

JAVA WEB



Juan Carlos Pérez Rodríguez

Sumario

Introducción.....	4
Páginas web estáticas y dinámicas.....	8
Tecnologías para programación web del lado del servidor.....	11
Lenguajes compilados a código intermedio. Java , Asp.Net.....	13
Comparativa script, CGI, código intermedio.....	14
Java Web.....	15
Instalación tomcat y creación proyecto web.....	16
Servlets.....	23
JSP.....	30
Java Bean.....	40
MVC con JavaBeans.....	43
Uso de JSLT.....	45
Enlazar un fichero CSS o JAVASCRIPT a un JSP.....	48
Sesiones.....	50
¿ Qué nos aporta el uso de sesiones ?.....	52
Ámbitos de una aplicación Web.....	55
welcome-file-list en web.xml.....	60
WEB-INF.....	62
Application Listener.....	64
Filtros.....	66
Anexo: Cookies.....	72
Tipos de cookie.....	72
Beneficios de las cookies.....	74
PELIGROS DE LAS COOKIES:.....	74
Creación y obtención de Cookies.....	75
Cabeceras HTTP al establecer y enviar una cookie.....	80
Parámetros disponibles con cookies.....	80
Restricciones en la clave,valor de una cookie.....	81
¿ Cómo borrar una Cookie ?.....	81
Anexo: Proyecto maven netbeans web con payara server.....	84

Juan Carlos Pérez Rodríguez

Introducción

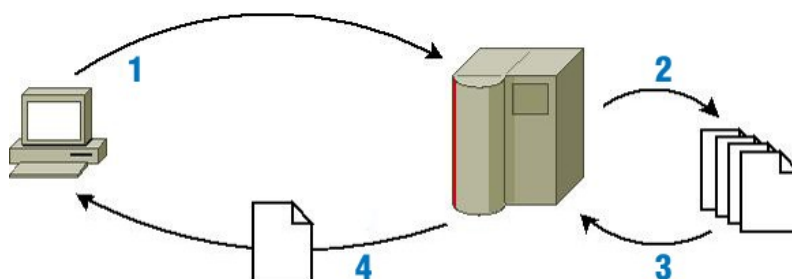
Cuando una página web se descarga a tu ordenador, su contenido define qué se debe mostrar en pantalla. Este contenido está programado en un lenguaje de marcado, formado por etiquetas, como puede ser HTML. dentro de estos lenguajes hay etiquetas para indicar que un texto es un encabezado, que forma parte de una tabla, o que simplemente es un párrafo de texto.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Título de la página</title>
  </head>

  <body>
    <!-- aquí pondremos el contenido de la página -->
  </body>
</html>
```

Esta página probablemente tenga un fichero de estilos CSS vinculado. La hoja de estilos se encuentra indicada en la página web y el navegador la descarga junto a ésta. En ella nos podemos encontrar, por ejemplo, estilos que indican que el encabezado debe ir con tipo de letra Arial y en color rojo, o que los párrafos deben ir alineados a la izquierda.

Ambos ficheros se descargan al ordenador del cliente procedentes de un servidor web como respuesta a una petición. El proceso gráficamente:



Los pasos son los siguientes:

1. **Tu ordenador solicita a un servidor web una página** con extensión .htm, .html o .xhtml.
2. **El servidor busca esa página** en un almacén de páginas (cada una suele ser un fichero).
3. Si **el servidor encuentra esa página**, la recupera.
4. Y por último **se la envía al navegador** para que éste pueda mostrar su contenido.

¿ Cómo se realiza esa petición y la respuesta ? Para ello hacemos uso de un protocolo de comunicaciones. Por ejemplo http

Ejemplo por telnet:

```
telnet blog.iespuertodelacruz.es 80
```

```
GET / HTTP/1.1
```

```
Host: blog.iespuertodelacruz.es
```

```
User-Agent: Mozilla/4.0
```

-----> hay que pulsar una línea en blanco para que lo coja <-----

obtendremos una respuesta HTML informándonos que la nueva localización es por medio de https

Lo que podemos ver en cada línea:

```
GET / HTTP/1.1
```

como podemos observar se establece una conexión se le envía una orden de tipo GET solicitando la ruta raíz: / especificando que se quiere hacer la comunicación mediante la versión del protocolo 1.1 de HTTP

```
Host: blog.iespuertodelacruz.es
```

se especifica el servidor

```
User-Agent: Mozilla/4.0
```

y cuál es el cliente de usuario

Vamos ahora a brindar una página estática y la obtendremos mediante el método anterior (telnet)

sudo apt-get install apache2

creamos el fichero a mostrar:

```
sudo gedit /var/www/html/estatica.html
```

con el contenido:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Título de la página</title>
  </head>
  <body>
    Esta es una página con contenido estático
  </body>
</html>
```

ahora la recuperaremos como antes. Desde un terminal:

```
telnet localhost 80
```

```
GET /estatica.html
```

Obtendremos:

```
root@alumnoVB:/var/www/html# telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET /estatica.html
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Título de la página</title>
  </head>
  <body>
    Esta es una página con contenido estático
  </body>
</html>
Connection closed by foreign host.
```

Así pues, usamos un protocolo de comunicaciones: HTTP para enviar y recibir la información del servidor. Obtendremos una salida codificada en texto y nuestro navegador nos lo transformará en lo que corresponda. Si es un texto HTML nos mostrará la página web, si es un fichero pdf: `'Content-type: application/pdf'` el navegador se encargará de procesar el tipo mime correspondiente y nos mostrará el pdf

Ahora bien, ¿y si queremos que cuándo accedamos a una misma ruta nos pueda devolver contenido dinámico ? Ya no nos valdrá dejar un fichero y que simplemente se lo descargue tendremos que poner programación del lado del servidor

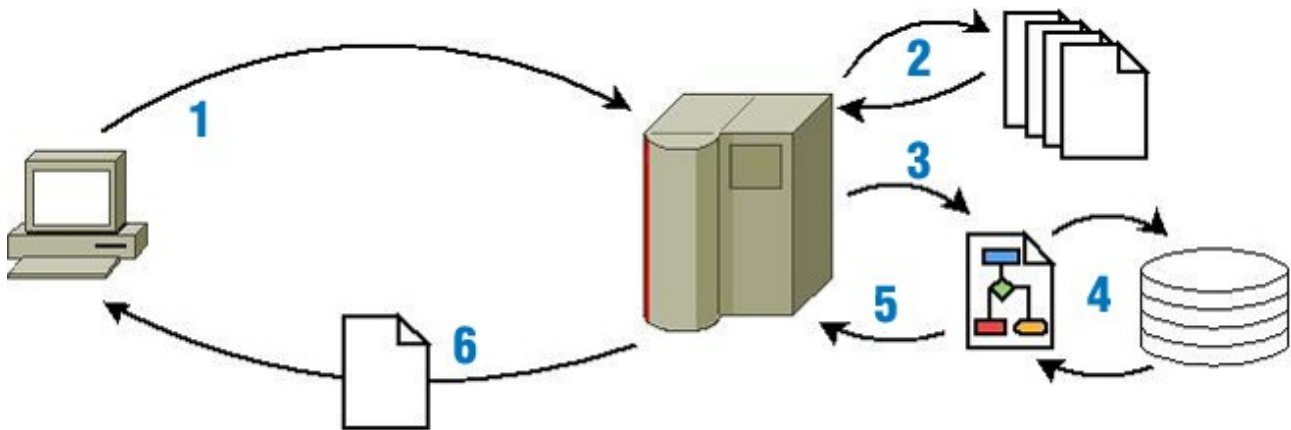
Páginas web estáticas y dinámicas

Las páginas que viste en el ejemplo anterior se llaman páginas web estáticas. Estas páginas se encuentran almacenadas en su forma definitiva, tal y como se crearon, y su contenido no varía. Son útiles para mostrar una información concreta, y mostrarán esa misma información cada vez que se carguen. La única forma en que pueden cambiar es si un programador la modifica y actualiza su contenido.

En contraposición a las páginas web estáticas, como ya te imaginarás, existen las páginas web dinámicas. Estas páginas, como su nombre indica, se caracterizan porque su contenido cambia en función de diversas variables, como puede ser el navegador que estás usando, el usuario con el que te has identificado, o las acciones que has efectuado con anterioridad.

Dentro de las páginas web dinámicas, es muy importante distinguir dos tipos:

- Aquellas que **incluyen código que ejecuta el navegador**. En estas páginas el código ejecutable, normalmente en **lenguaje JavaScript**, se incluye dentro del HTML (o XHTML) y se descarga junto con la página. Cuando el navegador muestra la página en pantalla, ejecuta el código que la acompaña. Este código puede incorporar múltiples funcionalidades que pueden ir desde mostrar animaciones hasta cambiar totalmente la apariencia y el contenido de la página. En este módulo no vamos a ver JavaScript, salvo cuando éste se relaciona con la programación web del lado del servidor.
- Como ya sabes, hay muchas páginas en Internet que no tienen extensión .htm, .html o .xhtml. Muchas de estas páginas tienen extensiones como .php, .asp, .jsp, .cgi o .aspx. En éstas, el contenido que se descarga al navegador es similar al de una página web estática: HTML (o XHTML). Lo que cambia es la forma en que se obtiene ese contenido. Al contrario de lo que vimos hasta ahora, esas páginas no están almacenadas en el servidor; más concretamente, el contenido que se almacena no es el mismo que después se envía al navegador. **El HTML de estas páginas se forma como resultado de la ejecución de un programa**, y esa ejecución tiene lugar en el servidor web (aunque no necesariamente por ese mismo servidor).



Pasos:

1. El **cliente web** (navegador) de tu ordenador **solicita** a un servidor web una **página web**.
2. El **servidor busca** esa página y la recupera.
3. En el caso de que se trate de una **página web dinámica**, es decir, que su contenido deba ejecutarse para obtener el HTML que se devolverá, el **servidor web** contacta con el módulo responsable de **ejecutar el código** y se lo envía.
4. Como parte del **proceso de ejecución**, puede ser necesario obtener información de algún [repositorio](#), como por ejemplo consultar registros almacenados en una base de datos.
5. El **resultado** de la ejecución será una página en **formato HTML**, similar a cualquier otra página web no dinámica.
6. El **servidor web envía el resultado** obtenido al navegador, que la procesa y muestra en pantalla.

Observar el punto 6, el navegador procesa el resultado, pero en esta parte hay que incluir que pudiera haber código javascript. Así el servidor envía un resultado y ese resultado puede incluir líneas de código fuente javascript ejecutables en el navegador del cliente.

En el momento que el cliente recibe el resultado se podría desconectar el ordenador de la red y la página con código javascript incluido seguiría ejecutandose Veamos un ejemplo:

Crear un fichero llamado: /var/www/html/estaticajavascript.html

y en su interior poner:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Título de la página</title>
  </head>

