

TEMA : Java Web Avanzado

Sumario

.....	1
Subir y bajar ficheros.....	2
Como subir ficheros.....	5
tipos de codificaciones posibles con POST.....	5
Crear el directorio donde guardar los ficheros.....	6
Recoger los ficheros del request y guardarlos.....	6
Bajar Ficheros.....	7
Como listar ficheros de un directorio.....	7
Como descargar cualquier tipo de fichero.....	8
Cookies.....	10
Tipos de cookie.....	10
Beneficios de las cookies.....	12
PELIGROS DE LAS COOKIES:.....	12
Creación y obtención de Cookies.....	13
Cabeceras HTTP al establecer y enviar una cookie.....	17
Parámetros disponibles con cookies.....	18
Restricciones en la clave,valor de una cookie.....	19
¿ Cómo borrar una Cookie ?.....	19
Enlazar un fichero CSS o JAVASCRIPT a un JSP.....	21
Sesiones.....	23
¿ Qué nos aporta el uso de sesiones ?.....	25
Ámbitos de una aplicación Web.....	29
Actividad Carrito de compras.....	34
DAO.....	36
welcome-file-list en web.xml.....	40
Uso de JSLT.....	43
Como ordenar una Collection de una Entity JPA.....	46
Filtros.....	53
Paginado de resultados.....	59

Subir y bajar ficheros

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

El index.html en su versión más básica:

Almacen Ficheros

[Listar archivos subidos](#)

Subir ficheros

Examinar...

No se han seleccionado archivos.

Subir

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

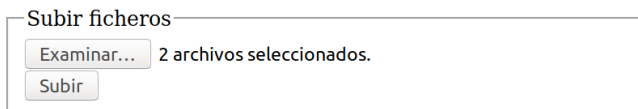
Lista ficheros:

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

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

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

Con dos ficheros seleccionados antes de pulsar “subir”:



Subir ficheros

Examinar... 2 archivos seleccionados.

Subir

La respuesta con el ok del servlet:

Fichero se ha subido!!

¿ qué cosas nuevas precisamos saber para realizar la aplicación ?

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

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

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

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

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

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

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

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

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

- * listarficheros

- * subirfichero

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

Como subir ficheros

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

tipos de codificaciones posibles con POST

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

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

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

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

como se codifica: x-www-form-urlencoded

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

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

text/plain: Las cargas útiles que utilizan el formato texto / plano están destinadas a ser legibles por el ser humano. No son confiablemente interpretables por computadora, ya que el formato es ambiguo (por ejemplo, no hay forma de distinguir una nueva línea literal en un valor de la nueva línea al final del valor)

Para consultarlo en contexto original ir a:

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

Crear el directorio donde guardar los ficheros

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

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

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

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

Recoger los ficheros del request y guardarlos

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

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

Así por ejemplo, la definición nos quedaría:

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

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

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

Bajar Ficheros

Lo siguiente ya corresponde al servlet: `ListarFicheros`

Como listar ficheros de un directorio

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

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

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

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

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

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

Como descargar cualquier tipo de fichero

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

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

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

y estableceremos las cabeceras de respuesta:

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

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

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

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

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

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

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

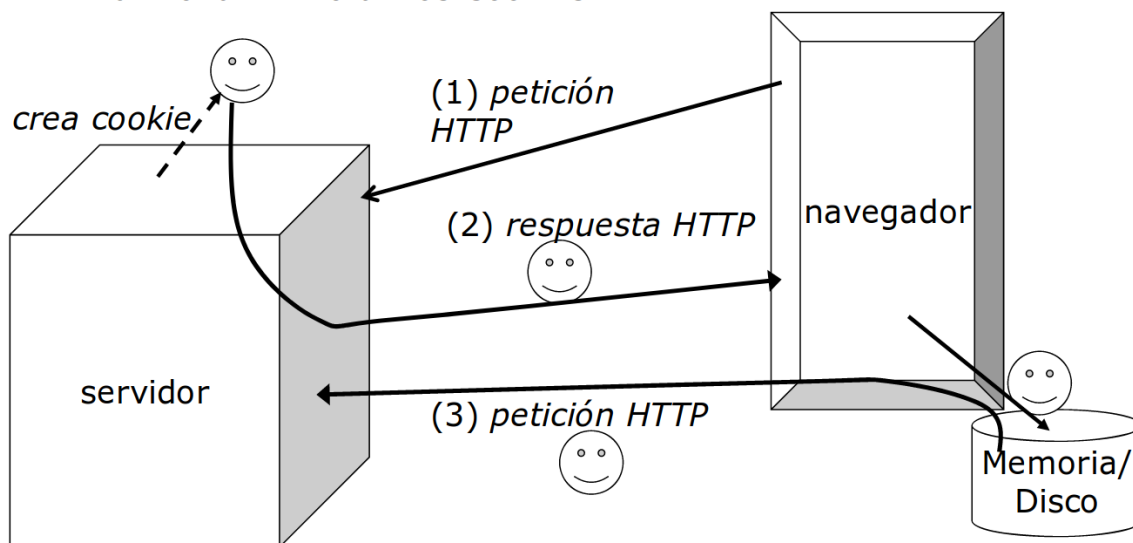
diversas páginas de los diversos webs que vamos visitando, si estos están gestionados por la misma empresa, esta podría realizar un seguimiento de todas esas visitas que realizamos en los webs de su red y trazar un perfil de nosotros como internautas y/o consumidores

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:

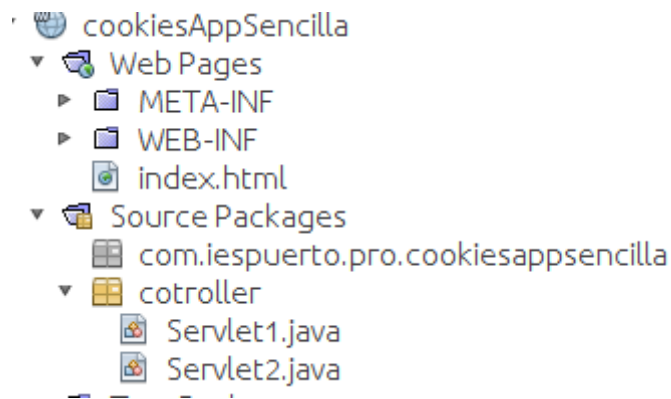
- No funcionan si el navegador del cliente ha desactivado las cookies
- Únicamente se puede enviar información en formato de texto

■ 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 1:** 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 2:** 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

● **Práctica 3:** Crear una app que el usuario envíe mediante un formulario un color en cada envío Cuando el usuario haya enviado el 5º color se le informa de los colores que ha introducido y el tiempo que ha tardado en hacerlo.

