

TEMA 2: Java Web

Sumario

.....	1
Java Web Introducción.....	2
Servlets.....	10
JSP.....	18
Java Bean.....	28
MVC con JavaBeans.....	30
Content-type.....	32
Sintaxis de un tipo MIME.....	33
Responder desde Servlet con diferentes Content-type.....	35
Devolver json:.....	35
Devolver imagen:.....	36
WEB-INF.....	39
Subir y bajar ficheros.....	41
Como subir ficheros.....	43
tipos de codificaciones posibles con POST.....	44
Crear el directorio donde guardar los ficheros.....	45
Recoger los ficheros del request y guardarlos.....	45
Bajar Ficheros.....	46
Como listar ficheros de un directorio.....	46
Como descargar cualquier tipo de fichero.....	47

Java Web Introducción

El modelo de aplicación Java EE consiste en el uso del lenguaje Java en una máquina virtual Java como base, y un entorno controlado llamada “middle tier” que tiene acceso a todos los servicios empresariales. El middle tier suele ejecutarse en una máquina dedicada.

Por lo tanto es un modelo multinivel en el que la lógica de negocio y la presentación corren a cargo del desarrollador y los servicios estándar del sistema a cargo de la plataforma Java EE.

Para las aplicaciones empresariales usa una arquitectura multinivel distribuida donde la lógica de la aplicación se divide en varios componentes según su función y estos componentes residen en distintas máquinas dependiendo del nivel al que pertenezca el componente de la aplicación.

Las aplicaciones Java EE están formadas por componentes. Los componentes son piezas de software autocontenidas y funcionales en si mismas que se integran con la aplicación y se comunican con otros componentes.

Se pueden clasificar en:

- Aplicaciones cliente y applets, que corren en el cliente.
- Servlets, componentes JSP y JavaServer Faces que corren del lado del servidor.
- Componentes EJB (Enterprise Java Beans) que corren del lado del servidor.

Los clientes pueden ser:

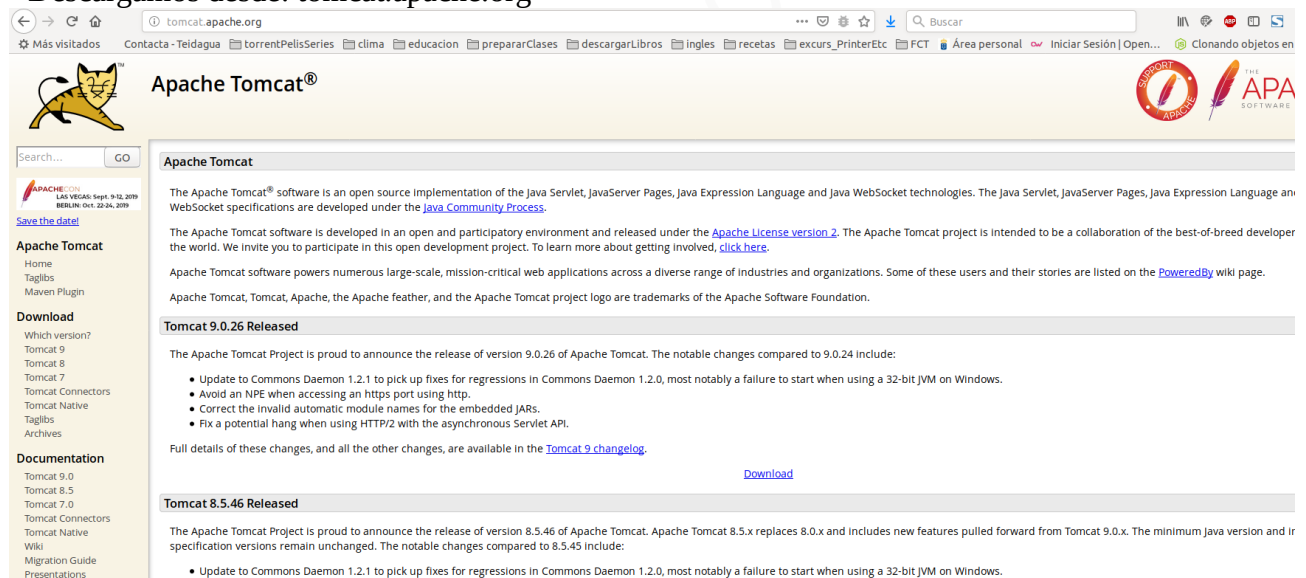
- Aplicaciones cliente: suelen ser aplicaciones gráficas o en línea de comandos que se comunican con la capa de negocio directamente o a través de conexiones HTTP.
- Clientes Web: tienen dos partes, páginas web dinámicas que contienen varios elementos que se generan en el lado del servidor y un navegador web que muestra las páginas generadas. Las tareas pesadas las ejecuta un servidor Java EE.

Los componentes web pueden ser servlets o páginas creadas con tecnologías JSP o JavaServer Faces.

- **Servlets**: son clases java que se ejecutan en el servidor, procesan peticiones y generan respuestas de forma dinámica.
- **Páginas JSP**: son ficheros de texto que se ejecutan como servlets pero cuyo contenido mezcla elementos estáticos con elementos interpretados.
- JavaServer Faces: es una tecnología que se apoya en servlets y JSP para proporcionar un conjunto de herramientas para crear interfaces de usuario para aplicaciones web.

Vamos a empezar preparando el equipo y el proyecto Netbeans así:

- Descargamos desde: tomcat.apache.org



The screenshot shows the Apache Tomcat website. The header includes the Apache Tomcat logo and navigation links. The main content area is titled "Apache Tomcat" and contains the following information:

- Apache Tomcat**: The Apache Tomcat® software is an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Java Servlet, JavaServer Pages, Java Expression Language and WebSocket specifications are developed under the [Java Community Process](#).
- Download**: Which version? Tomcat 9, Tomcat 8, Tomcat 7, Tomcat Connectors, Tomcat Native, Taglibs, Archives.
- Documentation**: Tomcat 9.0, Tomcat 8.5, Tomcat 7.0, Tomcat Connectors, Tomcat Native, Wiki, Migration Guide, Presentations.
- Tomcat 9.0.26 Released**: The Apache Tomcat Project is proud to announce the release of version 9.0.26 of Apache Tomcat. The notable changes compared to 9.0.24 include:
 - Update to Commons Daemon 1.2.1 to pick up fixes for regressions in Commons Daemon 1.2.0, most notably a failure to start when using a 32-bit JVM on Windows.
 - Avoid an NPE when accessing an https port using http.
 - Correct the invalid automatic module names for the embedded JARs.
 - Fix a potential hang when using HTTP/2 with the asynchronous Servlet API.Full details of these changes, and all the other changes, are available in the [Tomcat 9 changelog](#).
- Tomcat 8.5.46 Released**: The Apache Tomcat Project is proud to announce the release of version 8.5.46 of Apache Tomcat. Apache Tomcat 8.5.x replaces 8.0.x and includes new features pulled forward from Tomcat 9.0.x. The minimum Java version and its specification versions remain unchanged. The notable changes compared to 8.5.45 include:
 - Update to Commons Daemon 1.2.1 to pick up fixes for regressions in Commons Daemon 1.2.0, most notably a failure to start when using a 32-bit JVM on Windows.

En este caso tomaremos la versión tomcat 8

Tomcat 8 Software Downloads

Welcome to the Apache Tomcat® 8.x software download page. This page provides download links for obtaining the latest versions of Tomcat 8.x software, as well as links to the archives of older releases.

Users of Tomcat 8.0.x should be aware that it has reached [end of life](#). Users of Tomcat 8.0.x should upgrade to 8.5.x or later.

Note: End of life has been announced for 8.0.x only. 8.5.x is not affected by this announcement.

Quick Navigation

[KEYS](#) | [8.5.46](#) | [Browse](#) | [Archives](#)

Release Integrity

You **must verify** the integrity of the downloaded files. We provide OpenPGP signatures for every release file. This signature should be matched against the [KEYS](#) file which contains the OpenPGP keys of Tomcat's Release. We provide [SHA-512](#) checksums for every release file. After you download the file, you should calculate a checksum for your download, and make sure it is the same as ours.

Mirrors

You are currently using <http://mirrors.netix.net/apache/>. If you encounter a problem with this mirror, please select another mirror. If all mirrors are failing, there are *backup* mirrors (at the end of the mirrors list) that you can use.

Other mirrors:

8.5.46

Please see the [README](#) file for packaging information. It explains what every distribution contains.

Binary Distributions

- Core:
 - [zip \(pgp, sha512\)](#)
 - [tar.gz \(pgp, sha512\)](#)
 - [32-bit Windows zip \(pgp, sha512\)](#)
 - [64-bit Windows zip \(pgp, sha512\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)
- Full documentation:
 - [tar.gz \(pgp, sha512\)](#)
- Deployer:
 - [tar.gz \(pgp, sha512\)](#)

- Descomprimos en la carpeta del usuario: /home/usuario (al hacerlo nos generará una carpeta que dices tomcat con el número de versión)
(se supondrá que el usuario se llama: “usuario” reemplazar por “alumno” o cualquier nombre de usuario que corresponda)

Por simplicidad para instalaciones futuras será buena idea renombrar la carpeta a:
/home/usuario/tomcat

- Para evitar problemas de ejecución de algún fichero estableceremos permisos de ejecución a toda la carpeta bin;
chmod +x /home/usuario/tomcat/bin/*

- Es conveniente que nuestro tomcat esté en el PATH de usuario. Para ello nos vamos al final del fichero .profile (es un fichero oculto): /home/usuario/.profile

y escribimos al final del fichero (reemplazar la palabra usuario por la que corresponda)
export CATALINA_HOME=/home/usuario/tomcat
export PATH="\$PATH:\$CATALINA_HOME

- Para interactuar con el servicio debemos establecer un usuario y password Eso lo hacemos editando el fichero:
/home/usuario/tomcat/conf/tomcat-users.xml

al final de ese fichero agregaremos la información. En este caso se ha elegido por nombre de usuario: "admin" y por password: "1q2w3e4r"

```
<role rolename="manager-gui"/>
<role rolename="admin"/>
<user username="admin" password="1q2w3e4r" roles="manager-gui,admin"/>
```

