.sendRedirect("MostrarJSP.jsp");
}
```

Configuración `doGet` + redirección a otro JSP

PASO 3

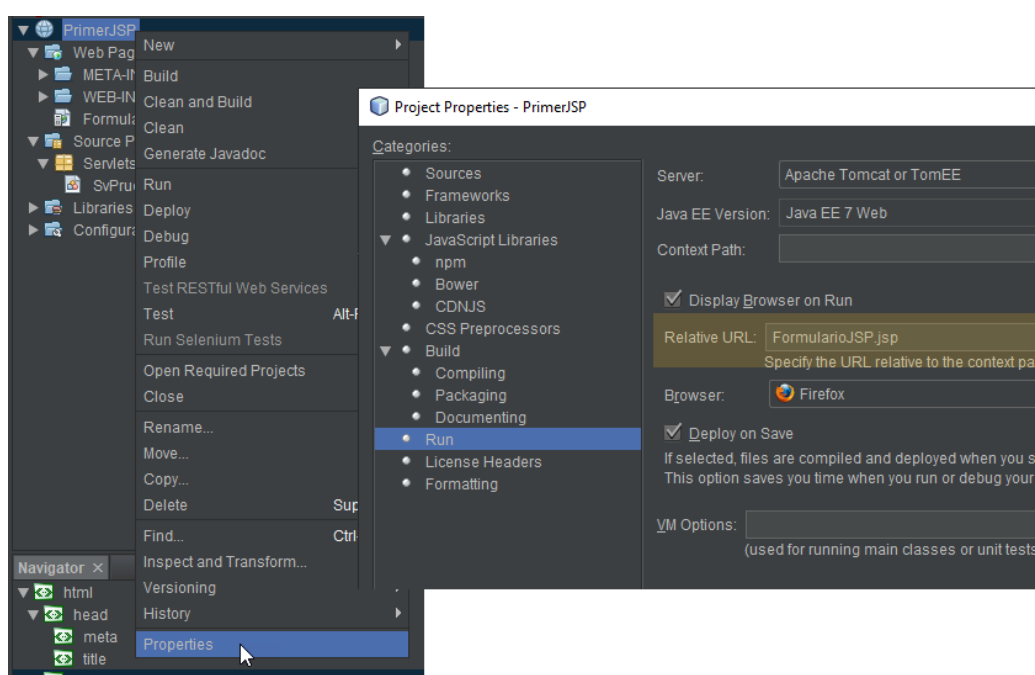
Creamos el nuevo JSP “MostrarJSP”, para recibir el parámetro enviado desde el servlet y mostrarlo gráficamente al usuario:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Clientes</title>
</head>
<body>
  <h1>Lista de Clientes</h1>
  <!-- utilizaremos código Java para traer y recorrer la lista-->
  <%
    List<Cliente> listaClientes = (List) request.getSession().getAttribute("listaClientes");
    int cont=1;
    for (Cliente cli : listaClientes) { %>
      <p><b>Cliente N°<%=cont%></b></p>
      <p>Dni: <%=cli.getDni() %></p>
      <p>Nombre: <%=cli.getNombre() %></p>
      <p>Apellido: <%=cli.getApellido() %></p>
      <p>Teléfono: <%=cli.getTelefono() %></p>
      <!-- Incremento mi contador,
      para mostrar correctamente cada num de cliente-->
      <% cont= cont+1;%>
    } %>
  </body>
</html>
```

Detalles en JSP "MostrarJSP" para visualizar los datos de los clientes cargados

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 Clientes” (Ilustración 22), deberíamos poder visualizar toda la lista de clientes en un nuevo JSP, tal y como se puede ver a continuación:



← → ↻ 🏠 localhost:8080/MostrarJSP.jsp

Lista de Clientes

Cliente N°1
Dni: 12345678
Nombre: Luisina
Apellido: de Paula
Teléfono: 444222357

Cliente N°2
Dni: 39887451
Nombre: Avril
Apellido: Lavigne
Teléfono: 777123575

Cliente N°3
Dni: 36442386
Nombre: Gianluigi
Apellido: Guidicci
Teléfono: 987441220

Resultado gráfico de los datos de clientes traídos a “MostrarJSP” desde el servlet

Sockets

Los sockets son un mecanismo que nos permite establecer un enlace entre dos programas que se ejecutan independientes el uno del otro (generalmente un programa cliente y un programa servidor). Java por medio de la librería `java.net` provee dos clases: **Socket** para implementar la conexión desde el lado del cliente y **ServerSocket** que permite manipular la conexión desde el lado del servidor.