  <body>
    <p>
      Esta es una página con algo de javascript
    </p>
    <button onClick="saludar()">Púlsame</button>
  </body>
  <script>
    function saludar(){
      alert("hola amigo");
    }
  </script>
</html>
```

Accedemos con el navegador mediante:

<http://localhost/estaticajavascript.html>

Cada vez que pulsemos en el botón mostrará un mensaje de alerta. Ese código del botón es del lado del cliente (mediante javascript) En la anterior página **NO HAY NADA** de programación del lado del servidor. Es un fichero estático que se descarga el cliente

Tecnologías para programación web del lado del servidor.

Los **componentes principales** con los que debes contar para ejecutar aplicaciones web en un servidor son los siguientes:

- Un **servidor web** para recibir las peticiones de los clientes web (normalmente navegadores) y enviarles la página que solicitan (una vez generada puesto que hablamos de páginas web dinámicas). El servidor web debe conocer el procedimiento a seguir para generar la página web: qué módulo se encargará de la ejecución del código y cómo se debe comunicar con él.
- El **módulo encargado de ejecutar el código** o programa y generar la página web resultante. Este módulo debe integrarse de alguna forma con el servidor web, y dependerá del lenguaje y tecnología que utilicemos para programar la aplicación web.
- Una **aplicación de base de datos**, que normalmente también será un servidor. Este módulo no es estrictamente necesario pero en la práctica se utiliza en todas las aplicaciones web que utilizan grandes cantidades de datos para almacenarlos.
- El **lenguaje de programación** que utilizarás para desarrollar las aplicaciones.

La primera elección que harás antes de comenzar a programar una aplicación web es la arquitectura que vas a utilizar. Hoy en día, puedes elegir entre:

- Java EE (Enterprise Edition), que antes también se conocía como J2EE. Es una plataforma orientada a la programación de aplicaciones en lenguaje Java. Puede funcionar con distintos gestores de bases de datos, e incluye varias librerías y especificaciones para el desarrollo de aplicaciones de forma modular.

Está apoyada por grandes empresas como Sun y Oracle, que mantienen Java, o IBM. Es una buena solución para el desarrollo de aplicaciones de tamaño mediano o grande. Una de sus principales ventajas es la multitud de librerías existentes en ese lenguaje y la gran base de programadores que lo conocen. Dentro de esta arquitectura existen distintas tecnologías como las páginas JSP y los servlets, ambos orientados a la generación dinámica de páginas web, o los EJB, componentes que normalmente aportan la lógica de la aplicación web.

- AMP. Son las siglas de Apache, MySQL y PHP/Perl/Python. Las dos primeras siglas hacen referencia al servidor web (Apache) y al servidor de base de datos (MySQL). La última se corresponde con el lenguaje de programación utilizado, que puede ser PHP, Perl o Python, siendo PHP el más empleado de los tres.

Dependiendo del sistema operativo que se utilice para el servidor, se utilizan las siglas LAMP (para Linux), WAMP (para Windows) o MAMP (para Mac). También es posible usar otros componentes, como el gestor de bases de datos PostgreSQL en lugar de MySQL.

Todos los componentes de esta arquitectura son open source. Es una plataforma de programación que permite desarrollar aplicaciones de tamaño pequeño o mediano con un aprendizaje sencillo. Su gran ventaja es la gran comunidad que la soporta y la multitud de aplicaciones de código libre disponibles.

- CGI/Perl. Es la combinación de dos componentes: Perl, un potente lenguaje de código libre creado originalmente para la administración de servidores, y CGI, un estándar para permitir al servidor web ejecutar programas genéricos, escritos en cualquier lenguaje (también se pueden utilizar por ejemplo C o Python), que devuelven páginas web (HTML) como resultado de su ejecución. Es la más primitiva de las arquitecturas que comparamos aquí. El principal inconveniente de esta combinación es que CGI es lento, aunque existen métodos para acelerarlo. Por otra parte, Perl es un lenguaje muy potente con una amplia comunidad de usuarios y mucho código libre disponible.
- ASP.Net es la arquitectura comercial propuesta por Microsoft para el desarrollo de aplicaciones. Es la parte de la plataforma .Net destinada a la generación de páginas web dinámicas. Proviene de la evolución de la anterior tecnología de Microsoft, ASP.

El lenguaje de programación puede ser Visual Basic.Net o C#. La arquitectura utiliza el servidor web de Microsoft, IIS, y puede obtener información de varios gestores de bases de datos entre los que se incluye, como no, Microsoft SQL Server.

Una de las mayores ventajas de la arquitectura .Net es que incluye todo lo necesario para el desarrollo y el despliegue de aplicaciones. Por ejemplo, tiene su propio entorno de desarrollo, Visual Studio, aunque hay otras opciones disponibles. La mayor desventaja es que se trata de una plataforma comercial de código propietario.

Lenguajes compilados a código intermedio. Java , Asp.Net

Son lenguajes en los que el código fuente original se traduce a un código intermedio, independiente del procesador, antes de ser ejecutado. Es la forma en la que se ejecutan por ejemplo las aplicaciones programadas en Java, y lo que hace que puedan ejecutarse en varias plataformas distintas.

En la programación web, operan de esta forma los lenguajes de las arquitecturas Java EE (servlets y páginas JSP) y ASP.Net.

En la plataforma ASP.Net y en muchas implementaciones de Java EE, se utiliza un procedimiento de compilación JIT. Este término hace referencia a la forma en que se convierte el código intermedio a código binario para ser ejecutado por el procesador. Para acelerar la ejecución, el compilador puede traducir todo o parte del código intermedio a código nativo cuando se invoca a un programa. El código nativo obtenido suele almacenarse para ser utilizado de nuevo cuando sea necesario.