- Ahora lo que resta es arrancar el servidor. De paso vemos cuál es el comando de detención:

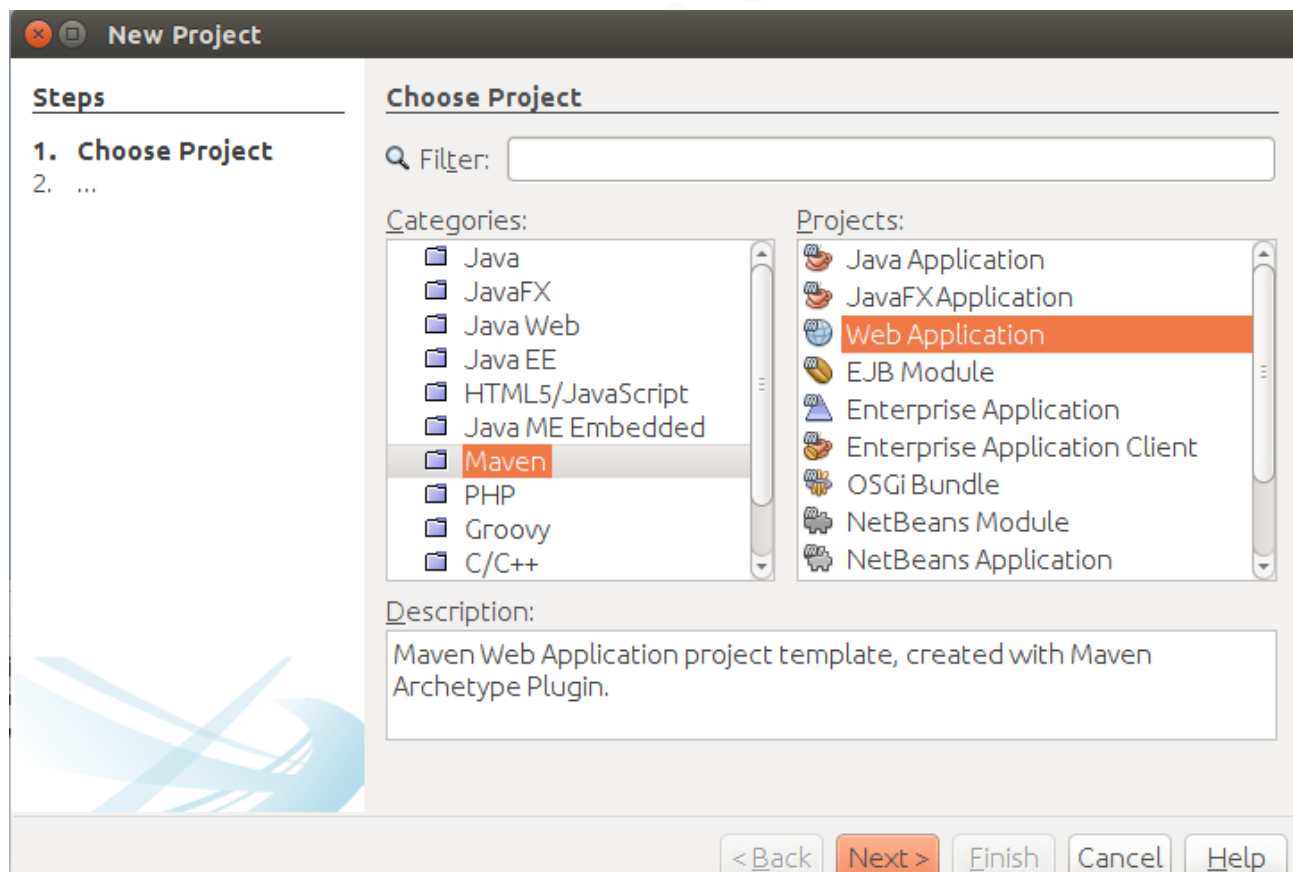
* Arrancar: /home/usuario/tomcat/bin/startup.sh

* Detener: /home/usuario/tomcat/bin/shutdown.sh

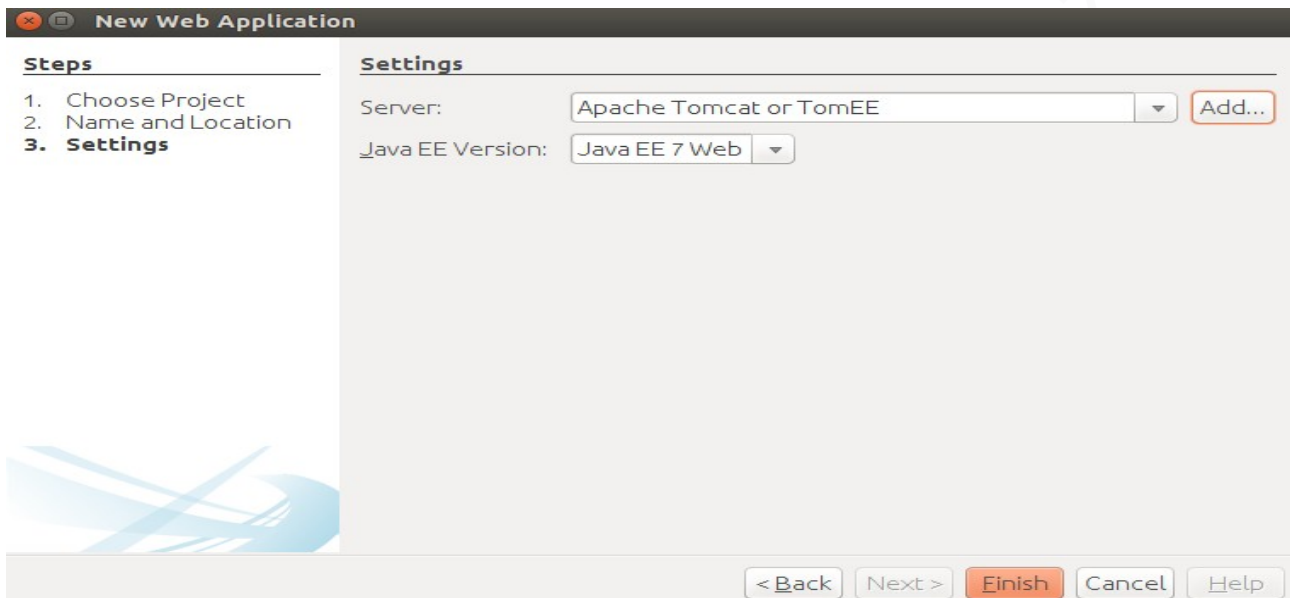
- Para acceder al tomcat abrimos el navegador y vamos a:

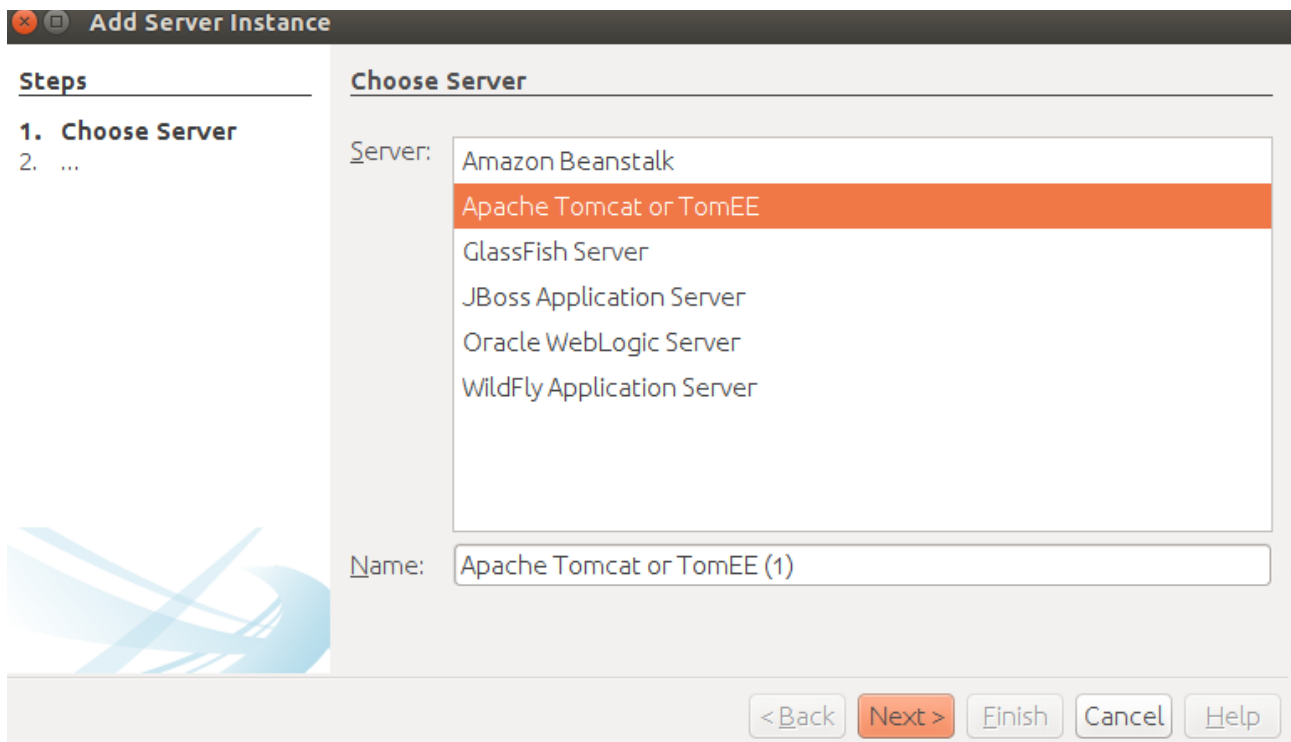
<http://localhost:8080>

- Ahora vamos a crear un proyecto maven netbeans:

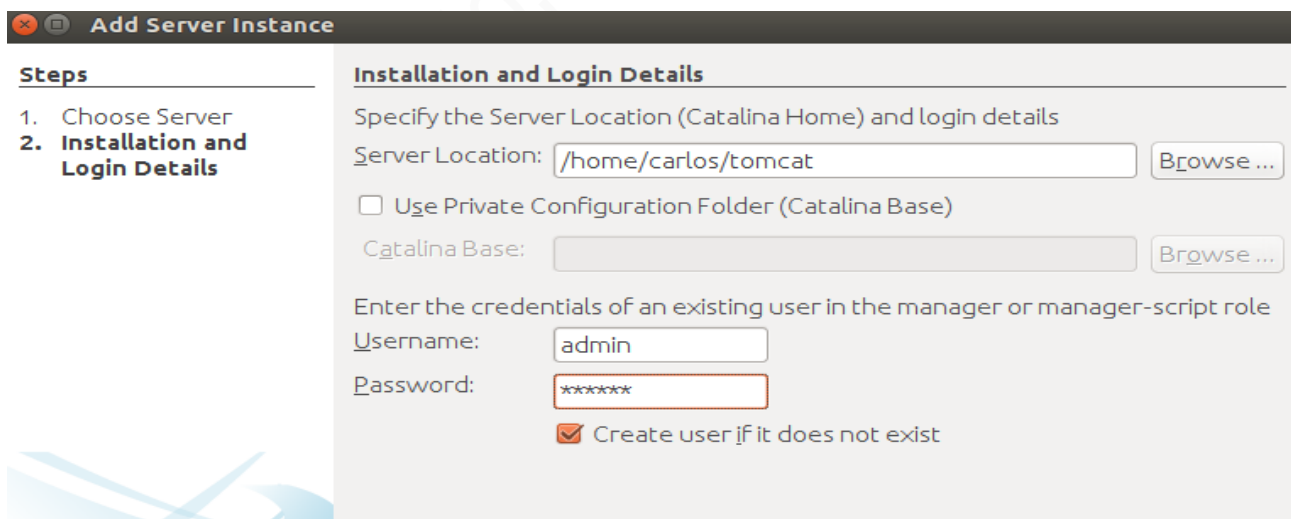


El wizard nos solicitará más adelante que establezcamos el Servidor web. Vamos a utilizar ahora tomcat que pondremos en nuestro home. Para ello seleccionamos tomcat y mediante el botón “Add” especificamos donde está la carpeta del tomcat

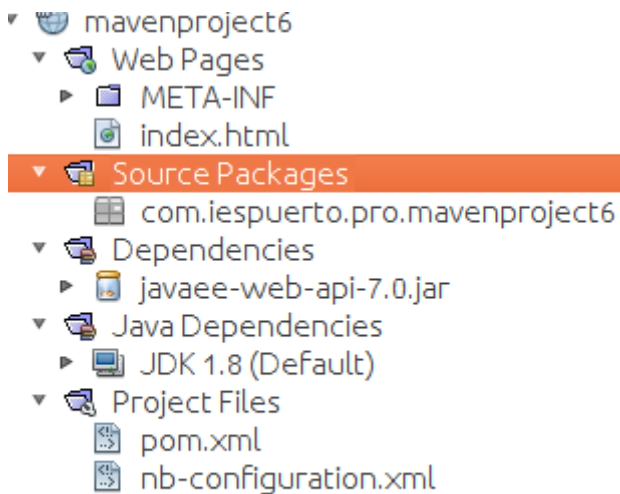




Observar que también le damos el usuario y la contraseña para acceder a tomcat



Veamos la posible estructura del proyecto creado:



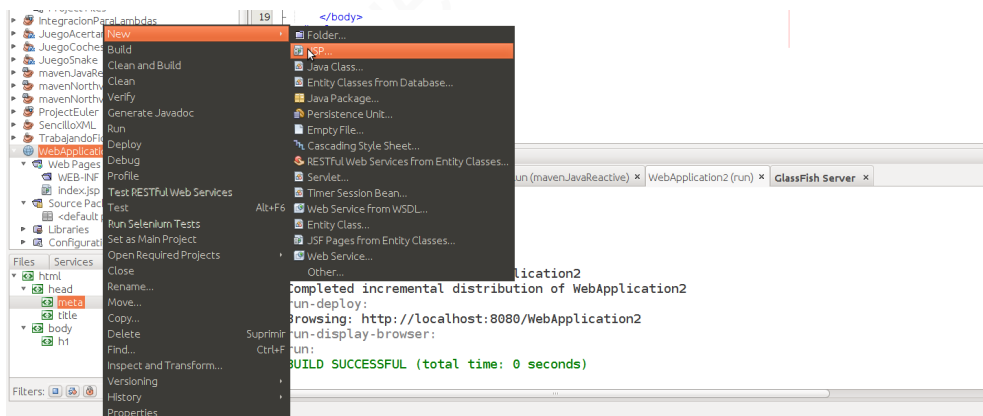
Vemos que después de crearnos el proyecto nos abre una página: index.html

Si lanzamos ahora mediante “run” observaremos que nos muestra la página web en el navegador

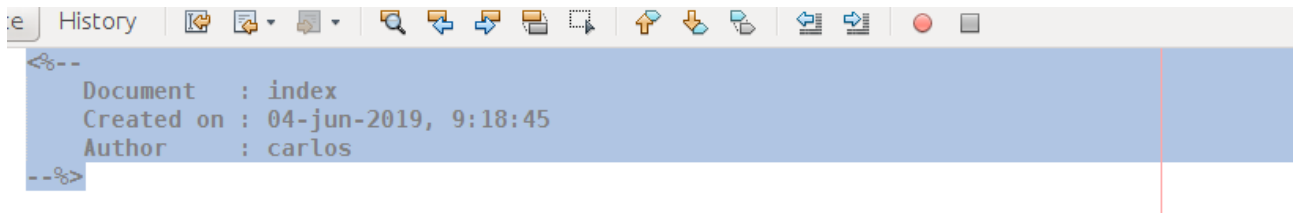
Ahora bien, esa página no es activa, es una página estática

Vamos a pasar a crear una página activa

Eliminamos el fichero: index.html y creamos un fichero: index.jsp
new-->jsp



En esta página lo primero que observamos en gris es la forma que tenemos en jsp de crear los comentarios (mediante: `<%-- --%>`)



La página que estamos visualizando es una página que por defecto escribiremos en HTML y cuando queramos que se interprete código Java lo pondremos dentro de: `<% %>`

Así el siguiente código nos devolverá una secuencia de números en la página web:

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Hello World!</h1>

    for (int idx = 0; idx < 10; idx++) {
        out.println("<p>" + idx + "</p>");
    }

  </body>
</html>

```

Podemos observar que el IDE nos colorea y distingue la parte que es código Java

también vemos como podemos desde el código Java que nos “escriba” en la página Web. Lo hacemos mediante el método: `out.println()`

hemos visto ya en ejecución un JSP veamos ahora la teoría:

Servlets

Es una clase en el lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor. Aunque los servlets pueden responder a cualquier tipo de solicitudes, estos son utilizados comúnmente para extender las aplicaciones alojadas por servidores web

Muy útiles para leer cabeceras de mensajes, datos de formularios, gestión de sesiones, procesar información, etc. Pero tediosos para generar todo el código HTML El mantenimiento del código HTML es difícil

Lo siguiente es una sección de los import de un servlet Java y la declaración de la clase java que soporta servlet:

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author carlos
 */
public class NewServlet extends HttpServlet {
```

Los servlets heredan de la clase **HttpServlet** y permiten gestionar elementos HTTP mediante las clases (se muestran en los import de arriba):

* **HttpServletRequest**: recibe la petición

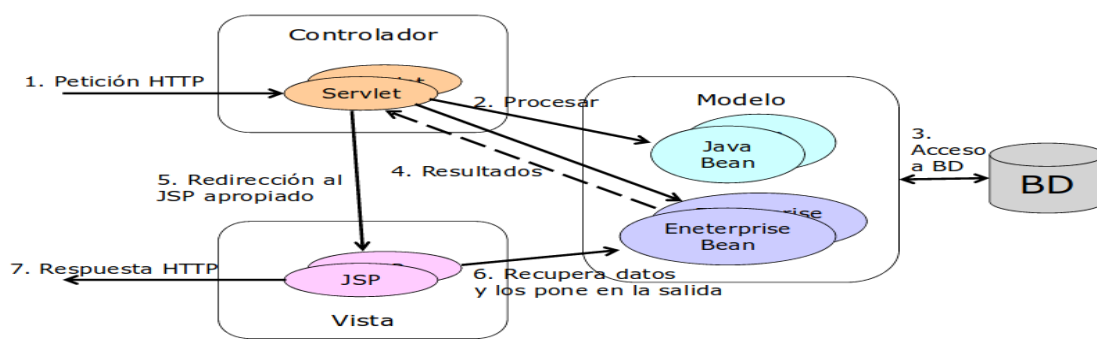
* **HttpServletResponse**: genera la respuesta.

* **HttpSession**: permite crear una sesión común a un conjunto de request (ámbito de sesión).

* **ServletContext**: gestiona la información común a todas las peticiones realizadas sobre la aplicación (ámbito de aplicación). Se obtiene a partir del método `getServletContext()` de la clase `HttpServlet`.

Los servlets en un modelo **MVC** está ubicado en la parte del controlador, ya que devuelve una respuesta (**Response**) (que podría ser una página jsp, un fichero, un video, audio,...) y debe interactuar con las peticiones (**Request**) de los usuarios

Arquitectura MVC en Java EE



Los servlets no tienen el método `main()` como los programas Java, sino que se invocan unos métodos cuando se reciben peticiones. A esta metodología se le llama ciclo de vida de un servlet y viene dado por tres métodos: `init`, `service`, `destroy`:

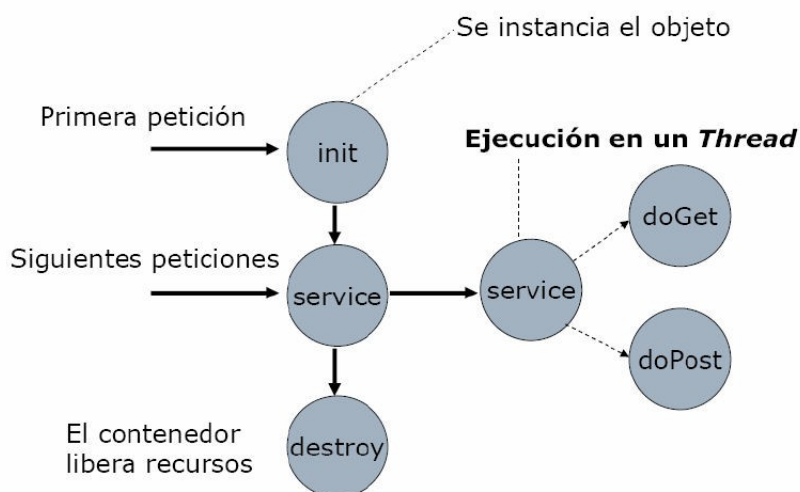
INICIALIZACIÓN: Una única llamada al método “`init`” por parte del servlet. Incluso se pueden recoger unos parámetros concretos con “`getInitParameter`” de “`ServletConfig`” iniciales y que operarán a lo largo de toda la vida del servlet.