La principal idea de la utilización de Sockets es el hecho de poder trabajar con diferentes computadoras o ejecuciones de un mismo programa cliente que se pueda conectar a un mismo servidor. Un claro ejemplo de un sistema que necesita de sockets puede ser un sistema de gestión de turnos, donde por ejemplo se tienen varios “Box”, donde cada persona encargada de atenderlos, al enviar una solicitud al servidor, recibe un nuevo número de cliente para atender consecutivo al de otros boxes. Un ejemplo del funcionamiento de una conexión socket puede verse en la siguiente Ilustración:



Ejemplo de funcionamiento de una conexión socket entre un cliente y un servidor

A continuación, veremos un ejemplo paso a paso de la utilización de sockets:

Implementación de Sockets

Para implementar sockets, es necesario que existan por lo menos dos aplicaciones que se comuniquen entre sí, una cliente y una servidor. Para ver un ejemplo de esto se puede implementar una simulación en donde, de igual manera, será necesario crear dos aplicaciones, una cliente (que se encargará de conectarse al

servidor) y otra servidor (que se encargará de esperar las peticiones por parte de los clientes).

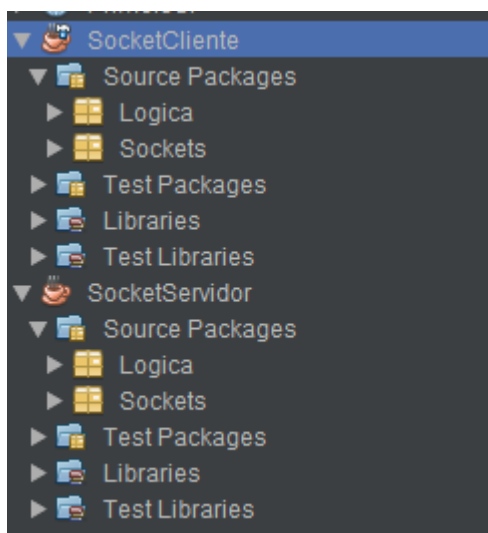
En ambos proyectos vamos a tener tres clases:

- **ConexionSockets:** Manejará los datos de conexión, como ser número de puerto, host, etc.
- **Cliente:** Manejará las conexiones y mensajes que enviará el cliente
- **Servidor:** Manejará la recepción de conexiones y mensajes del servidor.

Sin embargo, ambos proyectos tendrán un método Main diferente. Cada una de estas configuraciones las veremos paso a paso y en detalle a continuación.

PASO 1

En primer lugar, crearemos ambos proyectos. A uno lo llamaremos SocketCliente y al otro SocketServidor. En ambos crearemos un paquete lógica (donde irá el método main y cualquier lógica de negocio) y uno llamado Sockets, donde irán los correspondientes sockets que sean creados. Un ejemplo de esta organización podemos verla a continuación:



Ejemplo de proyectos creados

PASO 2

En ambos proyectos crearemos la clase ConexionSockets. En ella especificaremos variables para manejar el número de puerto, el host (dirección), los mensajes del servidor, las salidas tanto del cliente como del servidor, como así también la

creación de ambos sockets. La declaración de estas variables puede verse a continuación:

```
public class ConexionSockets {  
  
    private final int puerto = 8080; //Puerto para conectar  
    private final String host = "localhost"; //dirección ip para la conexión  
    protected String mensajeServidor; //variable para msjes recibidos en el servidor  
    protected ServerSocket socketServidor; //Socket del servidor  
    protected Socket socketCliente; //Socket del cliente  
    protected DataOutputStream salidaServidor, salidaCliente; //variables de salida
```