Para poder ejecutar aplicaciones Java EE en un servidor básicamente tenemos dos opciones: servidores de aplicaciones, que implementan todas las tecnologías disponibles en Java EE, y contenedores de servlets, que soportan solo parte de la especificación. Dependiendo de la magnitud de nuestra aplicación y de las tecnologías que utilice, tendremos que instalar una solución u otra.

Una vez instalada la solución que hayamos escogido, tenemos que integrarla con el servidor web que utilicemos, de tal forma que reconozca las peticiones destinadas a servlets y páginas JSP y las redirija. Otra opción es utilizar una única solución para páginas estáticas y páginas dinámicas. Por ejemplo, el contenedor de servlets Tomcat incluye un servidor HTTP propio que puede sustituir a Apache.

Comparativa script, CGI, código intermedio

Cada una de estas formas de ejecución del código por el servidor web tiene sus ventajas e inconvenientes.

- Los lenguajes de guiones tienen la ventaja de que no es necesario traducir el código fuente original para ser ejecutados, lo que aumenta su portabilidad. Si se necesita realizar alguna modificación a un programa, se puede hacer en el momento. Por el contrario el proceso de interpretación ofrece un peor rendimiento que las otras alternativas.
- Los lenguajes compilados a código nativo son los de mayor velocidad de ejecución, pero tienen problemas en lo relativo a su integración con el servidor web. Son programas de propósito general que no están pensados para ejecutarse en el entorno de un servidor web. Por ejemplo, no se reutilizan los procesos para atender a varias peticiones: por cada petición que se haga al servidor web, se debe ejecutar un nuevo proceso. Además los programas no son portables entre distintas plataformas.
- Los lenguajes compilados a código intermedio ofrecen un equilibrio entre las dos opciones anteriores. Su rendimiento es muy bueno y pueden portarse entre distintas plataformas en las que exista una implementación de la arquitectura (como un contenedor de servlets o un servidor de aplicaciones Java EE).

Java Web

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.

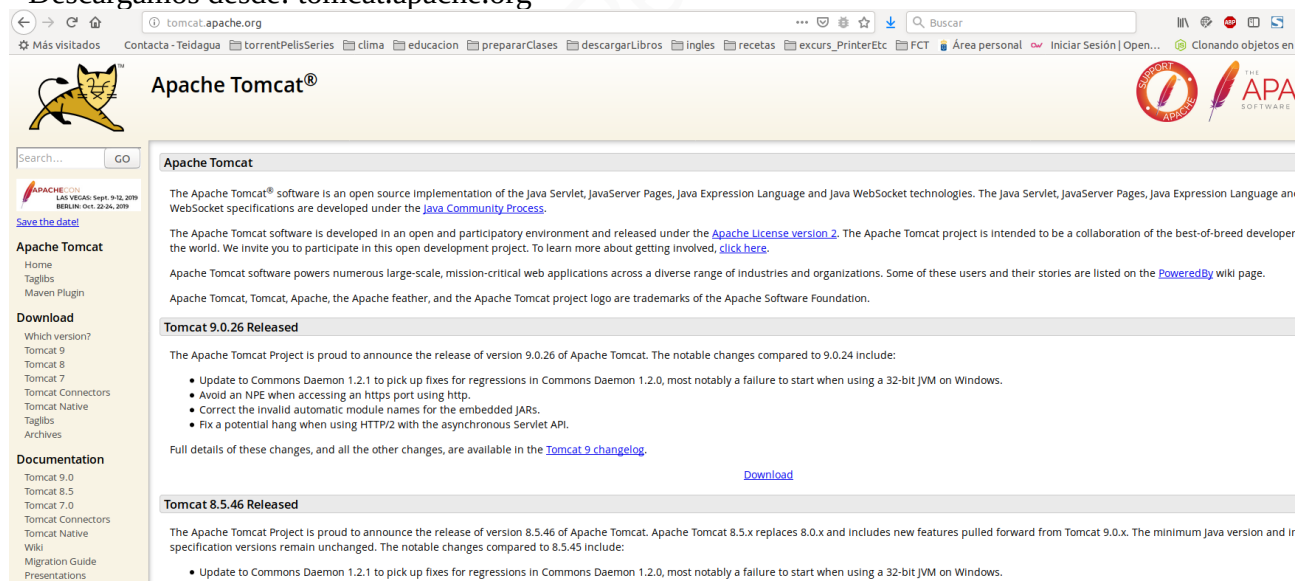
Los componentes web pueden ser servlets o páginas creadas con tecnologías JSP o JavaServer Faces.

- **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.

Instalación tomcat y creación proyecto 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 main content area is titled "Apache Tomcat" and describes the software as an open source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. It mentions that the software is developed under the [Java Community Process](#) and released under the [Apache License version 2](#). The website also lists several users and organizations that use Tomcat, including the [PoweredBy](#) wiki page.

The "Download" section is highlighted, showing the "Tomcat 9.0.26 Released" announcement. The announcement states that 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](#). A "Download" link is provided.

The "Tomcat 8.5.46 Released" section is also visible, stating that 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:

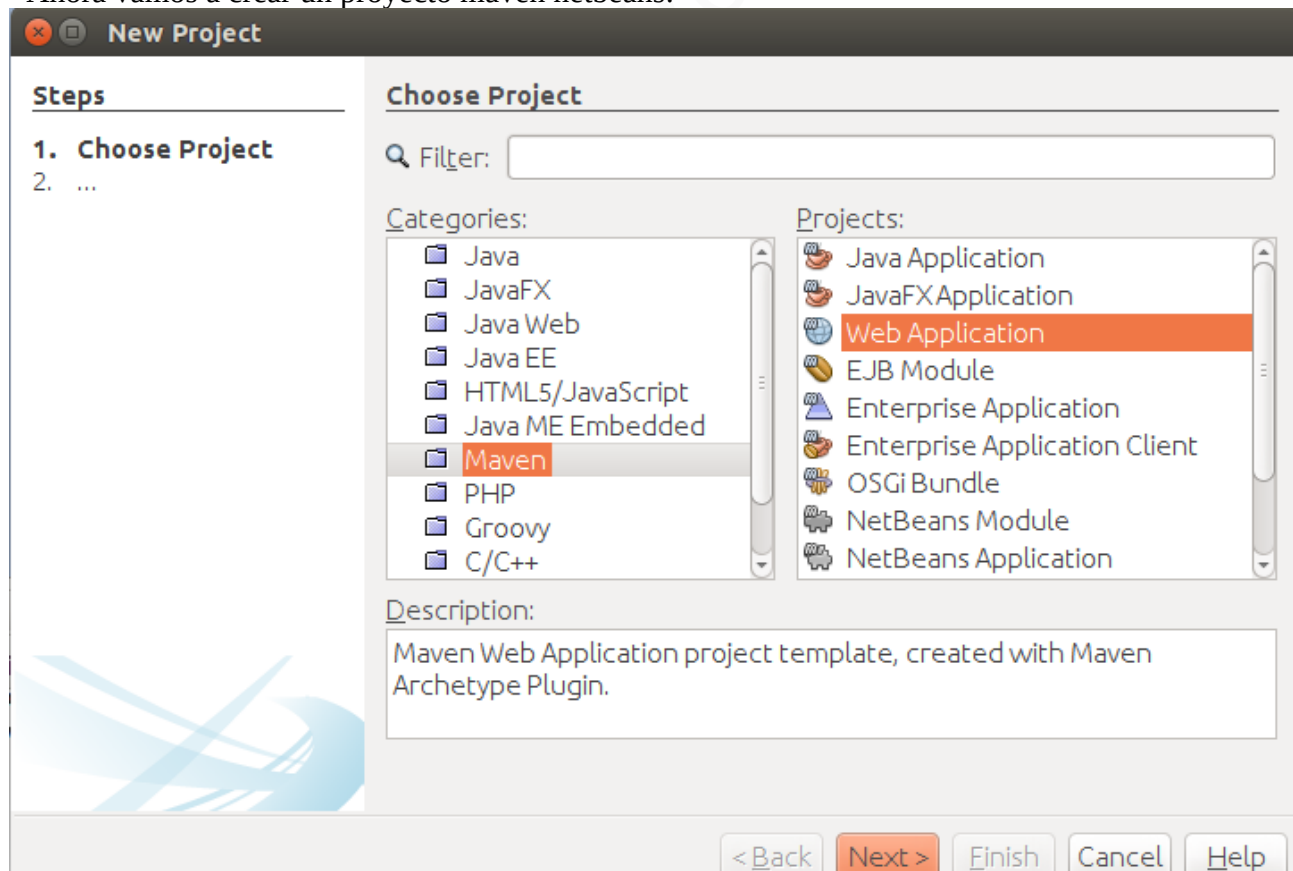
* Arrancar: /home/usuario/tomcat/bin/startup.sh

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

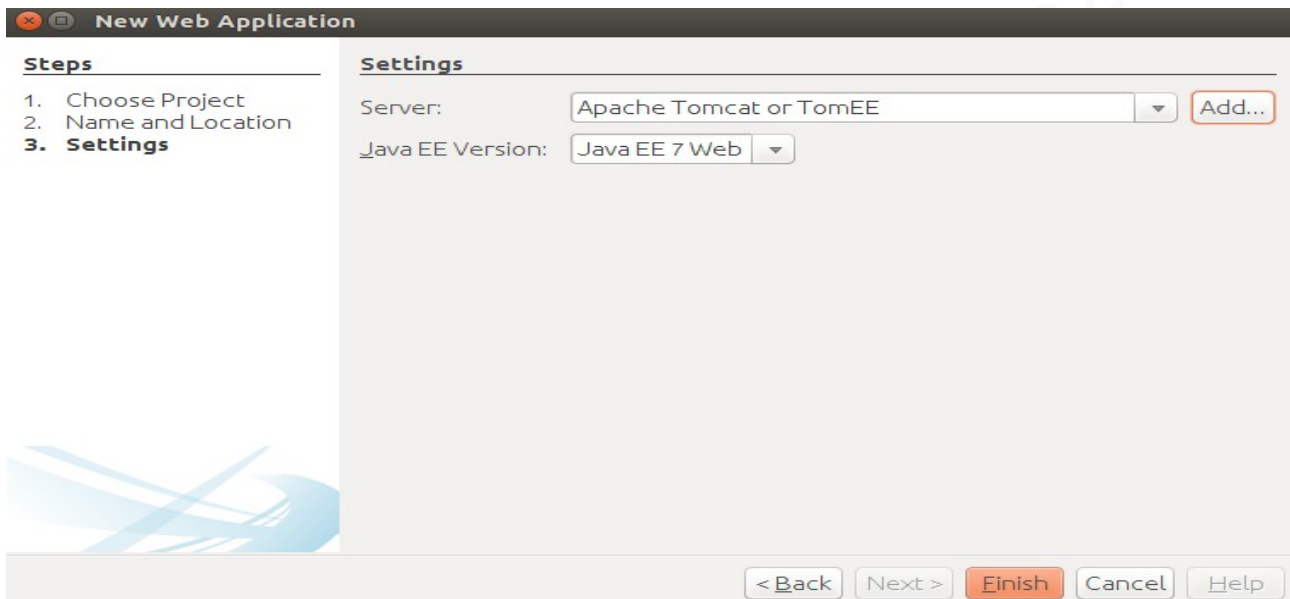
- Para acceder al tomcat abrimos el navegador y vamos a:

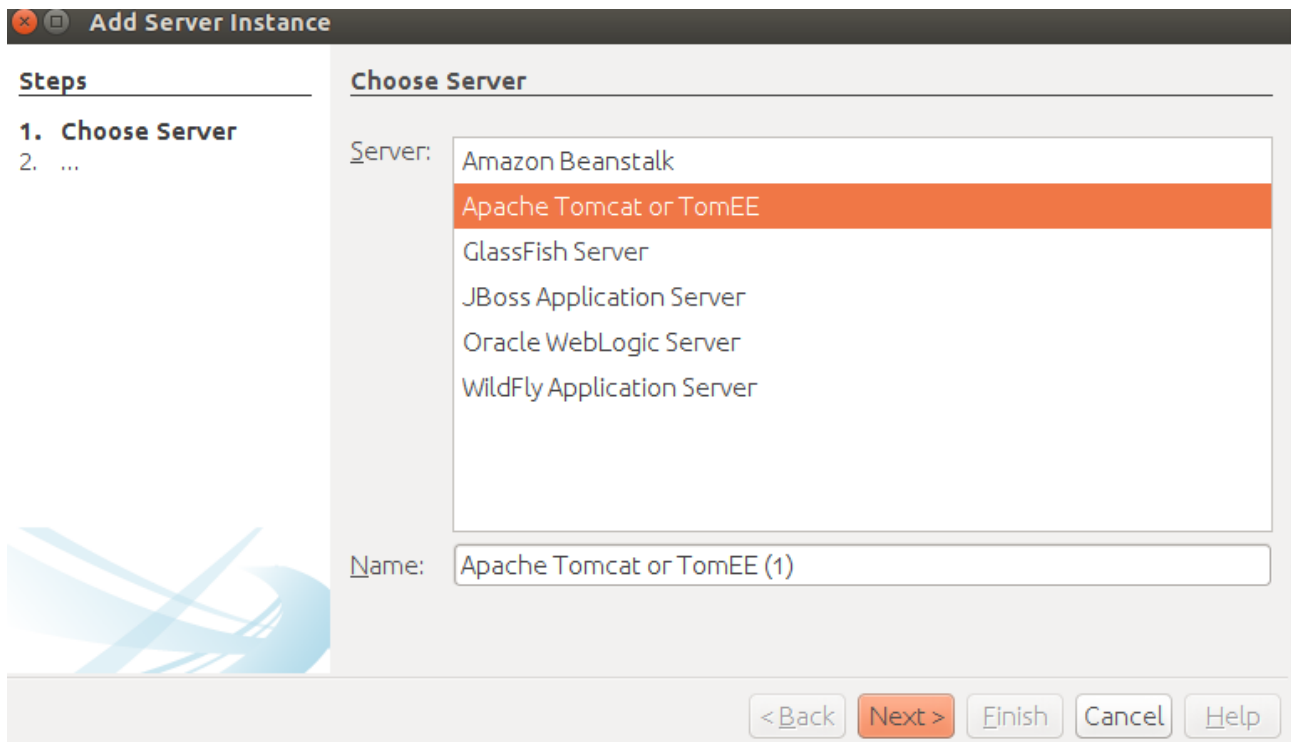
<http://localhost:8080>

- 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

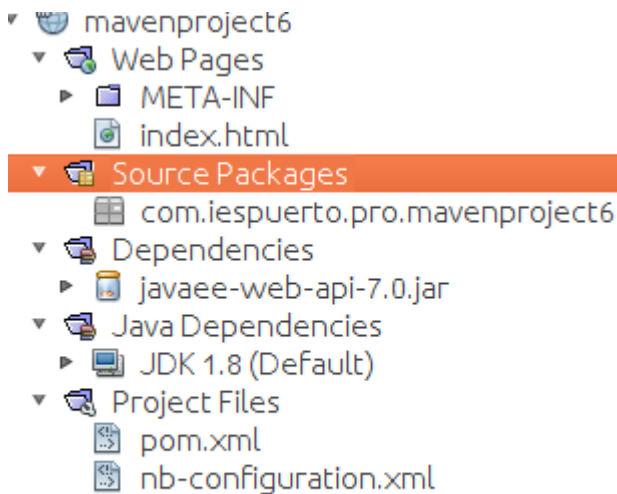




Observar que también le damos el usuario y la contraseña para acceder a tomcat



Veamos la posible estructura del proyecto creado:



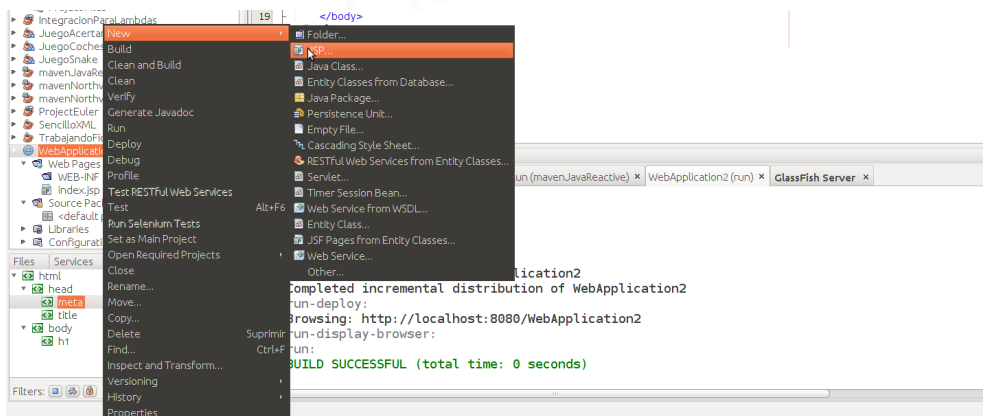
Vemos que después de crearnos el proyecto nos abre una página: index.html

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

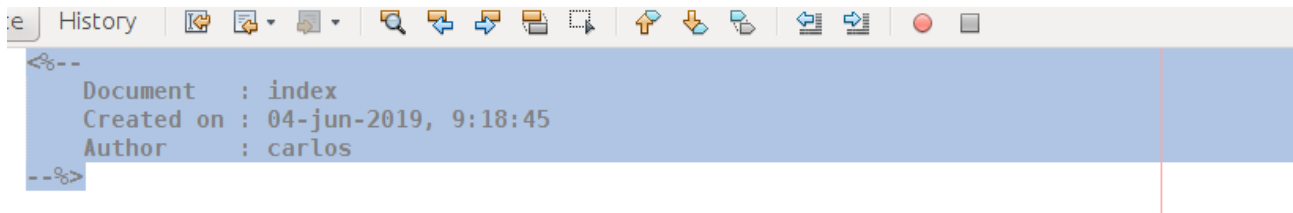
Ahora bien, esa página no es activa, es una página estática

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:

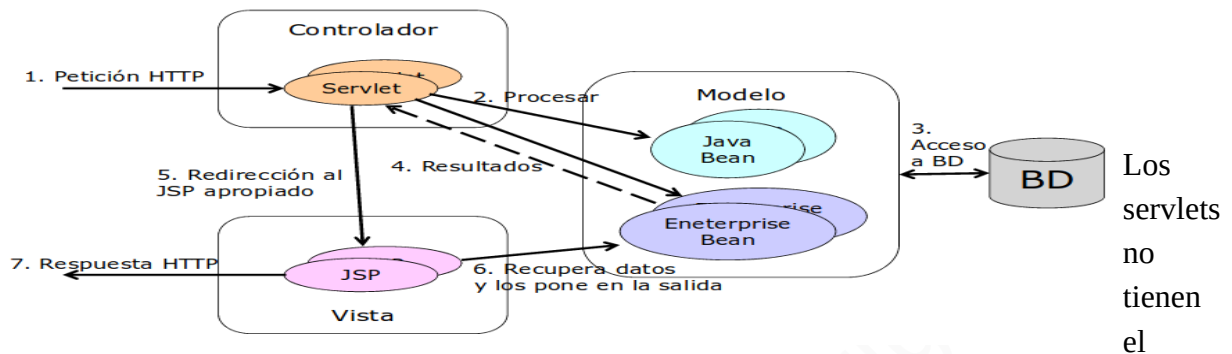
```
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
/**  
 *  
 * @author carlos  
 */  
public class NewServlet extends HttpServlet {
```

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



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:

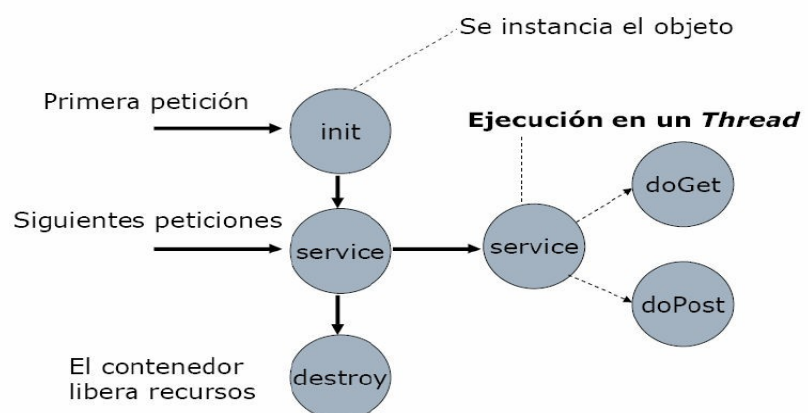
INICIALIZACIÓN: Una única llamada al método “init” por parte del servlet. Incluso se pueden recoger unos parámetros concretos con “getInitParameter” de “ServletConfig” iniciales y que operarán a lo largo de toda la vida del servlet.

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

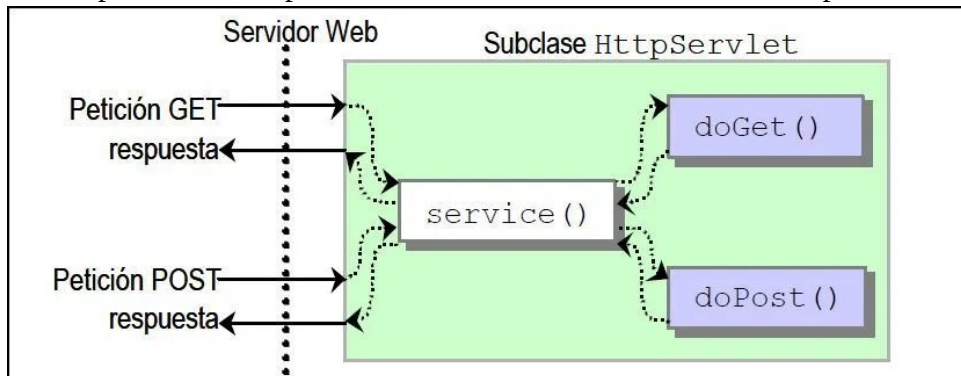
DESTRUCCIÓN: Cuando todas las llamadas desde el cliente cesen o un temporizador del servidor así lo indique o el propio administrador así lo decida se destruye el servlet. Se usa el método “destroy” para eliminar al servlet y para “recoger sus restos” (garbage collection).

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

Ciclo de Vida

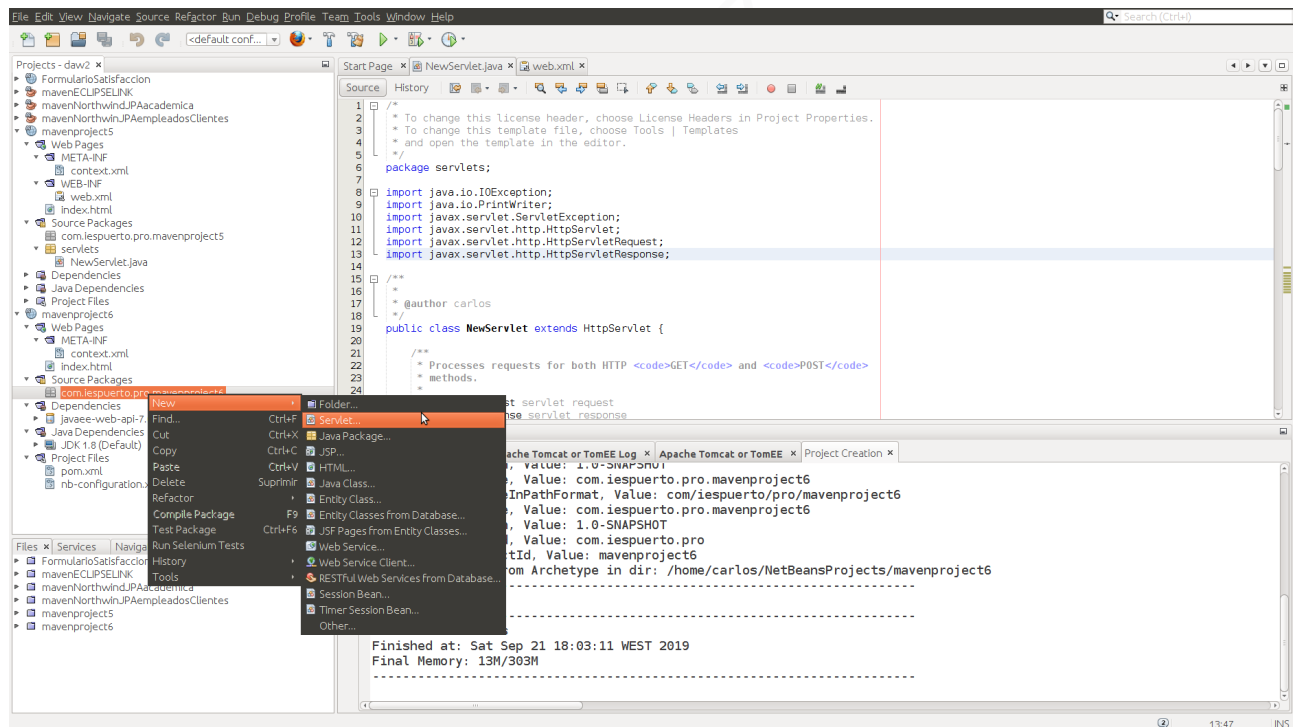


petición al servidor web. Estos dos métodos son **doGet()** y **doPost()** y nos sirven para atender las peticiones específicamente provenientes de métodos GET o POST respectivamente,

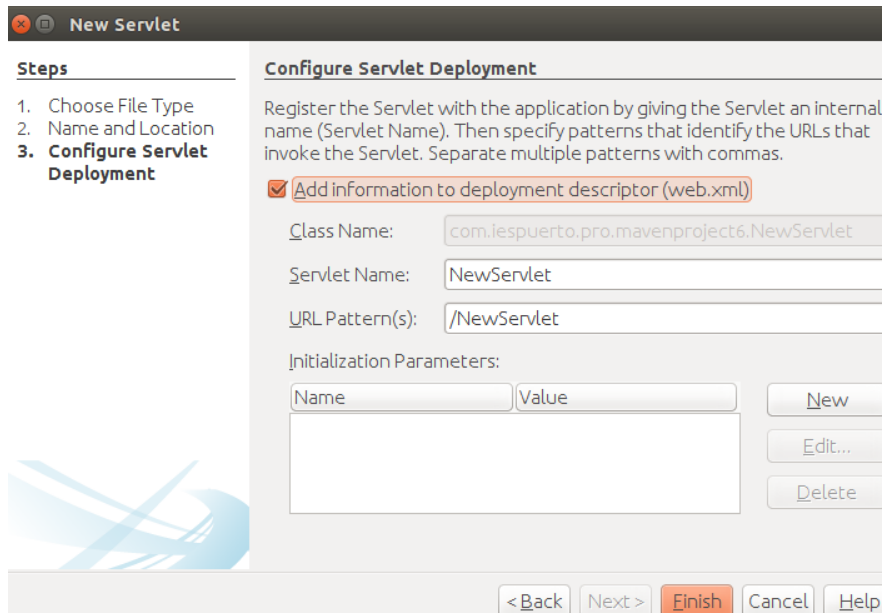


Veamos que tenemos en un servlet creado con netbeans:

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>");
        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

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 %>
```