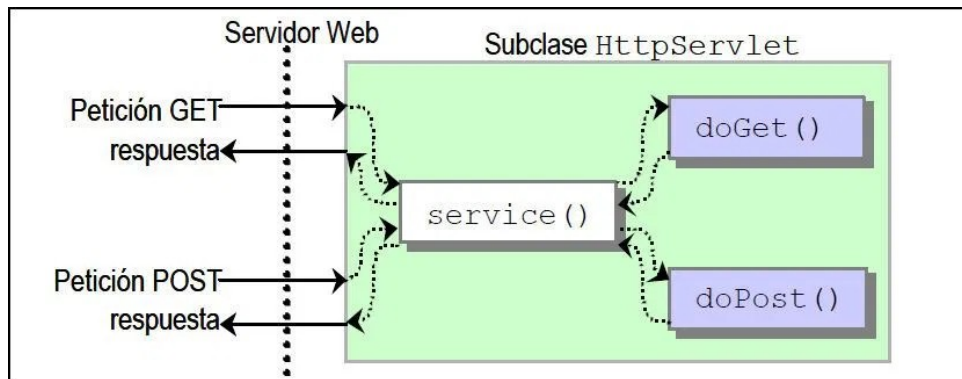
SERVICIO: una llamada a `service()` por cada invocación al servlet para procesar las peticiones de los clientes web.

DESTRUCCIÓN: Cuando todas las llamadas desde el cliente cesen o un temporizador del servidor así lo indique o el propio administrador así lo decida se destruye el servlet. Se usa el método “`destroy`” para eliminar al servlet y para “recoger sus restos” (garbage collection).

Cada vez que el servidor pasa una petición (distinta a la primera) a un servlet se invoca el método `service()`, este método habrá que sobreescribirlo (override). Este método acepta dos parámetros: un objeto petición (`request`) y un objeto respuesta. Los servlets http, que son los que vamos a usar, tienen ya definido un método `service()` que llama a `doXxx()`, con Xxx el nombre de la orden que viene en la petición al servidor web. Estos dos métodos son `doGet()` y `doPost()` y nos sirven para atender las peticiones específicamente provenientes de métodos GET o POST respectivamente,

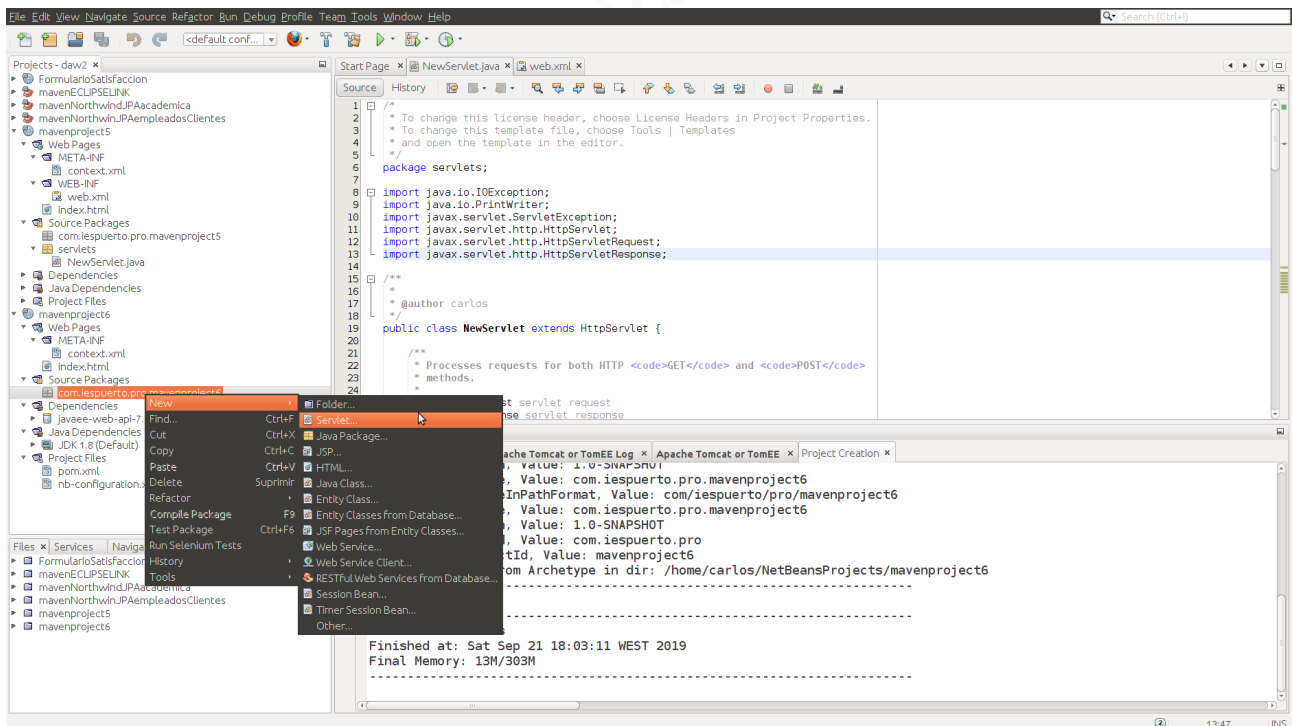
Ciclo de Vida



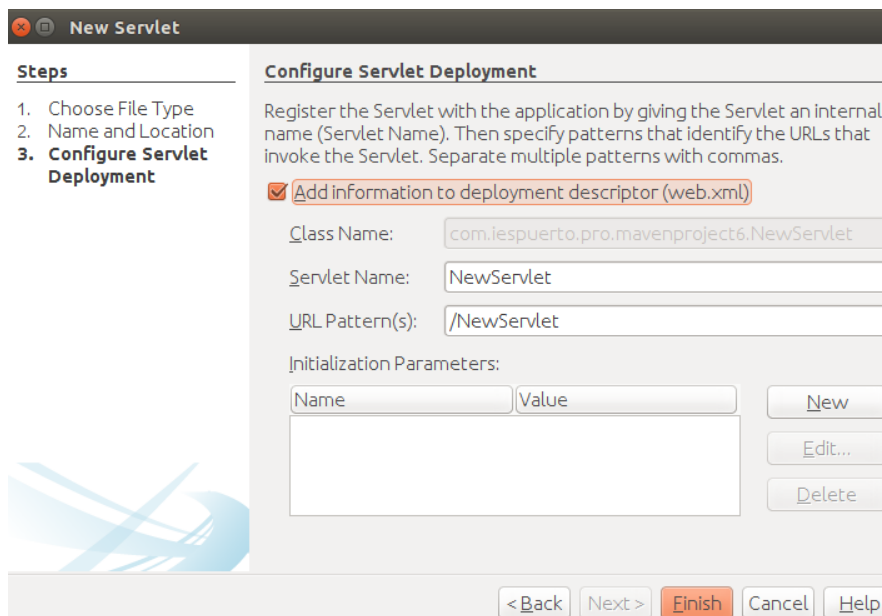


Veamos que tenemos en un servlet creado con netbeans:

Para ello creamos uno: botón derecho → new → servlet



En el wizard nos mostrará el nombre (URL pattern) por el que podremos acceder mediante el navegador a ese servlet También nos preguntará si queremos agregar la información en web.xml Esa es la forma en versiones anteriores de guardar la información del mapeo de los servlet con las url accesibles desde el navegador. Nosotros elegiremos que queremos que nos lo haga:



Una vez finalizado observemos el fichero creado:

los import:

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

Vemos que la clase extiende de HttpServlet:

```
public class NewServlet extends HttpServlet {
```

Y observamos los métodos:

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        /* TODO output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet NewServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet NewServlet at " + request.getContextPath()
+ "</h1>");Servlets
        out.println("</body>");
        out.println("</html>");
    }
}
```

Este método se ejecuta tanto si la petición se realiza a través de GET como si es a través de POST y lo único que hace es devolver un texto. Merece especial consideración las instrucciones:

- response.setContentType("text/html;charset=UTF-8"); Esta instrucción fija el tipo de contenido que será devuelto al cliente. EN ESTE CASO DEVUELVE HTML
- PrintWriter out = response.getWriter(); Esta instrucción crea el objeto "out" que permitirá escribir la salida.
- request.getContextPath(); , contiene el path desde donde se ha realizado la petición

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
```

generalmente para Get Este método se ejecuta por defecto

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
```

generalmente para Post

Cuando no existe un atributo “method” en una etiqueta form de HTML que especifique el método GET o POST, el servlet SIEMPRE, por defecto, responde con el método doGet().

Modificaremos el Servlet0 de manera que no exista el método processRequest. Esta vez haremos que cada uno de los métodos doGet() y doPost() responda cada uno por sí sólo.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    //processRequest(request, response);
    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>Servlet NewServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet NewServlet at " + "SE HA EJECUTADO DOGET" +
"</h1>");
        out.println("</body>");
        out.println("</html>");

    }
}

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    //processRequest(request, response);
    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>Servlet NewServlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet NewServlet at " + "SE HA EJECUTADO DOPOST"
+ "</h1>");
        out.println("</body>");
        out.println("</html>");

    }
}
```


Accedamos por telnet y pongamos post y get según el caso:

```
telnet localhost 8080
POST /mavenproject5/NewServlet HTTP/1.1
HOST: localhost
```

Observar que hay que poner el nombre del proyecto en lugar de **mavenproject5** y el nombre de nuestro servlet en lugar de **NewServlet** .