Declaración de variables en la clase ConexionSockets

Una vez hecho esto, crearemos el constructor de la clase, el cual recibirá como parámetro un tipo de conexión (si es del cliente o del servidor) y mediante una estructura if, controlará de qué tipo de conexión se trata, para a partir de esto, administrar la conexión del servidor o del cliente según se haya solicitado. Veamos un ejemplo de esto:

```
public ConexionSockets(String tipo) throws IOException //Constructor  
{  
    if(tipo.equalsIgnoreCase("servidor"))  
    {  
        //Creamos socket para el servidor  
        //se usa el puerto que configuramos anteriormente  
        socketServidor = new ServerSocket(puerto);  
        //creamos el socket para el cliente  
        socketCliente = new Socket();  
    }  
    else  
    {  
        //Creamos socket para el cliente  
        //le pasamos como parámetro el host y el puerto  
        socketCliente = new Socket(host, puerto);  
    }  
}
```

Constructor ConexionSockets y masnejo de conexiones

De esta manera, la clase completa nos debería de quedar de este modo:

```
package Sockets;

import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ConexionSockets {

    private final int puerto = 8080; //Puerto para conectar
    private final String host = "localhost"; //dirección ip para la conexión
    protected String mensajeServidor; //variable para msjes recibidos en el servidor
    protected ServerSocket socketServidor; //Socket del servidor
    protected Socket socketCliente; //Socket del cliente
    protected DataOutputStream salidaServidor, salidaCliente; //variables de salida

    public ConexionSockets(String tipo) throws IOException //Constructor
    {
        if(tipo.equalsIgnoreCase("servidor"))
        {
            //Creamos socket para el servidor
            //se usa el puerto que configuramos anteriormente
            socketServidor = new ServerSocket(puerto);
            //creamos el socket para el cliente
            socketCliente = new Socket();
        }
        else
        {
            //Creamos socket para el cliente
            //le pasamos como parámetro el host y el puerto
            socketCliente = new Socket(host, puerto);
        }
    }
}
```

Clase ConexionSockets completa

PASO 3

En ambos proyectos crearemos la clase Servidor, la cual extenderá (heredará) de la clase ConexionSocket que creamos anteriormente (para poder hacer uso de su constructor). Esto podemos visualizarlo en la siguiente Ilustración:

```
//Servidor va a heredar de ConexionSockets
//para poder usar sus métodos
public class Servidor extends ConexionSockets{

    //Implementa el Constructor de Conexión Sockets
    //pasamos el tipo para que Conexión lo tome e inicialice
    public Servidor() throws IOException{
        super("servidor");
    }
}
```

Constructor clase Servidor

Una vez hecho esto, especificaremos un método para “arrancar” o “iniciar” la ejecución del servidor. En este método mostraremos una serie de mensajes al usuario para que conozca en qué situación se encuentra el servidor como así también, especificaremos el código necesario para aguardar por una posible conexión del cliente. En caso de que el cliente se conecte, mostraremos un mensaje y luego de terminadas las acciones que este desee realizar, finalizaremos la conexión. Veamos un ejemplo:

```
//creamos un método para inicializar el servidor
public void startServer(){
    try{
        //el método accept inicializa el socket
        //queda a la espera de solicitudes
        System.out.println("Esperando por una Conexión...");
        socketCliente = socketServidor.accept();

        //Se obtiene el flujo de salida del cliente
        System.out.println("Cliente en línea");
        salidaCliente = new DataOutputStream(socketCliente.getOutputStream());

        //Enviamos un mensaje al cliente
        salidaCliente.writeUTF("Petición recibida y aceptada");

        //Se obtiene el flujo entrante desde el cliente
        BufferedReader entrada = new BufferedReader(new InputStreamReader(socketCliente.getInputStream()));
    }
}
```