 **Práctica 1:** Realizar la aplicación que se acaba de describir

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:

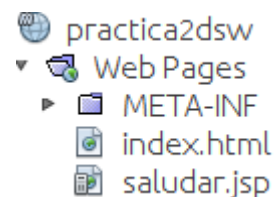
```
for( String parametro: request.getParameterMap().keySet()){  
    for(String valor: mapa.get(parametro)){  
        // aquí tenemos en: parametro el nombre del parámetro y en: valor los valores  
    }  
}
```

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:

J

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

● **Práctica 3:** hacer una página estática: index.html con un formulario GET:

nombre,
apellido

Se recibe en un servlet y este devuelve un html que dice:
tu nombre completo es: ...

● **Práctica 4:** modificar la anterior y que ahora también soporte POST.
A diferencia del GET ahora dirá:
Formulario recibido por POST. Tu nombre completo es: ...

● **Práctica 5:** Crear en index.html un enlace al servlet que procesa el nombre y apellido (el mismo al que apuntaba el formulario) y el enlace lleve ya un parámetro nombre y otro apellido (Ej. procesarformulario?nombre=paco&apellido=atta)
¿ Resuelve correctamente el servlet ? ¿ si es así, desencadena post, get ?

● **Práctica 6:** Crear una página con un formulario (puede ser en index.html u otro) para calcular imc (nombre, apellido, altura, peso) Se enviará a servlet que creará un objeto persona con esos datos y devolverá un html informando de un toString() de Persona que incluya el imc

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 7:** 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

Ahora vamos a realizar una actividad en la que hay un JSP con un formulario de persona (nombre, apellido, edad,..), agregar a un fichero los datos en formato csv

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 8:** En la actividad que calculábamos el imc redirigir a un jsp y enviar como atributo el objeto persona (recordar que debe ser un java bean) Mostrar el resultado correctamente formateado en el JSP Realizar la aplicación usando **EL** en el JSP

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:

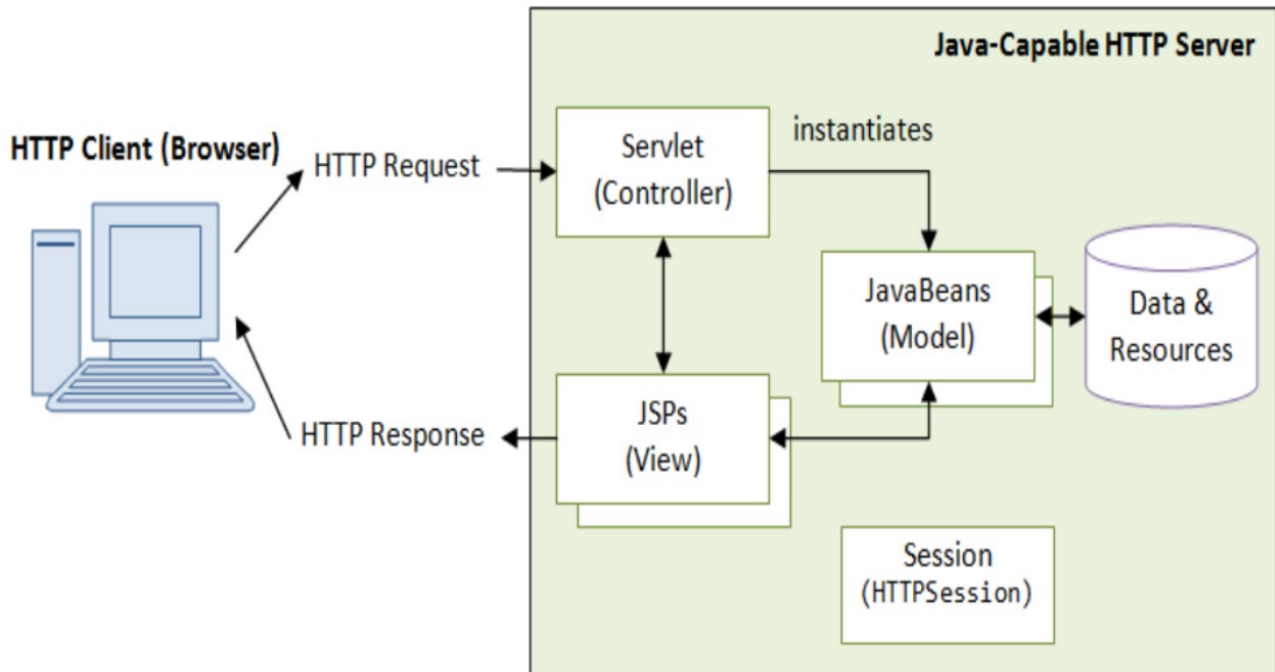
```
$ {(x, y) -> x + y}(3, 5.5)
```

Ejemplos y descripción más completa en:

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

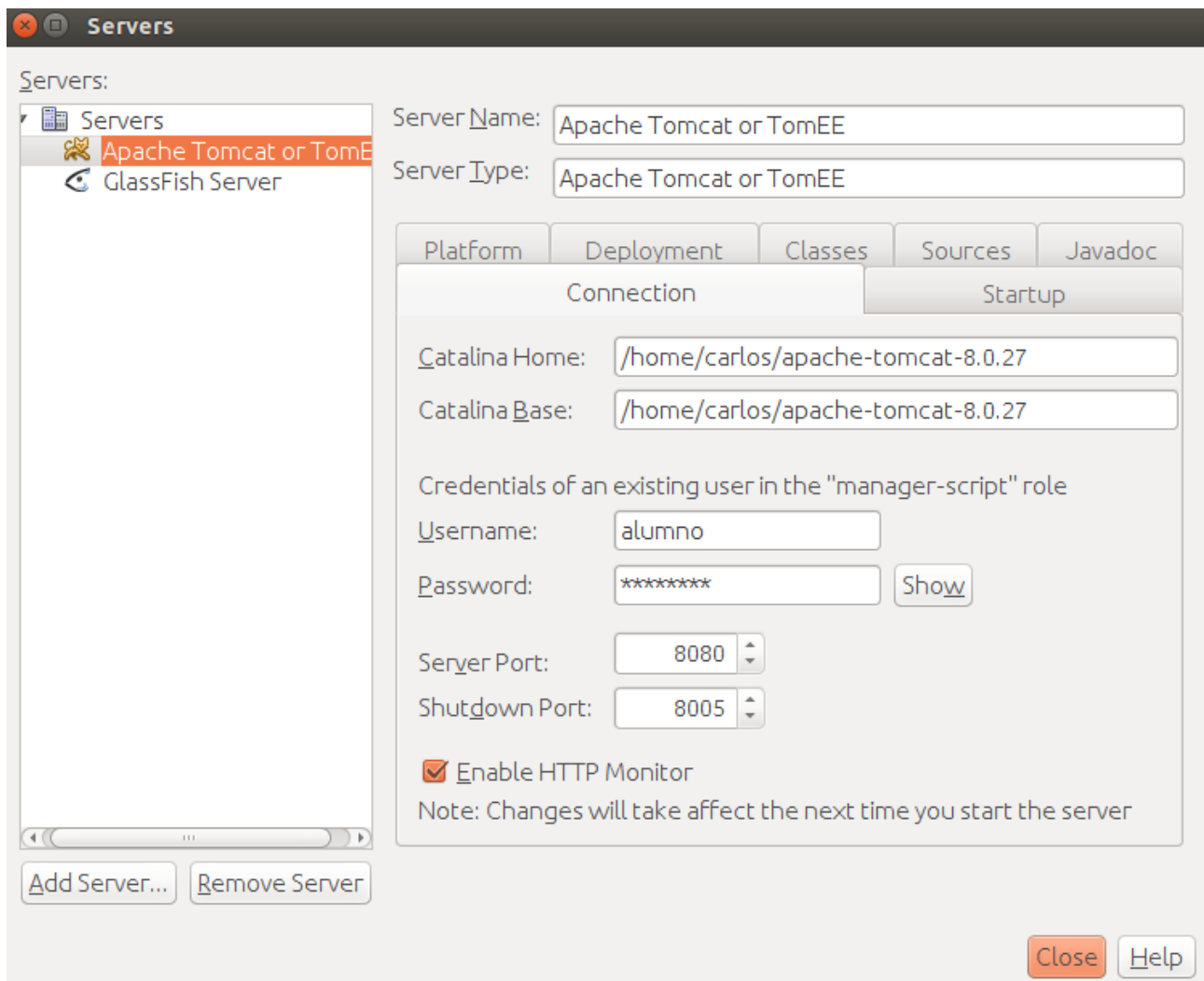
MVC con JavaBeans

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)



- **Práctica 9:** Hacer una página de login y que un servlet procese la información post si el usuario se llama admin y como clave puso: 1q2w3e4r Luego se redirigirá a un JSP y se le mostrará: OK! Bienvenido si todo ok.
si no coincide el par usuario contraseña un mensaje: NAK! Inténtelo de nuevo.
También se incluirá un enlace para ir a la página de inicio

Uso de JSLT

Sabemos que las páginas JSP nos permiten realizar las vistas de nuestra aplicación y ejecutar código mediante los scriptlets: `<% %>`

Ahora bien, si trabajamos de esa forma puede resultar incómodo tener que poner trozos de html dentro de los scriptlets para hacer un for, al igual que con un if. Veamos un ejemplo con un for que recorriera la lista de productos que se le muestra al usuario en la aplicación de Carrito:

```
<%
    for(Producto p : (ArrayList<Producto>) request.getSession().getAttribute("productos")) {
%>

        <div class="elemento">
            
            <h3><a href="gestorcarrito?agregar=<%=p.getIdproducto()%>"><%=p.getNombre()%></a></h3>
        </div>

<%
    }
%>
```

El código anterior funciona pero implica una visión no muy grata en la que se está mezclando código Java con HTML. Una solución es el uso de: JSTL

Según wikipedia:

La tecnología JavaServer Pages Standard Tag Library (JSTL) es un componente de Java EE. Extiende las ya conocidas JavaServer Pages (JSP) proporcionando cuatro bibliotecas de etiquetas (Tag Libraries) con utilidades ampliamente utilizadas en el desarrollo de páginas web dinámicas.