Hacemos lo mismo pero con get (no olvidar en ambos caso el retorno de carro después de nuestra última línea)

```
telnet localhost 8080
GET /mavenproject5/NewServlet HTTP/1.1
HOST: localhost
```

observar que la clase no tiene ni método init() ni método destroy. No son necesarios, ni obligatorios

● **Práctica 1:** Hacer las modificaciones pertinentes en los dos métodos y actuar mediante telnet con el servlet tal y como se describe con los comandos HTTP . Tomar captura de pantalla de la ejecución y obtención de la respuesta del servidor por telnet

JSP

Son páginas HTML con código Java embebido de dos formas:

- * Scriptlets: código Java multi-mensaje entre los símbolos `<% %>`. Cada mensaje debe ir separado por punto y coma.
- * Expresiones: un mensaje Java que devuelve un resultado. No finaliza con punto y coma y se escribe entre los símbolos `<%= %>`

Adicionalmente a lo anterior usaremos: `<%-- --%>` para los comentarios

Veamos una página jsp con esos tres elementos:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
  <title>Ejemplos JSP</title>
</head>
<body>
  <H1>Ejemplo de scriptlet</H1>
  <%
    int numero = 7, factorial = 1;
    for(int i=numero; i>1; i--) {
      factorial *= i;
    }
  %>
```

```

    %>
    <!-- Se muestran en negrita el número y el resultado del factorial -->
    <%= "El factorial de <b>"+numero+"</b> es <b>"+factorial+"</b>" %>
</body>
</html>

```

También podemos hacer declaraciones (de variables y métodos) que se puede usar en cualquier lugar de la página JSP mediante: `<%! %>`

```

<%!

private String calcularTabla(int num){
    String resultado = "";

    for(int i=1;i<11;i++){
        resultado += i + " * " + num + " = " + ( i * num ) + "<br>";
    }

    return resultado;
}

%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Calcular tabla del número 2</h1>
        <%
            String res = calcularTabla(2);

```

```
        out.println(res);  
    %>  
</body>  
</html>
```

También podríamos declarar un atributo del servlet generado del JSP mediante `<%! %>` (completamente desaconsejado)

```
<%! static contador = 0 %>
```

Ya hemos visto un par de veces (justo aquí encima también) la directiva **page**. Hablemos de ella:

Directiva page

Permite declarar una serie de propiedades para la página que se está desarrollando. Con esta directiva se pueden usar los siguientes atributos: language, extends, import, session, buffer, autoFlush, info, etc. Los más usuales son language, contentType, pageEncoding e import (para importar clases)

```
<%@page language="java"  
    contentType="text/html; charset=ISO-8859-1"  
    pageEncoding="ISO-8859-1"  
    import="java.util.*" %>
```

Los elementos HTTP definidos en los servlets están predefinidos en los JSP, como:

* request

- Objeto **HttpServletRequest** que permite acceder a la información de la solicitud

* response

- Objeto **HttpServletResponse** para generar la respuesta

* session

- Objeto `HttpSession` asociado a la petición
- Si no hubiera sesión será null

*** out**

Objeto `JspWriter` (similar a un `PrintWriter`) para generar la salida para el cliente

Bien, los JSP están pensados para ser esencialmente vistas de nuestra aplicación. Eso significa que la información se la pasarán principalmente los Servlet y en ocasiones directamente de un formulario ¿Cómo recoger la información que nos pasan? Para saber los parámetros que hemos recibidos podemos apoyarnos en un mapa del objeto `request`:

```
Map<String, String[]> mapa = request.getParameterMap();
```

Ahí tenemos todos los parámetros. Para recorrerlo podemos hacerlo como habitualmente con mapa:

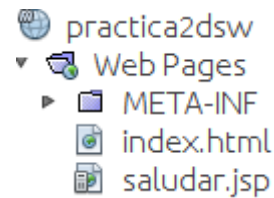
```
for( String parametro: request.getParameterMap().keySet()){  
    for(String valor: mapa.get(parametro)){  
        // aquí tenemos en: parametro el nombre del parámetro y en: valor los valores  
    }  
}
```

Vamos a realizar una actividad sencilla para practicar con JSP

La aplicación tendrá dos páginas.

Una página estática: index.html

y una página dinámica: saludar.jsp



index.html mostrará:

Saludar

introducir nombre1:

introducir nombre2:

introducir apellidos:

A la recepción de la información saludar.jsp mostrará:

Veamos todos los parámetros

- nombre:
 - Ana
 - Micaela
- apellidos:
 - Marín Mola

Te saludo: nombre: Ana Micaela ;; apellidos: Marín Mola ;; !!

Observar que se está enviando el parámetro nombre dos veces por ese motivo tanto Ana, como Micaela forman parte del array de String de la key: nombre

 **Práctica 2:** Realizar la aplicación que se acaba de describir

Vamos a analizar ahora las redirecciones. Algo habitual es que un servlet redirija a una página u otra según lo que tenga lugar. En ese proceso de redirección puede proceder a establecer información adicional para la página a la que va a redirigir.

Haremos uso de: `request.setAttribute()` para establecer información a la página que vamos a redirigir y usaremos: `request.getRequestDispatcher("nombrepagina.jsp");` y `RequestDispatcher.forward()` para hacer la redirección

Ejemplo:

```
request.setAttribute("prueba", 5); // establecemos el parámetro: prueba a: 5
// reenviamos:
RequestDispatcher rd = request.getRequestDispatcher("unaPagina.jsp");
rd.forward(request, response);
```

Nota: Observar que también existe el método: `response.sendRedirect()` que quizás se pensara usarlo en lugar de `request.getRequestDispatcher()` para la redirección. Con el primero, el navegador tiene que hacer otra petición al servidor para mostrar el JSP correspondiente (la URL en el navegador cambia), por lo que si se estuviera pasando información necesitas tener el objeto en la sesión (ya hablaremos de ellas), es decir, en el objeto session.

Si se utilizara el método `request.getRequestDispatcher`, el navegador no necesitaría hacer otra petición (en este caso, la URL del navegador sigue siendo la misma) y adicionalmente no hay que solicitar la información de los objetos de sesión

Adicionalmente observar que hemos usado `rd.forward(request, response);` Forward continúa el proceso de la misma petición en la URL que se le pasa, el navegador no hace nada. Redirect indica al cliente (browser) que tiene que redirigirse a la nueva página, creando una nueva petición.

El equivalente en JSP de `request.setAttribute()` es: `request.getServletContext().setAttribute()`

Observar que esto es diferente de **parameter** que usábamos hasta ahora. La forma de recorrer con request los **atributos** es mediante:

```
Enumeration<String> atributos = request.getAttributeNames();
```

```
while( atributos.hasMoreElements()){  
    String atributo = atributos.nextElement();  
    // atributo es la key, request.getAttribute(atributo) sería el value  
}
```

Si en lugar de recorrer los atributos queremos alcanzar un atributo en concreto usamos:

```
request.getAttribute(name) // donde name es el nombre del atributo
```

Debemos entender que `getParameter()` nos devuelve los parámetros. AQUELLOS PARÁMETROS QUE EL CLIENTE ENVÍA AL SERVIDOR. `getAttribute()` es desde el lado del servidor, son datos que establece un servlet y por ejemplo, se envían a un jsp. Adicionalmente observar que `getParameter()` devuelve String mientras que `getAttribute()` devuelve cualquier cosa

Vamos a ver lo anterior todo con un ejemplo completo:

la página **index.html**

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Start Page</title>  
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  
  </head>  
  <body>  
    <h1>Hello World!</h1>  
    <a href="procesar?unParametro=7" >pulsa aquí por favor </a>  
  </body>  
</html>
```

Observar que se llama a un servicio llamado: procesar y se le está enviando por método GET un parámetro llamado: unParametro con valor: 7

El contenido del fichero: **web.xml**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
  <servlet>
    <servlet-name>Pablo</servlet-name>
    <servlet-class>controller.ServletPrueba</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Pablo</servlet-name>
    <url-pattern>/procesar</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

Observar que “/procesar” está mapeado a un nombre de Servlet que es: “**Pablo**” y a su vez tenemos la ruta de ese nombre de Servlet que realmente coincide con una clase Java llamada: “**ServletPrueba**”

Veamos ahora parte del contenido de ServletPrueba.java:

```

public class ServletPrueba extends HttpServlet {

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

        //vamos a tomar el dato que nos envió la url ( mediante GET )
        String strUnParametro = request.getParameter("unParametro");

        int unParametro = 0;
        try{
            unParametro = Integer.parseInt(strUnParametro);
        }catch(Exception ex){}

        //vamos a sumar 5 al dato que nos venga en unParametro y lo vamos a enviar
        //mediante un atributo llamado: prueba
        request.setAttribute("prueba", unParametro + 5);

        //request.getRequestDispatcher("unaPagina.jsp").forward(request,response);
        RequestDispatcher rd = request.getRequestDispatcher("unaPagina.jsp");
        rd.forward(request, response);
    }
}

```

Vemos que se usa: `request.getParameter()` para obtener la información que nos enviaron

Después vemos que creamos un nuevo atributo de la request llamado: “prueba” y estamos ingresando el número que nos enviaron dentro del parámetro: “unParametro” y le sumamos 5

Luego vemos que redireccionamos hacia una página JSP llamada: **unaPagina.jsp**

Veamos ahora el contenido de: unaPagina.jsp

```
<%@page import="java.util.Enumeration"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    <%
      Enumeration<String> atributos = request.getAttributeNames();

      while( atributos.hasMoreElements()){
        String atributo = atributos.nextElement();
        out.write(atributo + "<br>");
        out.write(request.getAttribute(atributo) + "<br><br>");
        // atributo es la key, request.getAttribute(atributo) sería el value

      }

    %>
    <p>Efecto usando getParameter(): <%= request.getParameter("prueba") %>
    </p>
    <p>Efecto usando getAttribute(): <%= request.getAttribute("prueba") %>
    </p>
  </body>
</html>
```

Observamos como podemos recorrer todos los atributos sin conocer sus nombres

Y también que si buscamos el dato como un parámetro no lo encontraremos: (request.getParameter()) pero que si lo buscamos como atributo sí: request.getAttribute()

Tener en cuenta que con getAttribute() podemos tomar cualquier tipo de objeto, getParameter() devuelve únicamente String

- **Práctica 3:** Realizar la aplicación que desde index.html tenga un formulario para nombre y apellido con get y otro con post. Lo recibirá un servlet llamado: **procesarformulario** que establecerá un nuevo atributo llamado: **tipoformulario**. Ese parámetro se establecerá a: "bien!. Usó POST" si el usuario envió con post, y se establecerá a: "no apropiado. Usó GET", si fue con get Luego reenviará a una página jsp llamada: **resultado.jsp** Esta página mostrará el nombre y apellido introducidos y el contenido del atributo: **tipoformulario**

Java Bean

Según wikipedia:

JavaBeans son un modelo de componentes creado por Sun Microsystems para la construcción de aplicaciones en Java.