Inicio del método startServer y preparación del servidor para recibir mensajes


```
//Mientras haya mensajes del cliente
while((mensajeServidor = entrada.readLine()) != null){
    //Mostramos los mensajes recibidos
    System.out.println(mensajeServidor);
}

//Una vez que termino de leer los mensajes
//Finalizamos la conexión
System.out.println("Fin de la conexión");
socketServidor.close(); //Se finaliza la conexión con el cliente
}
catch (IOException e){
    System.out.println(e.getMessage());
}
```

Recepción de cada uno de los mensajes del cliente mientras existen y finalización de conexión

PASO 4

En ambos proyectos crearemos la clase Cliente, quien al igual que Servidor, extenderá (heredará) de la clase ConexionSocket que creamos anteriormente (para poder hacer uso de su constructor), tal y como puede verse a continuación:

```
public class Cliente extends ConexionSockets{

    //Constructor
    //pasamos el tipo para que Conexión lo tome e inicialice
    public Cliente() throws IOException{
        super("cliente");
    }
}
```

Constructor clase Cliente

A partir de esto, vamos a crear un método llamado startClient() para comenzar las peticiones por parte del cliente y así lograr que se comunique con el servidor. Dentro del método vamos a simular 3 peticiones de mensajes que serán enviados por el cliente. A partir de esto, el servidor se va a encargar de recibir estas 3 peticiones y mostrarlas como resultado. Una vez finalizado este proceso, cerraremos la conexión del socket del cliente mediante close(), tal y como fue hecho con el servidor. Veamos la clase Cliente completa con el método startClient:

```
package Sockets;
import java.io.DataOutputStream;
import java.io.IOException;

public class Cliente extends ConexionSockets{

    //Constructor
    //pasamos el tipo para que Conexión lo tome e inicialice
    public Cliente() throws IOException{
        super("cliente");
    }

    //Iniciamos el cliente
    public void startClient(){
        try{
            //Flujo de datos hacia el servidor
            salidaServidor = new DataOutputStream(socketCliente.getOutputStream());

            //Enviamos 3 mensajes para probar
            for (int i = 0; i < 3; i++){
                //Se escribe en el servidor usando su flujo de datos
                salidaServidor.writeUTF("Este es el mensaje número " + (i+1) + "\n");
            }
            //Finalizamos la conexión
            socketCliente.close();
        }
        catch (IOException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Clase cliente completa

PASO 5

Una vez creadas las clases en ambos proyectos dentro del paquete “Sockets”, deberemos setear ambos métodos Main para lograr así simular una conexión entre el cliente y el servidor.

En el main que corresponda al proyecto SocketServidor, será necesario inicializar una nueva instancia de Servidor. A partir de esto, es posible realizar el llamado al método startServer(), el cual creamos anteriormente dentro de la clase servidor. Este llamado permitirá que el servidor comience a ejecutarse en el host y puerto que hayan sido especificados y quede a la escucha de posibles mensajes por parte del cliente. Veamos un ejemplo:

```
package Logica;
import Sockets.Servidor;
import java.io.IOException;

public class SocketServidor {

    public static void main(String[] args) throws IOException {

        //creamos un nuevo servidor y lo iniciamos
        Servidor servi = new Servidor();
        System.out.println("Iniciando el servidor... \n");
        servi.startServer();

    }

}
```

Método Main en el proyecto SocketServidor

Por otro lado, en el método Main del proyecto Socket Cliente, será necesario inicializar una nueva instancia de Cliente. A partir de esto, será posible hacer uso del método stratClient() que comenzará a enviar peticiones desde el cliente al servidor. El código del método Main del cliente podemos observarlo a continuación:

```
package Logica;
import Sockets.Cliente;
import java.io.IOException;

public class SocketCliente {

    public static void main(String[] args) throws IOException {

        Cliente cli = new Cliente(); //Se crea el cliente

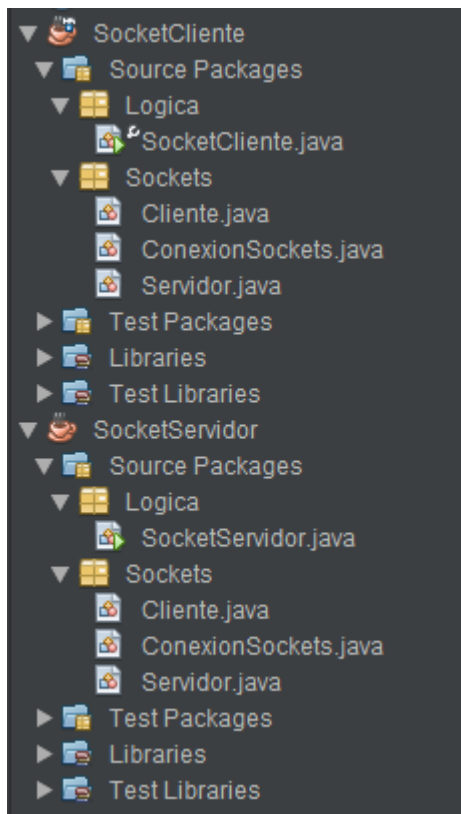
        System.out.println("Iniciando cliente\n");
        cli.startClient(); //Se inicia el cliente

    }

}
```

Método Main en el proyecto SocketCliente

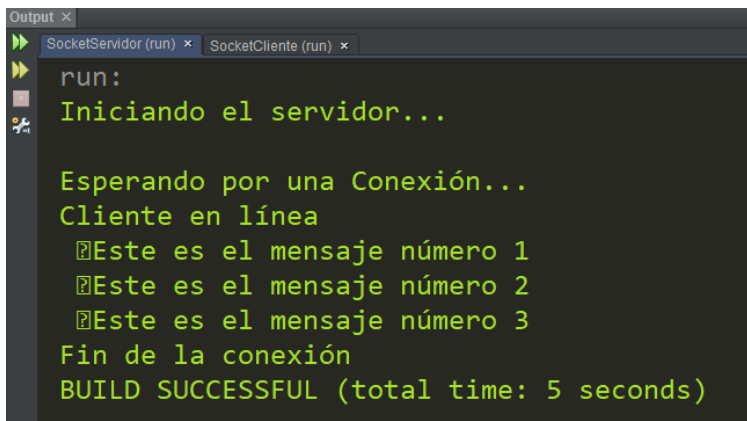
Una vez finalizados estos pasos, la estructura de nuestros proyectos debería quedar similar a la que se muestra en la siguiente Ilustración:



Ejemplo de estructura de proyectos Cliente y Servidor

PASO 6

Como último paso, para probar la conexión entre ambos sockets, habrá que ejecutar ambas aplicaciones. Es de vital importancia que en primer lugar se ejecute el servidor (como sucede en un ambiente real, donde el servidor debe existir para que el cliente se pueda comunicar), una vez levantado y que se encuentre en ejecución el servidor, podrá ser ejecutado el cliente para que empiece a enviar los mensajes. Un ejemplo del resultado de estas dos ejecuciones:



```
Output X
SocketServidor (run) x SocketCliente (run) x
run:
Iniciando el servidor...

Esperando por una Conexión...
Cliente en línea
  Este es el mensaje número 1
  Este es el mensaje número 2
  Este es el mensaje número 3
Fin de la conexión
BUILD SUCCESSFUL (total time: 5 seconds)
```

Salida de la consola a partir de la ejecución de ambas aplicaciones

En esta simulación se puede ver la interacción existente entre ambos Sockets, donde el servidor comienza su ejecución y aguarda las solicitudes del cliente. Esta situación en un ambiente real trata generalmente con sistemas diferentes ejecutándose en diferentes computadoras, donde existe una central o servidor que recibirá cada petición y se encargará de la atención. Si se toma como ejemplo el sistema de turnos que se mencionó anteriormente, el servidor de turnos se encontrará ejecutándose y con cada petición de cada socket cliente, aumentará en 1 el número de turno.