Veamos un ejemplo de que se puede hacer con JSTL:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<p><h1>Customer Names</h1></p>

<c:forEach items="${addresses}" var="address">
  <c:choose>
    <c:when test="${not empty address.lastName}" >
      <c:out value="${address.lastName}"/><br/>
    </c:when>
    <c:otherwise>
      N/A<br/>
    </c:otherwise>
  </c:choose><br/>
</c:forEach><br/>
```

En el código anterior observamos:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Es necesario incluir esa línea al principio de nuestra página JSP para el uso de la librería.

También podemos observar que tomamos una lista con el nombre: addresses (por ejemplo enviado como atributo de sesión) y le decimos que nos guarde cada objeto de la lista en la variable “address”

Si recordamos como funciona XSLT nos resultará similar el elemento choose. Si nos fijamos viene a ser similar a un IF-ELSE de código

Y al igual que en XSLT el IF del que dispone no tiene ELSE:

```
<c:if test="${usuario != null}">

  ${usuario.getNombre()}

</c:if>
```

Para poder hacer uso de JSTL debemos incluir en maven la siguiente dependencia:

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
```

- **Práctica 10:** Crear la ruta: {rutabaseaplicacion}/mostrarlistado
En esa ruta un Servlet leerá un fichero csv con los nombre y apellido de los alumnos de la clase (poner mínimo 5) Un campo es el nombre y otro el apellido. El servlet los pondrá en una lista: List<Alumno> donde Alumno es una clase con los atributos nombre, apellido
Luego redirigirá a una página jsp. En la página hacer uso de JSTL para que tome la lista y lo muestre en jsp Usar una lista no numerada para mostrar cada alumno
- **Práctica 11:** Crear una aplicación web que permita guardar y recuperar la información de alumnos. En esta versión habrá una página formulario para poder agregar alumnos (nombre apellido, id, curso) y un visor de los alumnos que tenemos apuntados. Adicionalmente se hará un buscador de alumno por id (el id queda a tu decisión que sea un cial, dni,...)
La información quedará guardada en un fichero XML o JSON. Se debe tener una correcta separación MVC

Enlazar un fichero CSS o JAVASCRIPT a un JSP

Supongamos que tenemos la siguiente organización de nuestro proyecto:



Como vemos en el raíz tenemos: index.jsp y también una carpeta llamada css. Entonces la forma de enlazarlo desde index.jsp sería:

```
<!DOCTYPE html >
<html>
    <head>
        <meta charset='utf-8' />
        <meta name='author' content='juan carlos p.r.'/>
        <meta name='viewport' content='width=device-width, initial-scale=1.0' />
        <title>IMC</title>
        <link href="css/estilos.css" rel="stylesheet" type="text/css">
```

Otra forma que nos permite obtener la dirección raíz y poder poner la ruta completa sería:

```
<!DOCTYPE html >
<html>
    <head>
        <meta charset='utf-8' />
        <meta name='author' content='juan carlos p.r.'/>
        <meta name='viewport' content='width=device-width, initial-
scale=1.0' />
        <title>IMC</title>
        <!--
        <link href="css/estilos.css" rel="stylesheet" type="text/css">
        -->
        <link rel="stylesheet" href="{pageContext.request.contextPath}/css/estilos.css" />
```

Observar que hemos hecho uso de expression language (EL)

Por último, aunque los estilos en general se considere, que no es preciso ponerlos en WEB-INF veamos como sería:

Supongamos la siguiente estructura para la ubicación del CSS



```
<!DOCTYPE html >
<html>
    <head>

        <meta charset='utf-8' />
        <meta name='author' content='juan carlos p.r.' />
        <meta name='viewport' content='width=device-width, initial-
scale=1.0' />

        <title>IMC</title>
        <!--
        <link href="css/estilos.css" rel="stylesheet" type="text/css">

                                <link rel="stylesheet" href="$
{pageContext.request.contextPath}/css/estilos.css" />
        -->
        <style><%@include file="/WEB-INF/css/estilos.css"%></style>

    </head>
```

Observar la directiva `@include file` de esa forma primero incorpora el css y si nos fijamos queda dentro de dos etiquetas style: `<style> ... </style>`

Práctica 12: En la actividad anterior de los alumnos en fichero csv, Ahora crear un fichero css que haga que la lista tenga un ancho de 400rem y un color de fondo de tonalidad gris. Hacer que las filas impares de la lista tengan un color de fondo blanco. Los estilos estarán en un fichero: `misestilos.css` que estará en una carpeta `css`. Observar que entre comentarios arriba tienes un ejemplo de como llamar con `pageContext` a esos estilos

Sesiones

En Java EE la sesión se representa mediante un objeto `HttpSession`. La API proporciona varios mecanismos para implementar las sesiones.

Las sesiones se mantienen con cookies o reescribiendo la URL. Si un cliente no soporta cookies o tiene el soporte de cookies desactivado se utiliza la reescritura de URL.

Las sesiones tienen un tiempo de espera (timeout) de forma que si no se accede a la información de una sesión en un tiempo determinado la sesión caduca y se destruye.

También se puede terminar explícitamente una sesión.

Obtenemos una sesión mediante el método `request.getSession()`. Este método puede recibir un boolean. Veamos los diferentes casos:

```
request.getSession(), request.getSession(true)
```

En ambos casos se obtendrá la sesión actual. Si no existiera ninguna sesión la crearía

```
request.getSession(false)
```

Se obtiene la sesión actual. Devuelve null en otro caso

Así pues algo habitual es ver sentencias del estilo:

```
HttpSession session = request.getSession();
```

Luego con ese objeto sesión podemos agregarle atributos a la sesión:

```
Usuario usuario = new Usuario();
```

```
session.setAttribute("usuario", usuario);
```

En el ejemplo anterior estamos agregando un objeto usuario a la sesión y accedemos mediante la clave: "usuario"

y podemos recoger los atributos de la sesión actual:

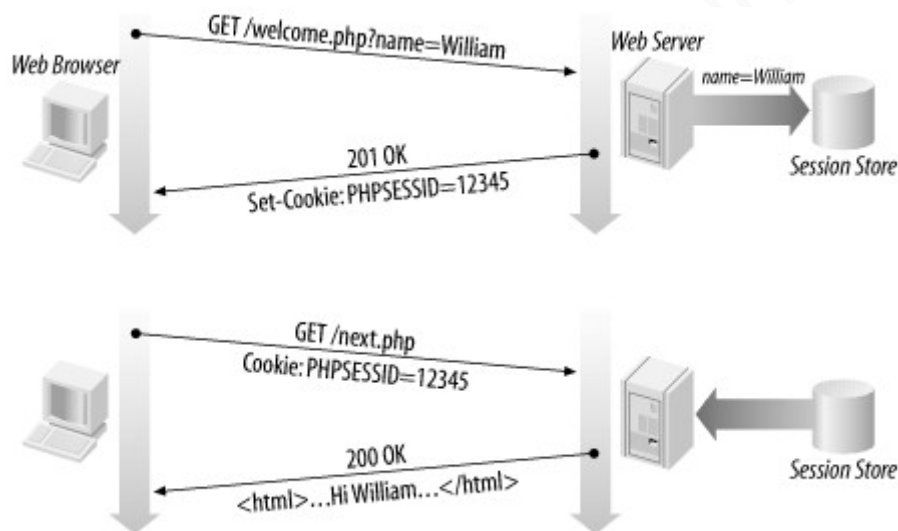
```
HttpSession session = request.getSession();  
Usuario usuario = (Usuario)session.getAttribute("usuario");
```

También podemos en cualquier momento “eliminar” la actual sesión:

```
HttpSession session = request.getSession();  
session.invalidate();
```

¿ Qué nos aporta el uso de sesiones ?

Ya hemos visto como funcionan las cookies y sabemos que se puede almacenar la información en el lado del usuario, enviando desde allí en cada consulta la información que precisa el servidor del usuario que éste ya le ha ido dando en peticiones anteriores. Ahora bien, eso significa que **en cada ocasión se está enviando más tráfico y hay que hacer más trabajo de recopilación de la información de cada petición**. Imaginemos que se guardara la información que ha ido transcurriendo de una a otra petición del cliente en el lado del servidor y **lo único que le enviáramos al servidor el un identificador**, una especie de uid, para distinguir que cosas son de un cliente respecto a otro cliente. Pues así funciona más o menos una sesión



Cuando el usuario accede a una página JSP se tiene establecido por defecto que se cree una sesión

Veamos un ejemplo:

Si miramos el servlet que se genera desde un JSP podemos ver el getSession():

```
try {
    response.setContentType("text/html;charset=UTF-8");
    pageContext = _jspxFactory.getPageContext(this, request, response,
        null, true, 8192, true);
    _jspx_page_context = pageContext;
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("\n");
    out.write("\n");
    out.write("\n");
    out.write("\n");
    out.write("\n");
    out.write("<!DOCTYPE html>\n");
    out.write("<html>\n");
    out.write("    <head>\n");
    out.write("        <meta http-equiv=\"Content-Type\" content=\"text/html; charsets
    out.write("        <title>JSP Page</title>\n");
}
```

Si en nuestro JSP incluyéramos:

```
<%@page session="false" %>
```

entonces no se crearía la cookie de sesión en el cliente

- **Práctica 13:** Crear un proyecto web en el que tengamos una página JSP visualizar tanto en HTTPMONITOR como en el navegador web que se crea la cookie de sesión al acceder a esa página (tomar captura de pantalla) y hacer lo propio estableciendo page session false comprobar que no crea la cookie de sesión y tomar captura de pantalla

Hay algunos métodos específicos de las sesiones que debemos conocer:

`session.getCreationTime()` Tiempo en milisegundos desde 1970 hasta que se creó la sesión

`session.getLastAccessedTime()` Tiempo en milisegundos desde que el cliente envió una solicitud relacionada con la sesión

`session.getId()` El id de la sesión

- **Práctica 14:** Crear un proyecto web en el que el usuario al acceder se le cree una sesión y se le muestre información de la sesión: El ID, la fecha de creación de la sesión, y el tiempo desde el último acceso. Así como un contador del número de accesos que ha realizado el cliente en la actual sesión
nota: `new Date(tiempoEnMilisegundosDesde1970)` nos da formato fecha

- **Práctica 15:** Crear un proyecto web en el que el usuario accede a `index.jsp` y se le muestra la hora actual del sistema y un mensaje que diga: “hola anónimo” Desde ahí habrá un enlace a `login.html` (también puede ser `login.jsp`, como se prefiera) donde el cliente podrá acceder con tu nombre y la password: `1q2w3e4r` Un Servlet recibe la petición de login Si todo sale bien redireccionará a `index.jsp` y ahora en lugar de “hola anónimo” dirá: “hola nombrealumno” si sale mal (no coincide user/pass) regresa a `login.html`

Nota: desde que es `index.jsp` sabemos que se crea una sesión, en el servlet se pondrá un atributo de sesión: (“usuario”, Usuario) donde la clase Usuario tiene toda la información del usuario (en este caso únicamente su nombre y password)

- **Práctica 16:** Modificar la actividad anterior para que ahora en la página: `login.jsp` se le informe al cliente cuál ha sido el error (el usuario no está registrado, la password no coincide) Crear otro login posible en esta ocasión usuario: “admin” password: “1q2w3e4r” Si ahora el cliente entra en `login.jsp` habiéndose ya logueado en la página dirá: “usuario: loquecorresponda ya está logueado. Si accede como otro usuario se perderá la actual sesión”
Si el usuario entra estando logueado se invalida (`session.invalidate()`) la sesión anterior y se inicia una nueva sesión con el nuevo usuario.
Nota: Observar que ya en este caso es importante que el acceso al login pase primero por el servlet siendo `login.jsp` meramente un formulario y mostrar los mensajes del servlet Así que el enlace de `index.jsp` hacia el login será hacia el servlet

Ámbitos de una aplicación Web

En una aplicación Web se han definidos varios ámbitos en los que se definen la visibilidad de los objetos y su duración.

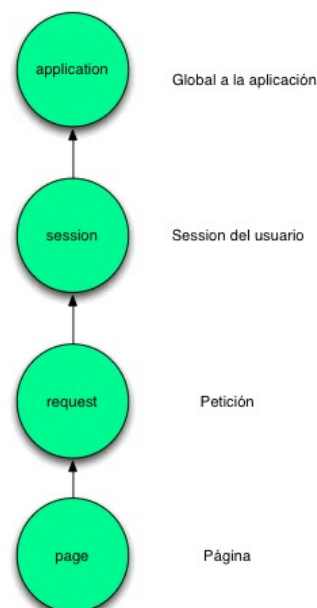
```
public class Ambitos extends HttpServlet {  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse  
response) throws ServletException, IOException {  
        //response.setContentType("text/html;charset=UTF-8");  
        ServletContext aplicacion = request.getServletContext();  
        HttpSession session = request.getSession();  
        aplicacion.setAttribute("variableAplicacion", "holaAplicacion");  
        session.setAttribute("variableSession", "holaSession");  
        request.setAttribute("variablePeticion", "holaPeticion");  
        RequestDispatcher despachador = request.getRequestDispatcher("/ambitos.jsp");  
        despachador.forward(request, response);  
    }  
}
```

- la variable aplicacion es del tipo ServletContext (ámbito aplicación)
- la variable session es del tipo HttpSession (ámbito sesión)
- la variable request es del tipo HttpServletRequest (ámbito request-solicitud>)

El scope de **Application** es compartido por todos los elementos de la aplicación (esto incluye múltiples sesiones)

el scope de **Session** pertenece a las variables que almacena cada usuario.

Por otro lado el **scope de petición** tiene un ciclo de vida mucho más corto ya que pertenece a las páginas que están ligadas a la misma petición.



Bien, en el código anterior se pudo observar como se puede acceder a los diferentes ámbitos desde un Servlet ¿ cómo se haría lo mismo desde JSP ? Mediante: **pageContext**

Para el código anterior de servlet las siguiente expresiones EL de JSP nos darían acceso a los datos:

```
${pageContext.servletContext.getAttribute("variableAplicacion")}
```

```
${pageContext.session.getAttribute("variableSession")}
```

```
${pageContext.request.getAttribute("variablePetición")}
```

También sabemos ya que existen las variables: request, session y podemos acceder al ámbito de aplicación mediante `getServletContext()`

Vamos a modificar el código del servlet anterior de esta forma:

```
public class Servlet1 extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        //response.setContentType("text/html;charset=UTF-8");
        ServletContext aplicacion = request.getServletContext();
        HttpSession session = request.getSession();
        String variableAplicacion =
(String)aplicacion.getAttribute("variableAplicacion");
        if( variableAplicacion == null)
            variableAplicacion = "";
        aplicacion.setAttribute("variableAplicacion", variableAplicacion +
"holaAplicacion dice el id: "+session.getId());
        String variableSession = (String)session.getAttribute("variableSession");
        if( variableSession == null)
            variableSession = "";
        session.setAttribute("variableSession", variableSession + " holaSession dice el
id: " + session.getId());
        request.setAttribute("variablePetición", "holaPetición");
        RequestDispatcher despachador = request.getRequestDispatcher("/ambitos.jsp");
        despachador.forward(request, response);
    }
}
```


Ámbito de página JSP

El último ámbito, el ámbito página, tiene que ver exclusivamente con páginas JSP. Objetos con ámbito página se almacenan en la instancia de `javax.servlet.jsp.PageContext` asociada a cada página y son accesibles sólo dentro de la página JSP en el que fueron creados.