Se usan para encapsular varios objetos en un único objeto (la vaina o Bean en inglés), para hacer uso de un solo objeto en lugar de varios más simples.

La especificación de JavaBeans de Sun Microsystems los define como "componentes de software reutilizables que se puedan manipular visualmente en una herramienta de construcción".

Requisitos de una clase para ser un JavaBean:

- Debe tener un constructor por defecto.
- Sus atributos de clase deben ser privados.
- Sus propiedades deben ser accesibles mediante métodos get y set
- Debe ser serializable

fuelle: wikipedia

Podremos a pesar de todo hacer uso de las ventajas de JavaBean sin necesidad de haber puesto la interfaz serializable (tener en cuenta que es buena idea que cumpla con ser serializable desde que muchas veces queremos transmitir nuestra información por la red)

Por lo general un JavaBean se asocia con una tabla en la base datos, de tal forma que las columnas de la tabla en la base de datos deben ser propiedades/atributos en un JavaBean, aunque pueden existir más propiedades, todo dependerá de las necesidades del programador. Observar que cuando generamos mediante JPA clases mapeadas desde una base de datos todas las clases que aparecen cumplen los requisitos que anteriormente hemos descrito

● **Práctica 4:** Desde un JSP tener un formulario para insertar en la tabla Persona mediante JPA. Tomar captura de pantalla de registros en la tabla persona mostrando la inserción correspondiente, así como de los datos al ponerlos en la página jsp (adicionalmente al proyecto netbeans)

El ejercicio anterior tiene más sentido si aplicamos la idea que tenemos de vista-controlador usando JavaBean. La ventaja de JavaBean, entre otras, es que podemos hacer uso de EL (Expression Language)

Haremos una página estática con formulario para introducir la persona que será enviado a un servlet. El servlet creará la nueva persona (que debe cumplir los requisitos para ser un JavaBean) y se agregará como un atributo para reenviar hacia una página JSP. Finalmente en una página JSP mostraremos los datos de la persona creada. Usaremos **EL** para este cometido. Sabiendo que en EL si ponemos: **`${expresion}`** nos evaluará el contenido. Así si por ejemplo, el servlet nos ha enviado la Persona en un atributo llamado: mipersona entonces podemos recuperar esa información en el JSP mediante:

```
<body><p>
${mipersona.nombre }
${mipersona.apellidos }
</p></body>
```

● **Práctica 5:** Realizar la aplicación descrita usando **EL** en el JSP y haciendo el procesamiento del formulario en el servlet (así como la inserción en la base de datos desde el servlet)

EL permite múltiples cosas:

Operaciones aritméticas:

```
${ mipersona.edad / 2 }
```

Condicionales:

```
${ (mipersona.edad >= 18)?"adulto":"menor" }
```

Uso de funciones en anotación funcional (al estilo de las lambdas). Lo siguiente devuelve 8.5:

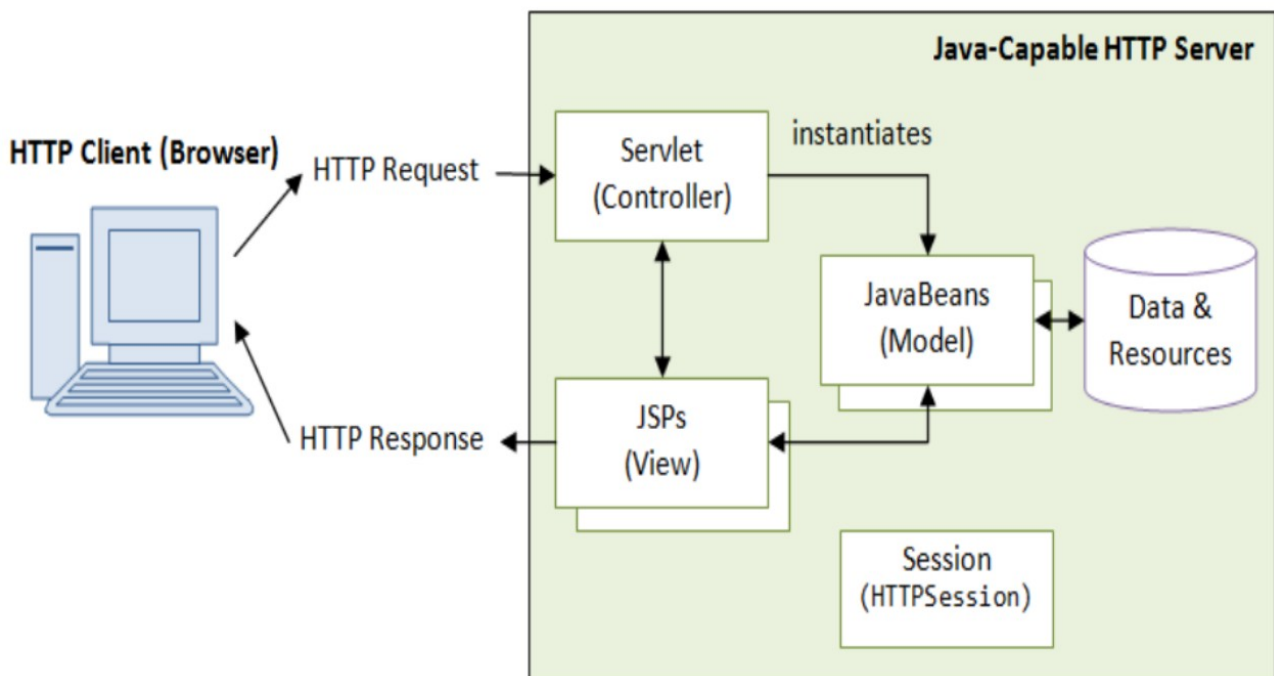
```
${((x, y) -> x + y)(3, 5.5)}
```

Ejemplos y descripción más completa en:

<https://docs.oracle.com/javaee/7/tutorial/jsf-el007.htm#BNAIM>

MVC con JavaBeans

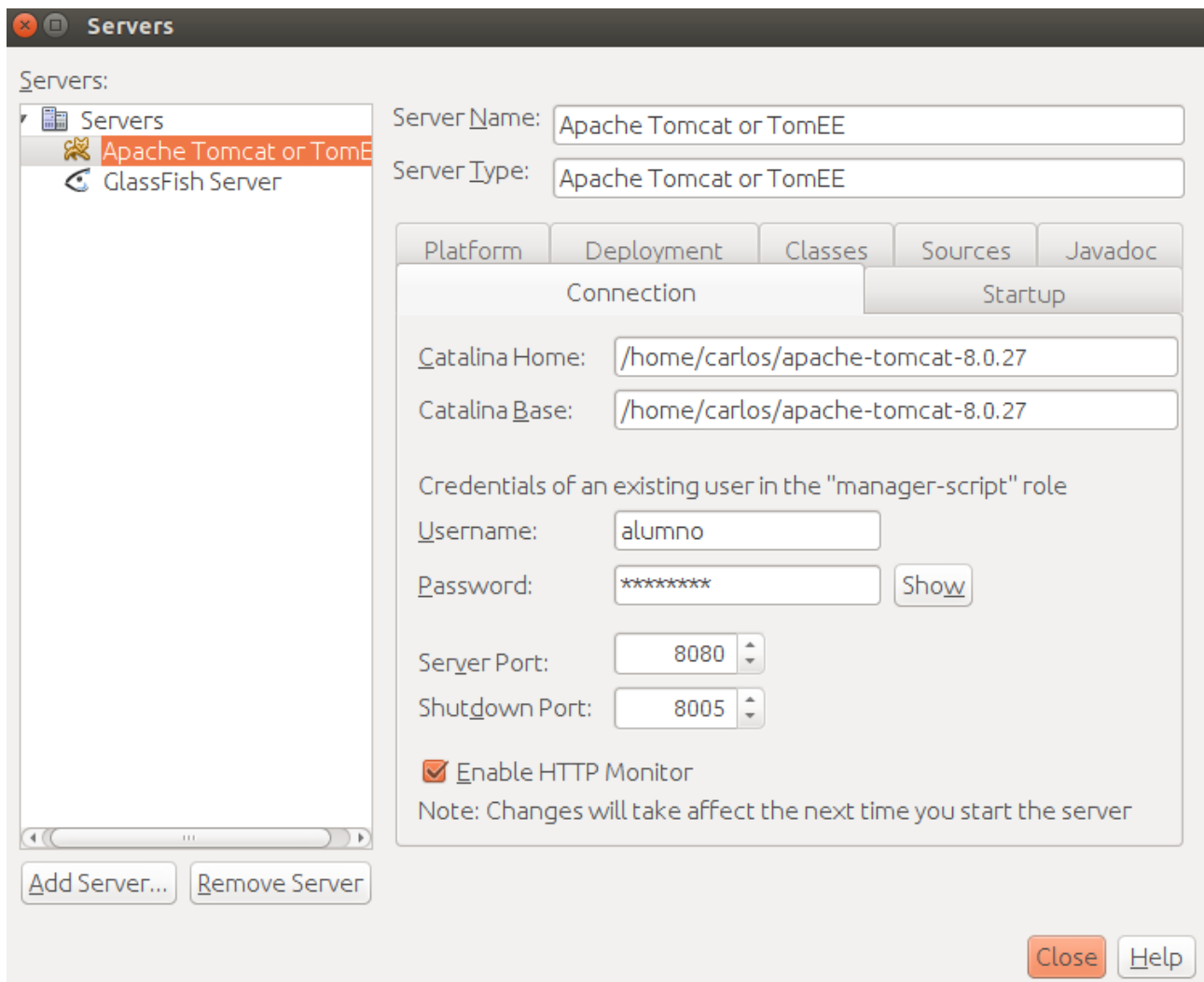
Ahora que hemos visto los JavaBean observemos de nuevo con este gráfico el funcionamiento de modelo-vista-controlador:



Fácilmente con JPA atacamos las bases de datos y como sabemos las clases que se crean cumplen el requisito de JavaBean. En general tomaremos la petición (request) en un servlet (parte

controlador) procesaremos mediante clases JavaBean procedentes del modelo y enviaremos Beans con la información que queremos que nos muestre la vista JSP. Luego desde ese fichero JSP visualizaremos los datos recibidos.

Nota: Para observar los mensajes en el servidor, los parámetros, cookies y demás nos puede ser útil hacer uso de: HTTP Server Monitor es una opción específica que ponemos en Netbeans cuando establecemos el servidor tomcat (observar en la siguiente imagen el checkbox Enable HTTP Monitor)



Content-type

Cuando vemos un Servlet generado por el IDE podemos ver algo similar a:


```

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        /* TODO output your page here. You may use following sample code. */
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet procesarformulario</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet procesarformulario at " + request.getContextPath() + "</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}

```

fijémonos en: `response.setContentType()` ahí observamos que estamos estableciendo que tipo de documento está dando como respuesta (response) a la solicitud (request) que el Servlet ha procesado. Los tipos MIME son los que se usan para establecer el tipo de documento.

El tipo Extensiones multipropósito de Correo de Internet (MIME) es una forma estandarizada de indicar la naturaleza y el formato de un documento. Está definido y estandarizado en IETF RFC 6838.

Los navegadores a menudo usan el tipo MIME (y no la extensión de archivo) para determinar cómo procesará un documento; por lo tanto, es importante que los servidores estén configurados correctamente para adjuntar el tipo MIME correcto al encabezado del objeto de respuesta.

Sintaxis de un tipo MIME

Viene definida por:

tipo/subtipo

La estructura de un tipo MIME es muy simple; consiste en un tipo y un subtipo, dos cadenas, separadas por un '/'. No se permite espacio. El tipo representa la categoría y puede ser de tipo discreto o multiparte. El subtipo es específico para cada tipo.

Un tipo MIME no distingue entre mayúsculas y minúsculas, pero tradicionalmente se escribe todo en minúsculas.

Veamos algunos ejemplos:

Tipo	Descripción	Ejemplo de subtipos típicos
text	Representa cualquier documento que contenga texto y es teóricamente legible por humanos	text/plain, text/html, text/css, text/javascript
image	Representa cualquier tipo de imagen. Los videos no están incluidos, aunque las imágenes animadas (como el gif animado) se describen con un tipo de imagen.	image/gif, image/png, image/jpeg, image/bmp, image/webp
audio	Representa cualquier tipo de archivos de audio	audio/midi, audio/mpeg, audio/webm, audio/ogg, audio/wav
video	Representa cualquier tipo de archivos de video	video/webm, video/ogg
application	Representa cualquier tipo de datos binarios.	application/octet-stream, application/pkcs12, application/vnd.ms-powerpoint, application/xhtml+xml, application/xml, application/pdf

Para documentos de texto sin subtipo específico, se debe usar `text/plain`. De forma similar, para los documentos binarios sin subtipo específico o conocido, se debe usar `application/octet-stream`.

En el ejemplo que pusimos antes del Servlet que nos generó el IDE observamos que nos puso el tipo: “`text/html`” estableciendo que estamos devolviendo una página web html

En ausencia de un tipo MIME, o en algunos otros casos en los que un cliente cree que están configurados incorrectamente, los navegadores pueden realizar **el rastreo MIME**, **que es adivinar el tipo MIME correcto mirando el recurso**. Cada navegador realiza esto de manera diferente y bajo diferentes circunstancias

Responder desde Servlet con diferentes Content-type

Vamos a trabajar como desde un Servlet podemos devolver cosas diferentes a un html como hemos hecho hasta ahora. Para ello haremos uso de lo que acabamos de ver respecto a los tipos MIME

Devolver json:

En response en lugar de `response.setContentType("text/html;charset=UTF-8");`

lo que pondremos es: `response.setContentType("application/json");`

y el resto es muy similar. Veamos ejemplo:



En la imagen de arriba vemos un proyecto llamado practica5dsw y se observa el código que se ha introducido en un **Servlet llamado Descargar**. Si comparamos con el código generado por el IDE cuando le pedimos un servlet vemos diferencia en las siguientes líneas:

```

response.setContentType("application/json");
PrintWriter out = response.getWriter();

out.println("{\"alumnos\":[\n"
    + "    {\"nombre\":\"Ana\", \"apellidos\":\"Marín\"},\n"
    + "    {\"nombre\":\"Marta\", \"apellidos\":\"Buendía\"},\n"
    + "    {\"nombre\":\"Marco\", \"apellidos\":\"Fernández\"}\n"
    + "]}");
out.close();

```

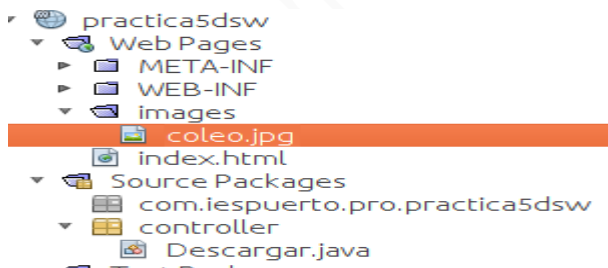
Principalmente lo que cambia es que se ha puesto: “application/json” en el content-type

Devolver imagen:

En esta ocasión el contenttype será:

```
response.setContentType("image/jpeg");
```

Supongamos que tenemos una imagen llamada: coleo.jpg en una carpeta llamada: images que cuelga directamente en Web Pages:



El código del servlet será:

```

* @author carlos
*/
public class Descargar extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException { ...15 lines }

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

        response.setContentType("image/jpeg");

        // File.separator devuelve slash / o backslash \ si linux o windows
        String pathToWeb = getServletContext().getRealPath(File.separator);
        File f = new File(pathToWeb + "images/coleo.jpg");
        BufferedImage bi = ImageIO.read(f);
        OutputStream out = response.getOutputStream();
        ImageIO.write(bi, "jpg", out);
        out.close();

    }
}

```

Veamos el código que cambia con más detalle:

Lo que haremos es “escribir” nuestra imagen en nuestro outputstream:

```

response.setContentType("image/jpeg");

// File.separator devuelve slash / o backslash \ si linux o windows
String pathToWeb = getServletContext().getRealPath(File.separator);
System.out.println(pathToWeb);

File f = new File(pathToWeb + "images/coleo.jpg");
BufferedImage bi = ImageIO.read(f);
OutputStream out = response.getOutputStream();
ImageIO.write(bi, "jpg", out);
out.close();

```

Fijémonos en `getRealPath()` mediante esa instrucción obtenemos la dirección raíz de nuestra aplicación. Hacemos uso de `File.separator` para que nos ponga barra o barra invertida si estamos en linux o windows respectivamente

Hemos usado `ImageIO` para leer desde nuestro sistema de ficheros la imagen y posteriormente escribirla en el `outputstream`

● **Práctica 6:** Sea el siguiente `index.html`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Probando descargas</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Probando descargas</h1>
    <a href="descargar?informacion=imagen" > imagen</a>
    <a href="descargar?informacion=json" > json</a>
    <a href="descargar" > sin parametros</a>
  </body>
</html>
```

Hacer un servlet apropiado para procesar los href de `index.html` donde devuelva imagen de una planta si se seleccionó “imagen” o un json: equipotrabajo con los nombres de algún compañero tuyo y el tuyo propio informando así de los componentes de un grupo de trabajo de actividades de clase.

Finalmente para el tercer enlace mostrar un html que diga: “Se está devolviendo HTML”

Si en la actividad anterior hubiéramos querido acceder directamente a la imagen habríamos podido (suponiendo que tomcat está en localhost, y el nombre de nuestro proyecto fuera `practica6dsw`) mediante url:

<http://localhost:8080/practica6dsw/images/coleo.jpg>

Una forma para verle mejor entonces el sentido a la descarga desde el servlet sería si la imagen no pudiera ser accesible desde la web. Esto se puede hacer fácilmente poniendo la carpeta images dentro de WEB-INF

WEB-INF

Vamos a ver la descripción oficial de esta carpeta y subcarpetas:

WEB-INF

Directorio que contiene todos los recursos relacionados con la aplicación web que no se han colocado en el directorio raíz y **que no deben servirse al cliente**. Esto es importante, ya que **este directorio no forma parte del documento público, por lo que ninguno de los ficheros que contenga va a poder ser enviado directamente a través del servidor web.**

En este directorio se coloca el archivo web.xml, donde se establece la configuración de la aplicación web.

WEB-INF/classes

Directorio que contiene todos los servlets y cualquier otra clase de utilidad o complementaria que se necesite para la ejecución de la aplicación web. Normalmente contiene solamente archivos .class.

WEB-INF/lib

Directorio que contiene los archivos Java de los que depende la aplicación web. Por ejemplo, si la aplicación web necesita acceso a base de datos a través de JDBC, en este directorio es donde deben colocarse los ficheros JAR que contengan el driver JDBC que proporcione el acceso a la base de datos. Normalmente contiene solamente archivos .jar.

Así pues si ponemos ficheros en esa carpeta web-inf no se podrá acceder directamente mediante url sino que se tendrá que procesar mediante un Servlet

Lo anterior ha dado lugar a que varios frameworks de Java recomienden en sus proyectos poner incluso, los ficheros JSP en el interior de web-inf.

En la página de spring (uno de los frameworks más usados) dice:

“As a best practice, we strongly encourage placing your JSP files in a directory under the 'WEB-INF' directory, so there can be no direct access by clients.”

Esto no debe tomarse como una máxima pero sí que nos ayuda a entender que, en parte complica el proyecto ya que todo jsp se accederá desde servlet, nuestras páginas serán un poco más seguras. **NO hay obligación de poner en esa carpeta, y en ningún caso parece lógico poner páginas html estáticas, estilos y javascript en su interior**

¿ Así pues como procederíamos para acceder al fichero ?

Una vez hubiéramos puesto la carpeta images dentro de WEB-INF debemos tomar la ruta completa al fichero mediante `getRealPath` igual que antes, únicamente incluiremos la parte de ruta WEB-INF:

```
String pathToWeb = getServletContext().getRealPath(File.separator);  
File f = new File(pathToWeb + "WEB-INF/images/coleo.jpg");
```

● **Práctica 6.1:** Modificar la práctica anterior para que ahora la carpeta images esté localizada en WEB-INF y dejar la aplicación funcional. Debe tomarse captura de pantalla tratando de acceder mediante url directamente a la imagen de la planta y captura accediendo mediante la petición al servlet

● **Práctica 7:** Hacer una página de login y que un servlet procese la información post si el usuario se llama admin y como clave puso: 1q2w3e4r se le mostrará: OK! Bienvenido si no coincide el par usuario contraseña un mensaje: NAK! Inténtelo de nuevo

Subir y bajar ficheros

Hemos visto el proceso de bajar algunos tipos de ficheros. Vamos a hacer una pequeña aplicación que permita al usuario salvaguardar todo tipo de fichero y recuperarlo posteriormente, PERO GUARDANDO EN WEB-INF para garantizar que si posteriormente queremos securizar con usuario y contraseña los ficheros almacenados no son accesibles mediante url.

El index.html en su versión más básica:

Almacen Ficheros

[Listar archivos subidos](#)

Subir ficheros

Examinar...

No se han seleccionado archivos.

Subir

Se observa que tenemos un enlace hacia un servlet que nos va a listar los ficheros almacenados. Veamos una salida posible de las que diera ese servlet:

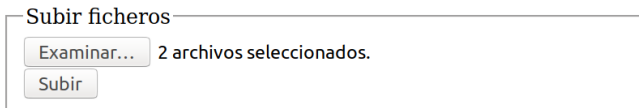
Lista ficheros:

- [the_lightHouseGirl.mp3](#)
- [theEnemy.mp3](#)

En este último servlet observamos que la lista de ficheros tiene enlaces de descarga. Si el usuario pulsa encima de esos enlaces se le devuelve el fichero solicitado (por medio del servlet, no como una url ya que está almacenado en WEB-INF)

Cuando pulsamos en el submit del formulario para subir ficheros nos avisa si la subida tuvo o no lugar. **Tener en cuenta que este formulario se enviará a un servlet distinto del anterior:**

Con dos ficheros seleccionados antes de pulsar “subir”:



Subir ficheros

Examinar... 2 archivos seleccionados.

Subir

La respuesta con el ok del servlet:

Fichero se ha subido!!

¿ qué cosas nuevas precisamos saber para realizar la aplicación ?

- 1 - Como subir ficheros
- 2 - Como listar ficheros de un directorio
- 3 - Como descargar cualquier tipo de fichero

Veamos por partes todo lo preciso. Primero echemos un vistazo a un index.html posible:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Almacen Ficheros</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Almacen Ficheros</h1>
    <br><br><br>

    <a href="listarficheros" >Listar archivos subidos </a>
    <br><br>

    <form method="post" action="subirfichero" enctype="multipart/form-dat
    <fieldset>
      <legend>Subir ficheros</legend>

      <input type="file" name="files" id="files" multiple/><br>

      <input type="submit" value="Subir" />
    </fieldset>

  </form>
</body>
</html>
```

Observamos que hay dos direcciones distintas (que en este caso corresponden a dos servlet distintos)

- * listarficheros

- * subirfichero

También vemos que el formulario para subirfichero tiene el enctype="multipart/form-data" esto es relevante para la parte de subir ficheros. Pero vayamos por partes

Como subir ficheros

Hemos visto que el formulario se estableció multipart/form-data ¿ por qué es esto ? Pues bien, Al enviar un formulario POST hay que codificar de alguna forma el cuerpo de nuestra petición. HTML tiene tres posibilidades:

tipos de codificaciones posibles con POST

- application/x-www-form-urlencoded (por defecto)
- multipart/form-data
- text/plain

Cuando enviamos ficheros (mediante `input type="file"`) la codificación más apropiada es: `multipart/form-data`. En la mayoría de otras ocasiones la opción por defecto (`application/x-www-form-urlencoded`) es la más apropiada y también es válida para subir ficheros pero para éste cometido es más eficiente form-data. En general evitaremos el uso de texto plano: `text/plain`

Veamos que nos dice una página que suele ser una buena referencia: developer.mozilla.org

application/x-www-form-urlencoded: Los valores son codificados en tuplas llave-valor separadas por '&', con un '=' entre la llave y el valor. Caracteres no-Alfanumericos en ambas (llaves, valores) son problemáticos para su codificación. Se usa un código con un tanto por ciento para identificarlos. Así por ejemplo, veamos como se codifica un carácter reservado: %24 hace referencia al: "\$". Esta codificación "percent encoded" hace que este tipo no sea adecuado para usarse con datos binarios (use multipart/form-data en su lugar)

como se codifica: x-www-form-urlencoded

- 'a' - 'z', 'A' - 'Z', y '0' - '9' no se modifican.
- El carácter de espacio se transforma en '+'.
• Los demás caracteres se convierten en string de 3-caracteres "%xy", donde xy es la representación hexadecimal con dos dígitos de los 8-bits del carácter

En cuanto a por qué evitar el texto plano (`text/plain`) leamos lo que dice el estandar HTML5:

text/plain: Las cargas útiles que utilizan el formato texto / plano están destinadas a ser legibles por el ser humano. No son confiablemente interpretables por computadora, ya que el formato es

ambiguo (por ejemplo, no hay forma de distinguir una nueva línea literal en un valor de la nueva línea al final del valor)

Para consultarlo en contexto original ir a:

<http://dev.w3.org/html5/spec/association-of-controls-and-forms.html#plain-text-form-data>

Crear el directorio donde guardar los ficheros

Podemos tener creada a mano una carpeta llamada: “almacen” dentro de WEB-INF pero también podemos crearla programáticamente para así estar preparados si necesitamos crear carpetas en futuras aplicaciones

Supongamos que tenemos en una variable de tipo string llamada: “path” la ruta a la carpeta que deseamos crear. Entonces haremos:

```
File directorio = new File(path);  
if (!directorio.exists())  
    directorio.mkdir();
```

Como vemos es tan sencillo como hacer uso del método mkdir()

Recoger los ficheros del request y guardarlos

Es importante tener en cuenta que vamos a usar codificación perteneciente a versiones de Java EE superiores a la 6 (Servlets mínimo versión 3.0) Pero para hacer uso de la misma debemos establecer una anotación que preceda el código del servlet: **@MultipartConfig**

Así por ejemplo, la definición nos quedaría:

```
import javax.servlet.http.Part;  
  
/**  
 *  
 * @author carlos  
 */  
@MultipartConfig  
public class SubirFichero extends HttpServlet {
```

Pues bien, una vez establecida esa anotación Basta con hacer lo siguiente en el método `doPost()`:

```
for (Part part : request.getParts()) {  
    String nombreFichero = part.getSubmittedFileName();  
    part.write(path + File.separator + nombreFichero);  
}
```

Tener en cuenta que la variable de tipo String: “path” es la misma que nombramos antes, y recoge el nombre de la carpeta donde vamos a almacenar los ficheros. También si nos fijamos en el código anterior podemos ver que es capaz de procesar varios ficheros que se hayan enviado en la solicitud a la vez y guardarlos con su respectivo nombre

Bajar Ficheros

Lo siguiente ya corresponde al servlet: `ListarFicheros`

Como listar ficheros de un directorio

Al igual que antes necesitamos el path completo de la carpeta donde estamos almacenando:

```
String path = getServletContext().getRealPath("") +  
    File.separator + "WEB-INF" + File.separator + ALMACEN;  
File carpeta = new File(path);
```

Ahora únicamente precisamos el método: `File.list()`

```
String[] listado = carpeta.list();
```

Como vemos devuelve un array de String conteniendo los nombres de los ficheros. Evidentemente si devuelve null o el tamaño del array es 0 significa que está vacía la carpeta

Ahora lo único que nos falta es saber como descargar cualquier tipo de fichero:

Como descargar cualquier tipo de fichero

Observar que para que la aplicación funcione correctamente tiene que conseguirse que del listado de ficheros que se le muestra al usuario, cuando pulse en uno de los elementos deberá enviarse la información pertinente al servidor para saber que debe proceder a darle el fichero solicitado al usuario. Se deja esta parte a la resolución del alumno

Supongamos pues que estamos en el servlet y sabemos cuál es el fichero que queremos descargar, entonces obtenemos su correspondiente File:

```
File fichero = new File(path + File.separator + nombreFichero);
```

y estableceremos las cabeceras de respuesta:

```
response.setContentType("application/octet-stream");  
response.setHeader( "Content-Disposition",  
    String.format("attachment; filename=\"%s\"", fichero.getName()));
```

Observar que hemos puesto en contenttype: application/octet-stream este siempre es un contenttype válido si no sabemos el tipo de fichero que estamos sirviendo

También debemos fijarnos que en la cabecera de respuesta le decimos que tenemos un fichero adjunto y le damos el nombre del fichero, para que pueda guardarlo con su nombre correspondiente.

Finalmente leemos el fichero de nuestro disco duro y lo escribimos en el stream de salida (enviándolo así al navegador del usuario)

```
try (FileInputStream in = new FileInputStream(fichero)) {  
    OutputStream out = response.getOutputStream();  
    byte[] buffer = new byte[4096];  
    int length;  
    while ((length = in.read(buffer)) > 0){  
        out.write(buffer, 0, length);  
    }  
    out.flush();  
} catch (Exception ex){}
```

● **Práctica 8:** Realizar el sitio web descrito para almacenar ficheros