Una vez que la respuesta se envía al cliente o la página remite a otros recursos, los objetos ya no estarán disponibles.

Cada página JSP tiene implícita una referencia a un objeto llamado: **pageContext** que se pueden utilizar para almacenar y recuperar objetos de ámbito página.

Tienen los mismos métodos `getAttribute()` y `setAttribute()` de otros ámbitos, y funcionan de la misma manera.

Para terminar con las sesiones refrescar y establecer su funcionamiento: Cuando el cliente hace una request hacia un servlet que ejecute: `request.getSession()` se crea una nueva sesión y se asigna un único ID para esa sesión. El servlet también establece una Cookie en la cabecera de la respuesta HTTP y en esa cookie va el ID que ha generado a la sesión. La cookie es almacenada en el navegador del cliente, y cada vez que el cliente hace las siguientes solicitudes (request) envía esa cookie informando del ID de sesión. El servlet mira en las cabeceras que envía el cliente y toma la información del ID de sesión, de esa forma toma toda la información que tiene almacenada sobre esa sesión en la memoria del servidor.

La sesión permanece activa mientras el usuario interactúe con ella. Queda establecido un "timeout" típicamente en `web.xml` por el que si el usuario sobrepasa esa cantidad inactivo se cierra la sesión. Por defecto en `web.xml` aparece 30 minutos:

```
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>NewServlet</servlet-name>
    <servlet-class>com.iespuerto.pro.maventiemposecion.NewServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet2</servlet-name>
    <servlet-class>com.iespuerto.pro.maventiemposecion.Servlet2</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>NewServlet</servlet-name>
    <url-pattern>/newServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Servlet2</servlet-name>
    <url-pattern>/Servlet2</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>newServlet</welcome-file>
  </welcome-file-list>
</web-app>
```

Si se quiere establecer un tiempo específico de inactividad para una sesión en lugar de para todas como sería si se hace uso de web.xml se puede establecer allí donde hemos ejecutado el getSession() mediante `setMaxInactiveInterval()` ejemplo:

```
HttpSession sesion = request.getSession();
sesion.setMaxInactiveInterval(60);
```

En el ejemplo queda establecida una inactividad de 60 segundos. Si en `setMaxInactiveInterval()` establecemos: 0 o un número negativo, ej: -1 entonces Java entenderá que la sesión de debe nunca expirar

Del lado del cliente el id de sesión permanece en el navegador hasta que se cierre el navegador.

Vamos a modelar una práctica en la que demos que la sesión se mantiene viva mientras el usuario interactúe pero cuando sobrepase el tiempo establecido de inactividad la sesión muere. Para ello crearemos 2 Servlet El Servlet1 es el que se arranca al iniciar la aplicación y crea una sesión, guarda en un atributo de aplicación la información del ID de sesión creado, y la fecha-hora de creación de la sesión. Devuelve un html con un enlace hacia el Servlet2. El Servlet2 muestra lo que está contenido en el atributo de aplicación y la información que tiene él almacenada sobre la sesión actual (para hacer esto hace uso de request.getSession(false) para garantizar que no crea una nueva sesión cuando una se cierre) Este servlet también agrega la información de la hora a la que fue accedido en el atributo de aplicación . Estableceremos en web.xml un tiempo de 1 minuto

La información que se muestre en el html que devuelve el servlet2 después de pulsar varias veces en refrescar podría ser:

en la aplicación tenemos guardada la siguiente información:
el id de sesión es: A8827B54FAD3BC721062F46548233F66
hora inicio: Mon Oct 21 19:43:40 WEST 2019
anterior inactivo: Mon Oct 21 19:43:40 WEST 2019
anterior inactivo: Mon Oct 21 19:43:42 WEST 2019
anterior inactivo: Mon Oct 21 19:44:02 WEST 2019
anterior inactivo: Mon Oct 21 19:44:33 WEST 2019
anterior inactivo: Mon Oct 21 19:45:19 WEST 2019

según este servlet el id de sesión es: A8827B54FAD3BC721062F46548233F66
la hora de creación de la sesión: Mon Oct 21 19:43:40 WEST 2019
la hora desde que está inactivo: Mon Oct 21 19:45:46 WEST 2019
le agregamos al atributo informacion del ámbito aplicación el lastaccessedtime()

Y cuando pase Más de ese minuto inactivo la salida del servlet2 podría ser:

en la aplicación tenemos guardada la siguiente información:
el id de sesión es: A8827B54FAD3BC721062F46548233F66
hora inicio: Mon Oct 21 19:43:40 WEST 2019
anterior inactivo: Mon Oct 21 19:43:40 WEST 2019
anterior inactivo: Mon Oct 21 19:43:42 WEST 2019
anterior inactivo: Mon Oct 21 19:44:02 WEST 2019
anterior inactivo: Mon Oct 21 19:44:33 WEST 2019
anterior inactivo: Mon Oct 21 19:45:19 WEST 2019
anterior inactivo: Mon Oct 21 19:45:46 WEST 2019

según este servlet la sesión devolvió null

 **Práctica17:** Realizar la práctica descrita arriba

welcome-file-list en web.xml

Veamos que dicen en la documentación que deja accesible google para este recurso:

Cuando las URL de tu sitio representan rutas a archivos estáticos o JSP en tu WAR, a menudo es una buena idea **que las rutas a los directorios también hagan algo útil**. Un usuario que visita la ruta de URL /help/accounts/password.jsp para obtener información sobre las contraseñas de la cuenta puede intentar visitar /help/accounts/ con el fin de encontrar una página que presente la documentación del sistema sobre la cuenta. **El descriptor de implementación puede especificar una lista de nombres de archivo que el servidor debe probar cuando el usuario accede a una ruta que representa un subdirectorío** del WAR (que no esté asignado de manera explícita a un servlet). El estándar de servlet llama a esto la "lista de archivos de bienvenida".

Por ejemplo, si el usuario accede a la ruta de URL /help/accounts/, el siguiente elemento `<welcome-file-list>` del descriptor de implementación le indica al servidor que debe verificar `help/accounts/index.jsp` y `help/accounts/index.html` antes de informar que la URL no existe:

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Vale, entonces para nuestro caso en concreto podríamos tener un Servlet detallado de la siguiente forma en `web.xml`:

```
<servlet>
  <servlet-name>GestorCarrito</servlet-name>
  <servlet-class>cotroller.GestorCarrito</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>GestorCarrito</servlet-name>
  <url-pattern>/gestorcarrito</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>gestorcarrito</welcome-file>
</welcome-file-list>
```

Así lo primero que se lanzará es el servlet

● **Práctica 18:** Crear un servlet que mire si el usuario admin con password: 1q2w3e4r está autenticado (guardamos una variable de sesión que lo informe) Este servlet será lo primero que se accederá en la aplicación por defecto (modificar el welcome-file-list) Si no lo está te lleva a la página de login. Si lo está te envía a index.jsp
En index.jsp se mostrará: bienvenido admin!
Y luego la información de datos propios: nombre alumno, apellido alumno , curso que realiza.
En el caso que no esté autenticado NO se verán los datos propios del alumno

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 19:** 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 el contenido de un fichero con datos personales (para la práctica puedes poner tu nombre apellido y curso que estudias) que esté almacenado en web-inf si no coincide el par usuario contraseña un mensaje: NAK! Inténtelo de nuevo

Application Listener

En ocasiones (especialmente cuando queremos tener establecidas las conexiones de una aplicación que precisa de bases de datos) queremos que se ejecuten cosas al inicio de la aplicación y antes de su cierre

Para conseguir lo anterior precisamos implementar: `ServletContextListener` y usar la anotación: `@WebListener` (otra alternativa es declararlo en `web.xml`)

El siguiente ejemplo es de una clase que se ejecuta al inicio de la aplicación web y no usa ningún wizard del ide para ser creada (basta con copiar y pegar el código en el package de los servlets, por ejemplo)

```
@WebListener
public class ApplicationSinWizard implements ServletContextListener{

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        Logger logger = Logger.getLogger("debug");
        logger.info("-----");
        logger.info("-----");
        logger.info("----- oh yeah! -----");
        logger.info("-----");
        logger.info("-----");
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

Lo único que hace el código es escribir mediante logger en la consola del servidor (tomcat, payara, ...) Pero podemos ver muy fácilmente que se ejecuta sin mayor problema el código al inicio.

El equivalente en `web.xml` es usar la etiqueta: `<listener>`

```
<listener>
  <description>ServletContextListener</description>
  <listener-class>es.iespuertodelacruz.jc.gestionwebalumnoconfichero.controller.servlets.AplicationListener</listener-class>
</listener>
```

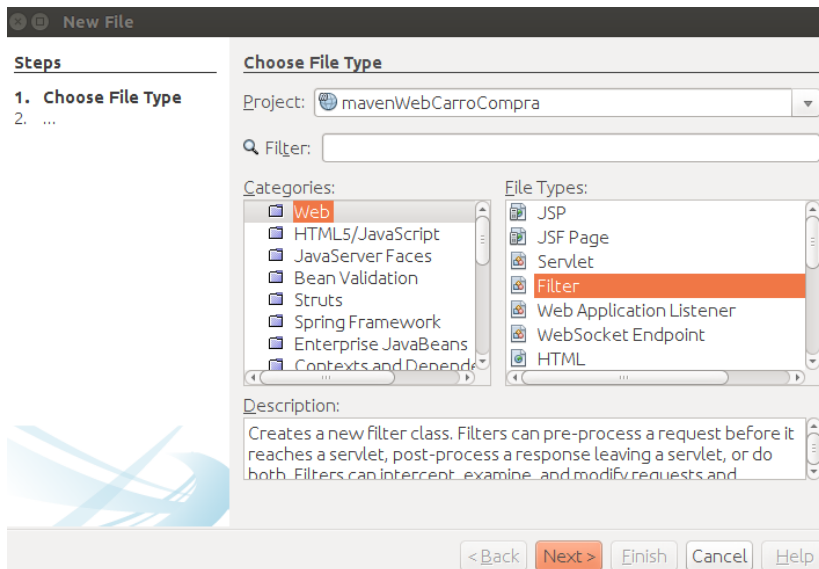

Juan Carlos Pérez Rodríguez

Filtros

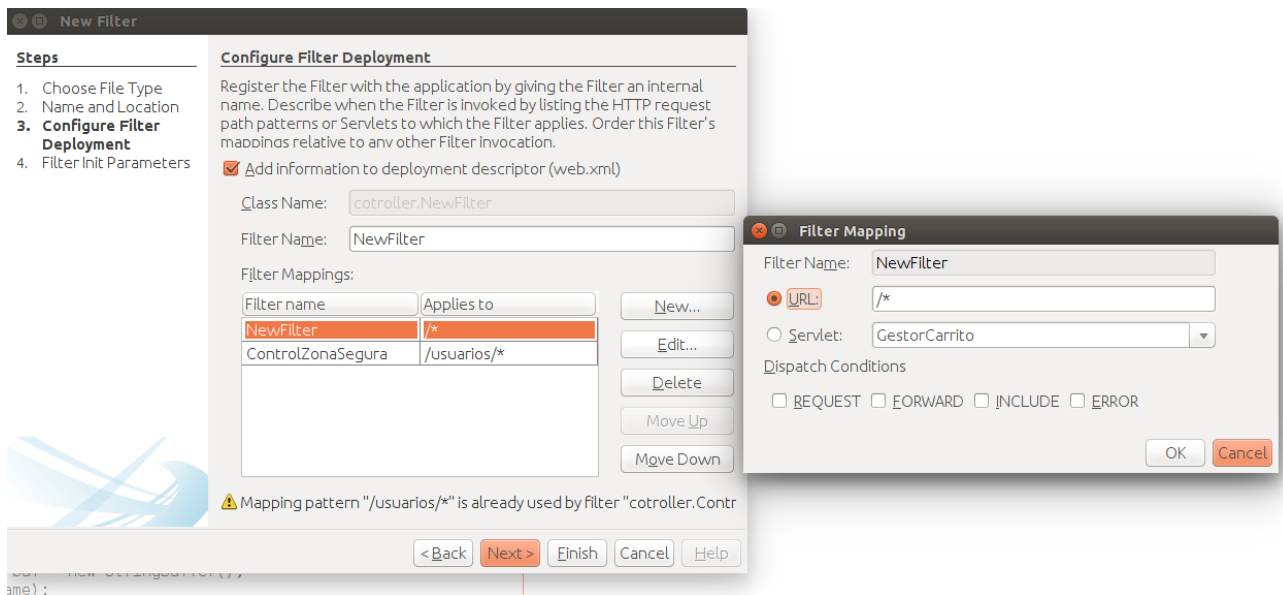
Un filtro de java se encarga de añadir una nueva funcionalidad a la aplicación colocandose entre el usuario y las páginas

Definir un filtro en Java es bastante sencillo sobre todo a partir de Servlet 3.0 que nos podemos apoyar en el sistema de anotaciones. Un Filtro implementa la interface Filter de JEE que aporte el método doFilter.

En Netbeans haríamos: New → Other → Web → Filter



Seguimos las indicaciones del IDE y llegamos a la ventana:



Observamos que le hemos dicho que establezca la información en web.xml y como vemos creamos las rutas (URL) para las que queremos que el Filtro se aplique Si hemos puesto:

`/*`

Significará que se aplica a todas las rutas

Si por ejemplo ponemos:

`/usuarios/*`

El filtro se aplica únicamente a la zona dedicada a los usuarios Tener las cosas organizadas por subcarpetas y subrutas nos ayudará a la aplicación correcta de filtros y mejor comprensión de nuestra aplicación por otros desarrolladores

En web.xml nos quedará algo parecido a:

```
<filter>
  <filter-name>F1</filter-name>
  <filter-class>controller.F1</filter-class>
</filter>

<filter-mapping>
  <filter-name>F1</filter-name>
  <url-pattern>/XYZ/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>F2</filter-name>
  <url-pattern>/XYZ/abc.do</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>F3</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

El orden que tengan los filtros en web.xml es relevante. En el caso descrito, Los filtros se ejecutaría primero F1, luego F2, y termina F3 que es el orden de primero al último establecido en web.xml

El filtro creado nos quedará parecido al siguiente

Nosotros
donde
vamos a
trabajar

```
public class ControlZonaSegura implements Filter {

    private static final boolean debug = true;

    // The filter configuration object we are associated with. If
    // this value is null, this filter instance is not currently
    // configured.
    private FilterConfig filterConfig = null;

    public ControlZonaSegura() {
    }

    private void doBeforeProcessing(ServletRequest request, ServletResponse response)
        throws IOException, ServletException {...26 lines}

    private void doAfterProcessing(ServletRequest request, ServletResponse response)
        throws IOException, ServletException {...23 lines}

    /**...9 lines */
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {

        if (debug) {
            log("ControlZonaSegura:doFilter()");
        }

        //doBeforeProcessing(request, response);

        boolean puedePasar = false;

        Throwable problem = null;
        try {

            HttpServletRequest req = (HttpServletRequest) request;
```

especialmente es en el método: doFilter()

Chequeremos lo que proceda y si corresponde enviaremos la request a otra dirección (el código siguiente estaría en el try de la imagen)

```
try {  
  
    HttpServletRequest req = (HttpServletRequest)request;  
    HttpServletResponse res = (HttpServletResponse)response;  
    HttpSession session = req.getSession();  
    // aquí serían nuestro check  
    String path = req.getServletContext().getContextPath();  
    res.sendRedirect(path+"/prueba.jsp");  
}
```

- **Práctica 20:** Crear un proyecto con varias página jsp y/o html. Una de ellas llamada: prueba.jsp Crear un filtro para todas las rutas: /* y hacer uso del código anterior. Comprobar que a cualquier acceso nos lleva a prueba.jsp. Si no funciona por qué puede ser ? Pensar en la posibilidad de llamadas recurrentes al filtro. Probar a poner en lugar del redirect un: getRequestDispatcher("prueba.jsp") ¿ ahora funciona ? Por qué motivo ?

- **Práctica 21:** Crear para la práctica anterior, en la versión con redirect, un atributo de aplicación contador. Cada vez que se ejecute el filtro aumentar ese contador. Mostrarlo mediante System.out.println() ¿ qué ha ocurrido ? ¿ cuántas veces se ejecuta el filtro ?

En el filtro también podemos crear código. Imaginemos que queremos tener un registro de las páginas que se visitan en nuestra aplicación. Podríamos individualmente en cada página registrar en una estructura de datos el conteo pero es más fácil con un filtro ya que ese sería el único sitio donde tendríamos que poner el código mientras que de otra forma habría que hacerlo por cada página

Dentro del doFilter() podríamos poner algo parecido a:

```

HttpServletRequest req = (HttpServletRequest) request;
HttpServletResponse res = (HttpServletResponse) response;
HttpSession session = req.getSession();

ServletContext contexto = request.getServletContext();
HashMap<String, Integer> urls;
if (contexto.getAttribute("estadistica") == null) {
    //creamos un objeto en el contexto
    urls = new HashMap<String, Integer>();
    urls.put(req.getRequestURL().toString(), 1);
    contexto.setAttribute("estadistica", urls);
} else {
    // actualizamos claves e incrementamos
urls = (HashMap<String, Integer>) contexto.getAttribute("estadistica");
    if (urls.get(req.getRequestURL().toString()) == null) {
        urls.put(req.getRequestURL().toString(), 1);
    } else {
urls.put(req.getRequestURL().toString(), urls.get(req.getRequestURL().toString()) + 1);
    }
}
}

```

Observar que estamos poniendo a nivel de APLICACIÓN el objeto:

```
ServletContext contexto = request.getServletContext();
```

```
contexto.setAttribute("estadistica", urls);
```

Para visualizar el objeto en cualquier JSP basta poner algo así:

```

<%
HashMap<String, Integer>
mapa=(HashMap<String, Integer>)application.getAttribute("estadistica");
Set<String> conjunto=mapa.keySet();
for (String clave :conjunto) {
    out.println("pagina : " +clave + " visitas : "+mapa.get(clave)+"<br/>");
}
%>

```

Observar que tomamos de: application el Mapa que ha puesto el Filtro

Cuando el filtro ve que se han satisfecho sus condiciones y queremos que la solicitud continúe su camino normal (hacia otros filtros o hacia el recurso solicitado) ejecutamos:

```
chain.doFilter((HttpServletRequest) request, (HttpServletResponse) response);
```

● **Práctica 22:** Crear un proyecto con varias página jsp y/o html. Crear un filtro para todas las rutas: /* y hacer uso del código anterior para guardar la información de visitas finalmente visualizar en una página jsp los datos

Anexo: Cookies

La primera cookie se creó en 1994 cuando un empleado de Netscape Communications decidió crear una aplicación de e-commerce con un carrito de compras que se mantuviese siempre lleno con los artículos del usuario sin requerir muchos recursos del servidor. El desarrollador decidió que la mejor opción era usar **un archivo que se guardara en el equipo del receptor, en lugar de usar el servidor del sitio web**

Según wikipedia una cookie es:

Una cookie (galleta o galleta informática) **es una pequeña información enviada por un sitio web y almacenada en el navegador del usuario, de manera que el sitio web puede consultar la actividad previa del navegador.**

Vamos a profundizar en el concepto:

Tipos de cookie

Según quien sea la entidad que gestione el equipo o dominio desde donde se envían las cookies y trate los datos que se obtengan, podemos distinguir:

- **Cookies propias:** Son aquellas que se envían al equipo terminal del usuario desde un equipo o dominio gestionado por el propio editor y desde el que se presta el servicio solicitado por el usuario.
- **Cookies de tercero:** Son aquellas que se envían al equipo terminal del usuario desde un equipo o dominio que no es gestionado por el editor, sino por otra entidad que trata los datos obtenidos través de las cookies.

Según el plazo de tiempo que permanecen activadas en el equipo terminal podemos distinguir:

- **Cookies de sesión:** Son un tipo de cookies diseñadas para recabar y almacenar datos mientras el usuario accede a una página web, permanecen en su equipo durante la visita (por ejemplo, hasta que cierra el navegador y finaliza la visita).
- **Cookies persistentes:** Son un tipo de cookies en el que los datos siguen almacenados en el terminal y pueden ser accedidos y tratados durante un periodo definido por el responsable de la cookie, y que puede ir de unos minutos a varios años.

Según la finalidad para la que se traten los datos obtenidos a través de las cookies, podemos distinguir entre:

- **Cookies técnicas:** Son aquéllas que permiten al usuario la navegación a través de una página web, plataforma o aplicación y la utilización de las diferentes opciones o servicios que en ella existan como, por ejemplo, controlar el tráfico y la comunicación de datos, identificar la sesión, acceder a partes de acceso restringido, recordar los elementos que integran un pedido, realizar el proceso de compra de un pedido, realizar la solicitud de inscripción o participación en un evento, utilizar elementos de seguridad durante la navegación, almacenar contenidos para la difusión de videos o sonido o compartir contenidos a través de redes sociales.
- **Cookies de personalización:** Son aquellas que permiten al usuario acceder al servicio con algunas características de carácter general predefinidas en función de una serie de criterios en el terminal del usuario como por ejemplo serian el idioma, el tipo de navegador a través del cual accede al servicio, la configuración regional desde donde accede al servicio, etc.
- **Cookies de análisis:** Son aquéllas que permiten al responsable de las mismas, el seguimiento y análisis del comportamiento de los usuarios de los sitios web a los que están vinculadas. La información recogida mediante este tipo de cookies se utiliza en la medición de la actividad de los sitios web, aplicación o plataforma y para la elaboración de perfiles de navegación de los usuarios de dichos sitios, aplicaciones y plataformas, con el fin de introducir mejoras en función del análisis de los datos de uso que hacen los usuarios del servicio.

Ejemplo de cookie:

Request	Cookies	Session	Context	Client and Server	Headers
Incoming cookies					
Name		JSESSIONID			..
Value		86D57EBF5987804707655C184741830F			..
No outgoing cookies					

La anterior imagen es fácil obtenerla creando un formulario jsp. **Por defecto JSP genera una cookie de sesión**

Algunos **ejemplos** de que pueden hacer las cookies por ti son:

- Recordar tu usuario y contraseña en una web
- Mostrar publicidad online en función de tus intereses
- Obtener [estadísticas](#) si eres un webmaster
- Recordar preferencias en una web
- Compartir en redes sociales
- Llenar un carrito de la compra en una [tienda online](#)

Beneficios de las cookies

Por estos motivos que te he listado **las cookies son beneficiosas y nos hacen la vida más fácil**. Por ejemplo, a mi no me apetece tener que loguearme con mi usuario y contraseña cada vez que entro en una red social en mi casa, las cookies ya se acuerdan por mi.

El uso de las cookies habitualmente:

Según un reporte de la Unión Europea sobre protección de datos que analizó cerca de 500 páginas web, el 70% de las cookies son de terceros y rastrean la actividad que se genera para ofrecer publicidad personalizada.

Otras sirven para personalizar el servicio que ofrece el sitio web, en función del navegador en uso.

Y otras son "técnicas" y sirven para controlar el tráfico, identificar el inicio de sesión del usuario, almacenar contenidos o permitir el uso de elementos de seguridad.

PELIGROS DE LAS COOKIES:

Acceder a leer las cookies que haya dejado un usuario en el ordenador mediante el navegador y así recabar información del usuarios

No se puede hablar realmente de peligro que una empresa rastree nuestro comportamiento en su servicio web desde que nosotros la información que le estamos dando lo hacemos libremente. Ahora bien, si que debemos ser conscientes que están haciendo un perfil nuestro con nuestros gustos y demás características de usuario. Por ejemplo, por medio de anuncios insertados en las

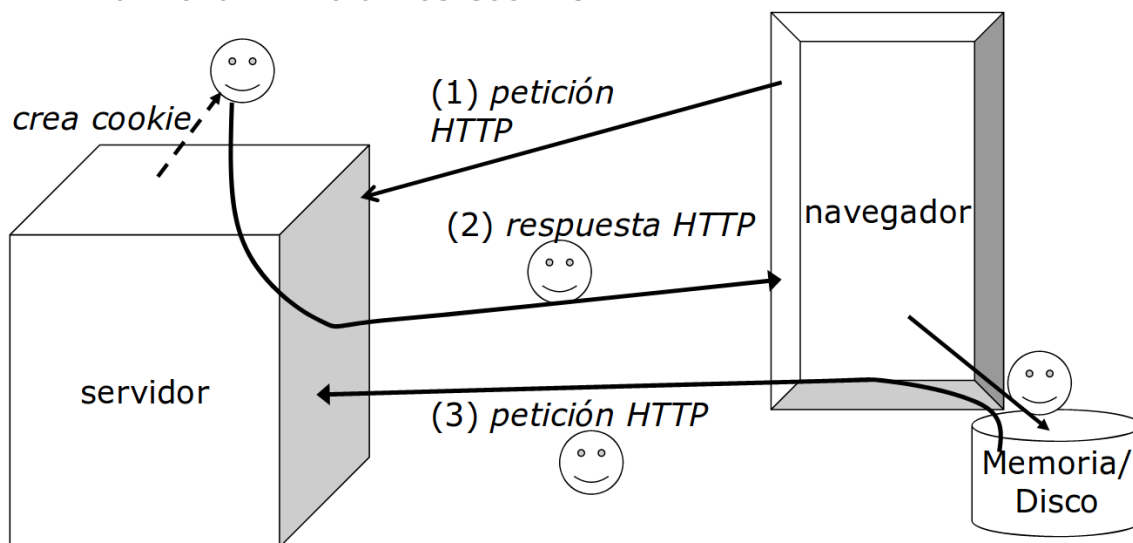
diversas páginas de los diversos webs que vamos visitando, si estos están gestionados por la misma empresa, esta podría realizar un seguimiento de todas esas visitas que realizamos en los webs de su red y trazar un perfil de nosotros como internautas y/o consumidores

la **Directiva 2009/136/CE** dice que: Los sitios web deberán pedir permiso a los usuarios para poder utilizar cookies y almacenarlas, salvo aquellas que sean absolutamente imprescindibles para el normal uso de un sitio web en el que el usuario se haya registrado y obliga a los webs a informar de manera clara y sencilla de lo que van a hacer con las cookies e indicarlo claramente en el momento de pedir autorización para su instalación

Más que peligros, cuenta como desventaja de las cookies que:

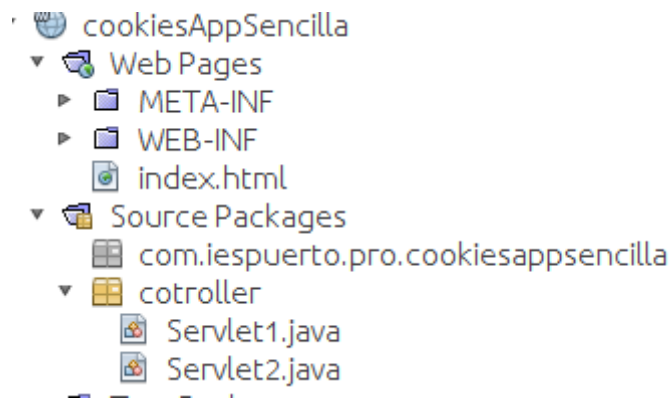
- No funcionan si el navegador del cliente ha desactivado las cookies
- Únicamente se puede enviar información en formato de texto

■ Funcionamiento de los Cookies



Creación y obtención de Cookies

Vamos a crear una app sencilla con la siguiente estructura:



Podemos observar que tenemos un index.html y dos servlet: Servlet1, Servlet2

el contenido de 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>
    <form method="post" action="Servlet1">

      <label> nick: </label>
      <input type="text" name="nick"/>
      <input type="submit" value="enviar" />

    </form>

    <a href="Servlet2" >acceso a servlet2</a>
  </body>
</html>
```

Vemos que enviamos por post a Servlet1 un nick introducido por el usuario Y también vemos que podemos acceder por Get a Servlet2

El único código que vamos a poner en Sevlet1 está en el método doPost();

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

```

        throws ServletException, IOException {

    String nick = request.getParameter("nick");
    if(nick != null && !nick.isEmpty()){
        Cookie c = new Cookie("nick", "Mr/Ms "+nick);
        c.setMaxAge(60); // está en 60 segundos
        response.addCookie(c);
    }

    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>Servlet1</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Se ha creado la cookie!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}

```

Bien, el código interesante es el de la creación de la cookie:

```

Cookie c = new Cookie("nick", "Mr/Ms "+nick);
c.setMaxAge(60); // está en 60 segundos
response.addCookie(c);

```

Vemos que **las cookies se crean como UN PAR CLAVE-VALOR**. Así, en este caso, habrá una key (clave) llamada: “nick” y un value (valor) que contiene: “Mr/Ms “ y el texto introducido por el usuario.

También vemos que podemos establecer cuanto tiempo de vida tiene una Cookie. Hay que tener en cuenta que **las cookies se crean en el navegador del cliente**, y con esa cantidad de tiempo le estamos diciendo al navegador que las mantenga durante 60 segundos.

Finalmente agregamos a nuestra respuesta (response.addCookie) la cookie para que el navegador del cliente la guarde.

Ahora leeremos el código del doGet() del Servlet2 que es donde único se ha puesto código:

```

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

```

```

Cookie listaCookies[] = request.getCookies();

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>Servlet1</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<ul>");
    for( Cookie c: listaCookies){
        out.print("<li> ");
        out.print(c.getName()+" ": "+c.getValue());
        out.println("</li> ");
    }
    out.println("</ul>");

    out.println("</body>");
    out.println("</html>");
}
}

```

Bien, hemos destacado en amarillo las líneas importantes. La primera nos dice como recogemos la información que nos envía el navegador mediante cookies:

```
Cookie listaCookies[] = request.getCookies();
```

Vemos que nos devuelve un array de cookies

Esta lista de cookies se puede recorrer mediante:

```

for( Cookie c: listaCookies){
    out.print(c.getName()+" ": "+c.getValue());
    //getName() devuelve la clave, getValue() el valor
}

```

● **Práctica 23:** Realizar la app de creación de cookies descrita. Tomar captura de pantalla que muestre la ejecución observando que la información que se le envía al Servlet1 (el nick) aparece en la respuesta del Servlet2 Tomar captura de pantalla después de 60segundos y explicar lo que ha ocurrido

● **Práctica 24:** Crear una app que el usuario en la página index.html tenga enlace a otras 3 páginas jsp. Cada una de esas páginas acceden mediante un formulario a 3 servlet distintos. En las tres debe aparecer las preferencias del usuario que haya establecido. En el primera combinación jsp-servlet: Servlet1 podrá elegir entre 3 radiobutton para tres colores: azul, verde, gris. Al enviar el formulario debe cambiar el fondo de las páginas al color que haya establecido. En la segunda combinación jsp-servlet (En otro formulario) puede establecer su nombre, por ejemplo: Bender, y en todas las 3 páginas jsp dirá: hola Bender! En el tercer y último jsp-servlet (Servlet3) habrá un formulario para enviar “pensamiento” la frase que envíe debe aparecer en la página que devuelve Servlet3

Una alternativa es enviar como cookie únicamente un uid buscar ese uid en el modelo de nuestra app y retornar la información almacenada para ese uid:

```
String uid = new java.rmi.server.UID().toString();
```

```
cookie = new Cookie("uid",java.net.URLEncoder.encode(uid,"UTF-8"));
```

Cabeceras HTTP al establecer y enviar una cookie

Desde el servidor se podría enviar la siguiente cabecera para establecer dos cookies:

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=strawberry
```

y a partir de entonces el navegador siempre que se comunique con el servidor enviaría las siguientes cabeceras:

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Parámetros disponibles con cookies

una cookie puede tener varios parámetros. Veamos mediante ejemplo:

```
Set-Cookie: ID=Gaigao%20pnchyyr;
           Domain=prueba.iespuerto.com;
           Path=/privado
           Expires=Wed, 21-Feb-2040 21:20:00 GMT;
           Secure;
           HttpOnly;
```

clave: ID

valor: Gaigao%20pnchyyr

dominio: La cookie se usará únicamente en páginas que cuelguen de: prueba.iespuerto.com

ruta: La cookie se usará únicamente en páginas que estén dentro del directorio: /privado

expiración: la cookie desaparecerá el 21-02-2040

secure: Se ha escrito: Secure; eso significa que la cookie se enviará únicamente por conexiones https

httponly: está diciendo que la cookie es accesible únicamente desde http, y no desde otros métodos como, por ejemplo, javascript

Observar que la combinación de dominio y ruta nos dice que la cookie se usará en rutas del estilo: **prueba.iespuerto.com/privado/mipagina**

Restricciones en la clave, valor de una cookie

Si leemos en la web de mozilla nos dice:

– <clave>: es definido como un token y como tal solo puede contener caracteres alfanuméricos más !#\$%&'*+,-.^_|~. No puede contener ningún espacio, coma o punto y coma. Tampoco están permitidos los caracteres de control (\x00, \x1F más \x7F) y no se debería utilizar el signo = que es utilizado como separador.

– <valor> puede contener cualquier valor alfanumérico excepto espacios, comas, punto y coma, caracteres de control, barra invertida y comillas dobles. En caso de que sea necesario el uso de estos caracteres, el valor de la cookie deberá ser codificado (reemplazar esos caracteres por su código ASCII); en JavaScript lo podemos hacer con encodeURIComponent(); en PHP se podría hacer con urlencode(); al leer el valor habría que descodificarlo con decodeURIComponent() o urldecode(). En java sería con algo como:

```
String q = "random word £500 bank $";
```

```
String codificado = "q=" + URLEncoder.encode(q, "UTF-8");
```

¿ Cómo borrar una Cookie ?

Debemos tener en cuenta que la cookie está en el navegador del cliente y no está plenamente bajo el control de nuestro servlet. Así hay navegadores que para saber que nos referimos a una cookie en concreto requieren toda la información que define esa cookie, no únicamente el nombre de la cookie. La cookie queda establecida por la tupla: NOMBRE, RUTA(PATH), DOMINIO Para enviarle esa información crearemos una nueva cookie con todos los datos de la que queremos borrar estableciendo su tiempo de vida a 0

```
Cookie delCookie = new Cookie(myCookie.getName(), myCookie.getValue())
delCookie.setDomain(myCookie.getDomain());
delCookie.setPath(myCookie.getPath());
delCookie.setMaxAge(0); // aquí decimos que queremos que muera
resp.addCookie(delCookie);
```

Nota: Es importante añadir a lo anterior que si no ponemos nada en: `setDomain()` entonces el dominio queda en null y eso nos dará problemas para establecer ese dominio. En ese caso el código anterior quedaría así:

```
Cookie delCookie = new Cookie(myCookie.getName(), myCookie.getValue())
if( myCookie.getDomain() != null )
    delCookie.setDomain(myCookie.getDomain());
delCookie.setPath(myCookie.getPath());
delCookie.setMaxAge(0); // aquí decimos que queremos que muera
resp.addCookie(delCookie);
```

¿ Y qué si ponemos otro dominio ?

Que no habría problema, pero tener en cuenta que si establecemos un dominio en el cual no estamos realmente trabajando hará que las cookies no las guarde el navegador. Por ejemplo, si establecemos `setDomain("midominio.com")` y estamos haciendo las pruebas en localhost sin que sea un dominio real fallará. Ahí habría que solucionar con técnicas de DNS, tocando `/etc/host` o cosas similares

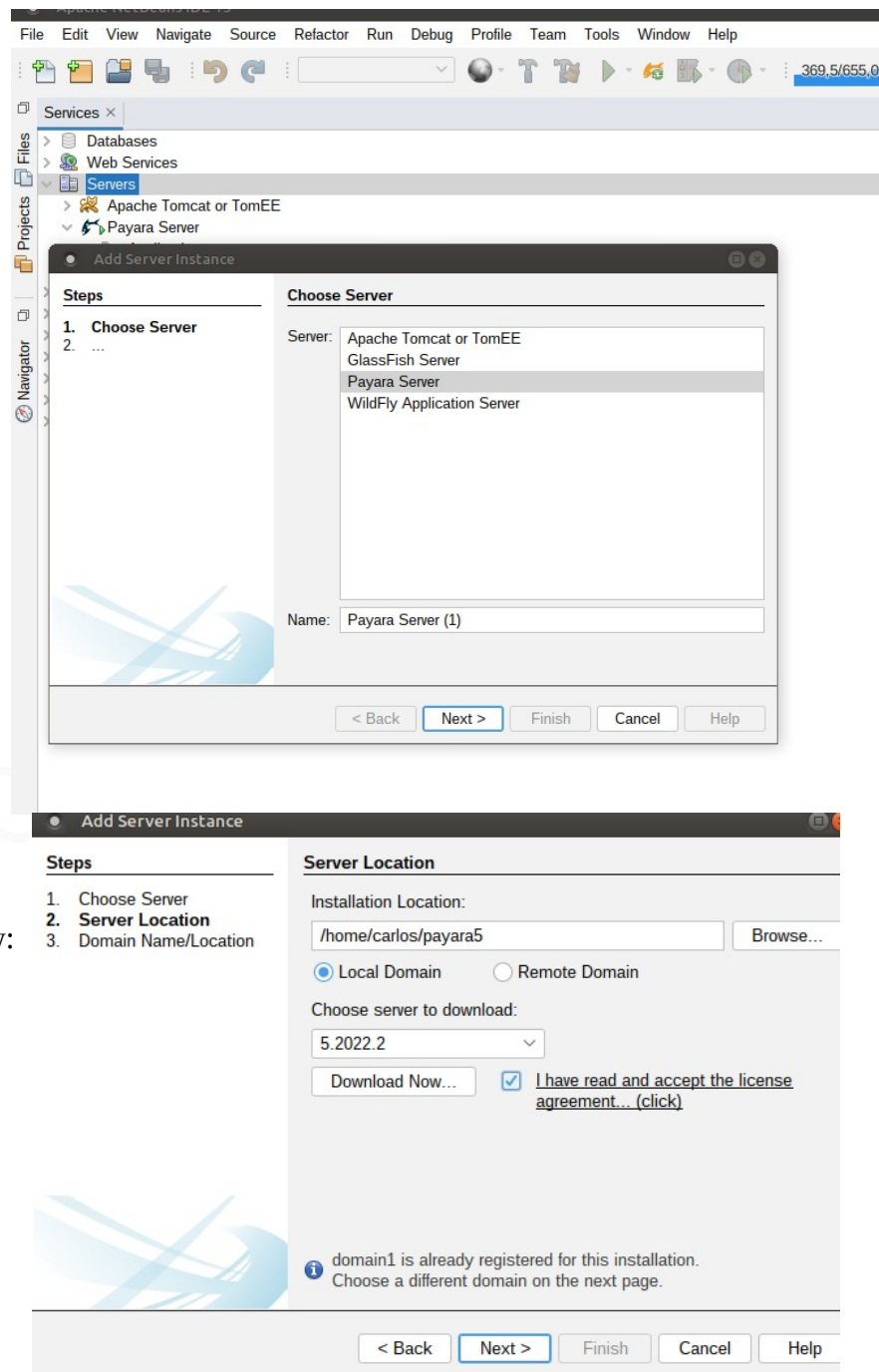
● **Práctica 25:** Crear una app basada en la práctica de cookies con el nick. Quitarle el tiempo de vida establecido a la cookie comprobar que no se muere pasados 60 segundos Crear un acceso a un segundo servlet que elimine la cookie Comprobar que después de ejecutar el servlet2 queda borrada la cookie al acceder al servlet1

Juan Carlos Pérez Rodríguez

Anexo: Proyecto maven netbeans web con payara server

Para crear un servidor payara desde netbeans 13 es fácil ya que tiene un wizard que nos descarga el server: Services → Servers → botón derecho → Add server

Seleccionamos payara:



Pulsamos en download now:

host: localhost, domain: domain1, das port: 4848, http port: 8080, target: vacío, user name: admin, Password: vacío

Add Server Instance

Steps

1. Choose Server
2. Server Location
3. **Domain Name/Location**

Domain Location

Domain:

Host: ☒ Loopback

DAS Port: HTTP Port: ☐ Default

Target:

User Name:

Password:

Si tenemos problema para arrancarlo porque hay problemas con el puerto 5900 (vnc) cambiamos la configuración en el fichero:

(home de payara)/glassfish/domains/domain1/config/domain.xml

Guscar la línea que hable de hazelcast-runtime-configuration y establecemos 59000 como puerto de inicio en lugar de 5900:

```
<hazelcast-runtime-configuration start-port="59000"></hazelcast-runtime-configuration>
```

Juan Carlos Pérez Rodríguez