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"));
```

● **Práctica 4:** Hacer el ejercicio anterior pero guardando en el servidor la información salvo el uid que se guardará en el cliente como una cookie

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 5:** 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

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>`

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:

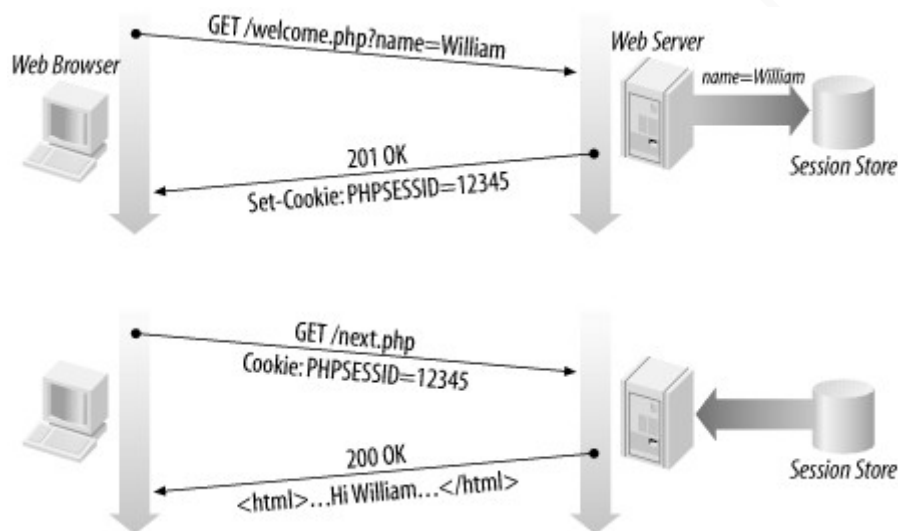
```
HttpSession session = request.getSession();  
Usuario usuario = (Usuario)session.getAttribute("usuario");
```

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; charset=");
    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 6:** 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 creo 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 7:** 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 8:** Modificar la práctica que dice: “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... “
En esta ocasión la información quedará en el servidor mediante una sesión en lugar de enviar las diferentes cookies

- **Práctica 9:** 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 10:** 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.

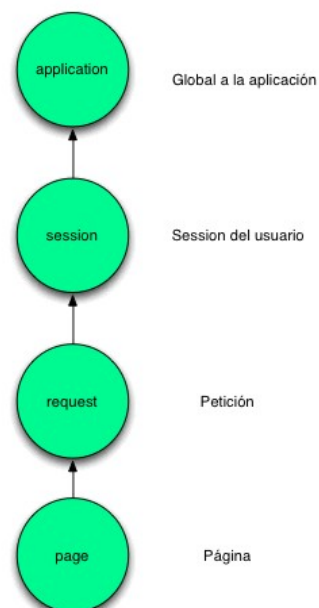
```
public class Ambitos extends HttpServlet {  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse  
response) throws ServletException, IOException {  
        //response.setContentType("text/html;charset=UTF-8");  
        ServletContext aplicacion = request.getServletContext();  
        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);
    }
}
```

● **Práctica11:** Crear un proyecto web con el servlet anterior siendo ese servlet lo primero que se inicia al arrancar la aplicación. Construir el jsp: `ambitos.jsp` para visualizar el contenido de las variables. Abrir dos navegadores distintos (de esa forma el servlet va a crear dos sesiones distintas) y ejecutar intercaladamente en cada navegador la aplicación. Tomar captura de pantalla y explicar que variables están compartiendo los dos navegadores y cuales no

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.

● **Práctica12:** Modificar la práctica anterior para que se puedan distinguir los ámbitos de `request` (solicitud) de los demás

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.maventiemposesion.NewServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>Servlet2</servlet-name>
    <servlet-class>com.iespuerto.pro.maventiemposesion.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 1minuto**

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

 **Práctica13:** Realizar la práctica descrita arriba

Actividad Carrito de compras

Vamos a ir desarrollando un carrito de compras y aprovecharemos su realización para ir introduciendo diferentes conceptos teóricos

Veamos la página que desarrollaremos inicialmente:



El usuario hace click sobre el elemento que quiera y se va incorporando en el carrito. Si hace click varias veces en un mismo artículo, por ejemplo click 3 veces en Portátil, habrá agregado tres portátiles al carrito

Observar que ese proceso se puede realizar sin necesidad de estar autenticado, en la imagen adjunta se han agregado tres productos y aparece una ventana de login que nos informa que el usuario no está autenticado

Primero debemos realizar la base de datos que vamos a utilizar. Entre otras se precisarán las siguientes tablas:


- producto

- carrito
- detalle_carrito
- usuario

Los carritos mínimamente dispondrán de un ID, un campo fecha y un identificador de usuario

detalle_carrito surge de la relación entre producto y carrito ya que es del tipo N:M

Usuario registrará la info de usuario y una password que usaremos para autenticarlo en la aplicación

 **Práctica 14:** Realizar la base de datos solicitada y tomar captura del modelo relacional obtenido (hay herramientas que lo extraen de las tablas creadas como mysql workbench por ejemplo)

Vamos a abordar el siguiente paso para tener preparado el acceso a nuestra base de datos cuando la aplicación lo precise

DAO

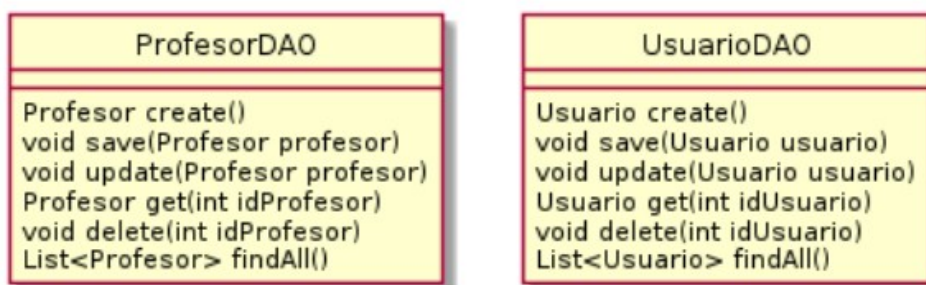
Según wikipedia:

En software de computadores, un objeto de acceso a datos (en inglés, data access object, abreviado DAO) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

Los Objetos de Acceso a Datos son un Patrón de los subordinados de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Los Objetos de Acceso a Datos pueden usarse en Java para aislar a una aplicación de la tecnología de persistencia Java subyacente (API de Persistencia Java), la cual podría ser JDBC, JDO, Enterprise JavaBeans, TopLink, EclipseLink, Hibernate, iBATIS, o cualquier otra tecnología de persistencia

El modelo DAO podría incluso tomar por cada clase que necesitamos crear una clase equivalente DAO. Por ejemplo, supongamos que nuestra aplicación usará dos clases que queremos persistir. Una se llama Profesor y la otra Usuario entonces el patrón DAO nos podría dar:



Si nos fijamos, entonces con la línea:

```
profesorDAO.update( profesor )
```

estaríamos en ese método update() tocando la base de datos para actualizar la información del objeto: profesor que tenemos en memoria

En el estilo de programación que hemos usado mediante JPA ya prácticamente tenemos eso saldado mediante las Entity que JPA nos crea de las tablas de la base de datos. Nosotros adicionalmente lo único que hemos estado creando son los métodos para “tocar” esas entities

Veamos un posible ejemplo de código para obtener la lista de productos:

```
public class OperarDB {

    private static EntityManagerFactory emf;

    private static EntityManagerFactory obtenerEntityManagerFactory(){
        if( emf == null || ( emf != null && !emf.isOpen() ) ){
            emf = Persistence.createEntityManagerFactory("unidadPersistencia");
        }
        return emf;
    }

    public static EntityManager crearEntityManager(){
        emf = obtenerEntityManagerFactory();
        return emf.createEntityManager();
    }

    public static ArrayList<Producto> obtenerProductos(){
        EntityManager em;
        em = crearEntityManager();
        List<Producto> datos;
        datos = em.createNamedQuery("Producto.findAll", Producto.class).getResultList();
        em.close();
        return new ArrayList<Producto>(datos);
    }
}
```

Observar que ahora si quisiéramos desde cualquier parte de la aplicación obtener la lista de productos pondríamos:

```
ArrayList<Producto> productos = OperarDB.obtenerProductos();
```

Y nos habríamos independizado de la gestión de la base de datos en el resto de la aplicación. Así que vamos a poner en: OperarDB todos aquellos métodos que necesitamos para la gestión de la base de datos

Por cada clase-tabla debemos crear en OperarDB los métodos para la creación, modificación, borrado y búsqueda del objeto

Así pues, por ejemplo para la clase Producto debemos tener:

- obtenerProductos() Nos devolverá el conjunto de productos
- buscarProductoPorId(int id) Nos devolverá el producto con el id
- buscarProductoPorNombre(String nombre) Nos devolverá el producto el nombre (observar que no se contempla que haya varios productos con el mismo nombre)
- actualizarProducto(Producto p) recibe un producto que ha sido modificado en memoria RAM y que ahora corresponde llevar esos cambios a la base de datos. Para ello se realizará una búsqueda del objeto en la base de datos por id y luego se ejecutará un merge() para la actualización:

```
Producto productoDB = em.createNamedQuery("Producto.findByIdproducto", Producto.class)
.setParameter("idproducto", p.getIdproducto())
productoDB.setNombre( p.getNombre() );
//entityManager.merge(productoDB);
transaction.commit();
```

Realmente lo anterior es un poco innecesario. Ya que el merge() lo que hace es enlazar nuestro objeto RAM con la base de datos y eso ya lo tenemos desde que hicimos la búsqueda en la base de datos. Tiene más sentido el merge() si lo hubiéramos realizado con el producto que hemos recibido como parámetro: entityManager.merge(p); Lo importante es la ejecución de los setNombre() etc **DENTRO DE LA TRANSACCIÓN** y luego con el commit() ya tiene lugar la actualización

- borrarProducto(Producto p) recibe un producto que esta en memoria RAM se busca por id en la base de datos y luego se borra (siempre dentro del entitymanager abierto, y la transacción abierta)

Como ya hemos explicado, debemos seguir las reglas de MVC para la creación de la aplicación, así pues **los JSP serán básicamente Vistas de la aplicación**, sin lógica propia, enviando o obteniendo la información pertinente a los controladores. **Los servlet harán de controladores y también los filtros** (ya los explicaremos más adelante), interactuarán con el modelo (los JavaBean de la base de datos principalmente)

Tendremos como mínimo un paquete **controller**, un paquete **model** dentro de source. Los JSP dentro de Web Pages también tendrán subdivisiones. Se creará una carpeta **admin** en webpages para los recursos jsp de admin por ejemplo

● **Práctica 15:** Crear OperarDB y las clases pertinentes para el acceso a la base de datos. En la clase OperarDB deben estar todas las operaciones pertinentes:

- buscarProductoPorNombre(String nombre)
- borrarProducto()
- ...

Se probará el funcionamiento de estos métodos desde un Servlet que se arranque al inicio de la aplicación Para ello es posible que haya que modificar welcome-file-list Leer lo siguiente

¿ cómo garantizamos que sea el Servlet lo primero que se acceda al entrar en la aplicación web ? Para ello haremos uso de los **welcome-file de web.xml**

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

Vamos a ir modelando la aplicación:

1. La imagen que se mostró antes es la página inicial de la aplicación. El usuario inicialmente no tiene por qué estar logueado pero sí que tendrá acceso a ver los productos disponibles a comprar y tiene a su disposición un carrito para ir almacenando aquellos artículos que quiera. Eso significa que debemos recorrer la base de datos de productos e ir mostrando esos productos. Como hemos planteado MVC esto no lo hará una página JSP directamente sino que tiene que haber un **Servlet que sea pues El PRIMER elemento que vamos a llamar** y es este servlet quién solicita los datos a la base de datos y se los pasa al JSP.

Y ¿ cómo es la mejor opción para enviar esos datos del Servlet al JSP ? Pues **mediante atributos de la sesión**. Así pues, habrá un servlet que será lo primero que se inicie y éste creará una sesión, consultará a la base de datos y pondrá como atributos de sesión los datos antes de hacer un forward al JSP correspondiente.

Ok ya tenemos detallado que vamos a realizar primero ese servlet

2. ¿ qué vamos a poner en ese servlet ?

¿ tiene sentido acceder por post a este servlet ? Quizás únicamente debamos implementar el `doGet()` Y en ese `doGet()` sabemos que debemos iniciar sesión: `request.getSession()`

y también sabemos que debemos buscar en la base de datos la lista de productos para luego enviarlos como atributos:

```
session.setAttribute("productos", productos)
```

y también sabemos que haremos un forward al JSP correspondiente:

```
request.getRequestDispatcher("productos.jsp").forward(request,response);
```

En cuanto al acceso a la base de datos podemos hacerlo mediante JDBC directo o mediante JPA En ambos casos debemos disponer de una o varias clases que nos encapsulen el acceso a la

base de datos y de esa forma si tuviéramos que cambiar de Gestor de Base de Datos por ejemplo, nuestros Servlets controladores no tendrían que modificar su código. Por ejemplo, el siguiente trozo podría ser del servlet GestorCarrito que nombramos antes:

```
ArrayList<Producto> productos = OperarDB.obtenerProductos();  
session.setAttribute("productos", productos);
```

Si nos fijamos en: OperarDB.obtenerProductos() Es una línea de código que no se tiene que modificar en el Servlet si tuviéramos que hacer cambios en la base de datos que usemos. Los cambios los registraríamos en todo caso en la clase: OperarDB

● **Práctica 16:** Crear el servlet que resuelva lo que hemos explicado así como las clases pertinentes para el acceso a la base de datos. El servlet deberá enviar como atributo de sesión la lista de productos y Crear el JSP que se ejecutará después del forward del servlet y mostrar la lista de productos

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 17:** Hacer uso de JSTL para que en el práctica anterior el for que recorra la lista de artículos se haga con un `forEach jstl`

Continuando con nuestra aplicación Carrito de Compras:

3. Lo siguiente que vamos a implementar es agregar objetos al carrito. Es importante remarcar que de momento no vamos a guardar el carrito en la base de datos. Queremos emular la situación de posibles clientes que acceden a la página y que prueba a ir comprando cosas pero sin llegar a terminar el proceso. ¿ Realmente queremos agregar carga a la base de datos para una situación así ? Podemos conservar las modificaciones del carrito en memoria y si el posible cliente decide “grabar” el carrito entonces se podría proceder a ese paso de guardar en la base de datos

Para lo anterior tendremos que guardar esos objetos en algún lugar que sea diferente para cada cliente que acceda a nuestra aplicación. No debe mostrarse el carrito de otro

¿ cómo solventarlo ? Sabemos que tenemos una estrategia sencilla que es el uso de sesiones

Así nuestro servlet de inicio deberá crear una sesión:

HttpSession session = request.getSession();

y guardar el carrito en esa sesión. Como vamos a guardar la fecha veamos como podemos poner la fecha en el carrito:

```
Carrito carrito= new Carrito();  
Calendar calendar = Calendar.getInstance();  
java.sql.Date fecha = new java.sql.Date(calendar.getTime().getTime());  
carrito.setFecha(fecha);
```

Aprovechamos para comentar como podemos ordenar (por ejemplo por fecha) mediante JPA

Nota: Como ordenar una Collection de una Entity JPA

Para poder ordenar una colección (por ejemplo en el caso anterior podría quererse la lista de carrito-pedidos que ha realizado un usuario por fecha)

```
@OneToMany(mappedBy = "fkUsuario")  
@OrderBy(value = "fecha ASC")  
private Collection<Carrito> carritoCollection;
```

Como vemos hemos puesto una anotación: **@OrderBy()** y le estamos diciendo que queremos que nos ordene la collection obtenida de carritos por fecha en orden ascendente

también deberemos crear un atributo de sesión para poner en ella nuestro carrito:

```
session.setAttribute(session.getId(), carrito);
```

Cada vez que queramos agregar un producto al carrito recibiremos un parámetro que nos avise de esa situación. Pongamos que ese parámetro se llama: agregar

Así pues bajo la condición de que hemos recibido ese parámetro:

```
String agregar = request.getParameter("agregar");
```

```
if( agregar != null){
```

¿ qué es lo que debemos hacer ? Observar que el carrito puede tener ya el producto que nos envían así que en ese caso lo que se modificará es la cantidad. Si el producto es diferente entonces debemos crear un nuevo objeto: DetalleCarrito para agregarlo a la lista del carrito:

Supongamos que hemos obtenido del parámetro agregar el id del producto:

```
int id = Integer.parseInt(agregar);  
DetalleCarrito detalleCarrito = new DetalleCarrito();  
Producto p = OperarDB.buscarProductoPorId(id);  
detalleCarrito.setIdproducto(p);  
detalleCarrito.setIdcarrito(carrito);  
detalleCarrito.setPrecio(p.getPrecio());  
detalleCarrito.setCantidad(1);  
carrito.getDetalleCarritoCollection().add(detalleCarrito);
```

Recordar que también se debe contemplar el otro caso que es que el producto ya esté en el carrito Así que se deberá hacer una búsqueda para determinar si el producto que el usuario quiere agregar ya está en el carrito en tal caso no es necesario crear un nuevo DetalleCarrito sino modificar la cantidad que hay en el anterior

4. Grabar el carrito-pedido

¿ Qué circunstancias tienen que darse ? :

- debe haber un usuario activo en la sesión que tengamos ya su información registrada en la base de datos. No es lógico un pedido anónimo
- En el momento de guardar el carrito que está en memoria se le pone el usuario que corresponde, se tiene que verificar que el pedido es posible (que haya stock) Para ello una posibilidad es dejar a la

base de datos con la responsabilidad de controlar la concurrencia de cambios de stock. Si no hubiera problemas de concurrencia basta con consultar previamente el stock y si no es suficiente abortar desde el programa. Ahora bien si hubiera problemas de concurrencia (varios servlet atacando la misma base de datos, por ejemplo) entonces lo mejor es dejar que las técnicas propias de la base de datos gestionen la concurrencia. Así podríamos poner un trigger parecido al siguiente:

```
CREATE DEFINER=`root`@`%` TRIGGER `appcarrocompras`.`producto_BEFORE_UPDATE`  
BEFORE UPDATE ON `producto`  
FOR EACH ROW  
BEGIN  
    if new.stock < 0 then  
        signal sqlstate '45000' set message_text="stock negativo";  
    end if;  
END
```

Como se ve, se lanza una excepción para impedir que tenga lugar la actualización de la tabla producto

Se debe tener en cuenta que al lanzarse una excepción se va a realizar un rollback automático y la transacción muere. Así que ponemos en el catch el siguiente if:

```
        et.commit();  
    }catch(Exception ex){  
        if(et != null && et.isActive()){  
            et.rollback();  
        }  
        bienGrabado = false;  
    }  
    em.close();  
    return bienGrabado;
```

Bien, entre las consideraciones anteriores aún nos queda responder: ¿ Y cómo solventamos que el usuario esté activo ?

La anterior afirmación tiene varias implicaciones en la aplicación:

4.1 Hay que crear un formulario para registrar usuarios y desde servlet guardar en la base de datos. Observar que debemos guardar password con función hash aplicada. NO SE GUARDAN PASSWORD EN TEXTO PLANO.

¿Cómo generamos ese hash ? Hay librerías disponibles

Pongamos en el pom la siguiente dependencia para la librería **bcrypt**:

```
<dependency>
  <groupId>org.mindrot</groupId>
  <artifactId>jbcrypt</artifactId>
  <version>0.4</version>
</dependency>
```

El funcionamiento de esa librería es muy sencillo:

```
String textoPlano = "1q2w3e4r";
String enHash = "";
// mediante hashpw() obtenemos una String con el hash de nuestro textoPlano
enHash = BCrypt.hashpw(textoPlano, BCrypt.gensalt());
// Bcrypt.checkpw() toma como primer parámetro un texto plano ( sin hash ) y lo
// compara con otro texto que se le ha aplicado hash si devuelve true es que el texto plano
// es el mismo que el que se usó para generar el hash
boolean resultado = BCrypt.checkpw(textoPlano, enHash);
```

Así pues haremos uso de `BCrypt.hashpw()` y guardaremos el hash obtenido en la base de datos. Luego al hacer un login se usará `BCrypt.checkpw()` para verificar la clave introducida por el cliente con la clave que tenemos guardada en la base de datos.

4.2 Hay que crear un login. El servlet que recibe la petición se encargará de poner el objeto Usuario en la sesión actual.

4.3 Hay que crear un logout. Puede ser una idea interesante “eliminar” toda la información de la sesión cuando el usuario hace logout. En ese caso haremos uso de: `session.invalidate()`

factura: 75 usuario: lui

Nombre	Cantidad	Precio/ud.	Subtotal
Portátil	3	1300.00€	3900.0€
Total: 3900.0			

- **Práctica 18:** Crear el trigger descrito Crear un Servlet que se acceda cuando el usuario pulse en “Guardar” Se le debe mostrar al cliente si el pedido realmente ha tenido lugar o no pudo hacerse por falta de stock Si tiene éxito se le mostrarán los datos de la factura (parecidos a la imagen de encima) Tener en cuenta que como aún no hemos gestionado los usuario pueden aparecer nulos. Una vez correctamente gestionado se debe impedir que un pedido no tenga un usuario vinculado

Registrar nuevo usuario:

Datos usuario:

Usuario
Password
Repetir password
Crear usuario

- **Práctica 19:** Crear el formulario de registro con 2 campos password, un Servlet para responder a ese formulario (que debe comprobar que el usuario no está repetido en DDBB, y que la password de formulario coincide, guardar en DB e informar al usuario de si a saliendo bien o no el registro)

Login

Usuario
Password
Entrar

- **Práctica 20:** Crear formulario de login y servlet que controle ese formulario. La función del servlet es validar usuario y password. Si todo sale bien, poner en la sesión el usuario

recuperado de la base de datos Si sale mal se le informa al usuario de si el usuario no existe o si la clave no es apropiada en la misma página de login

● **Práctica 21:** Crear un servlet para ejecutar el logout. Se accederá mediante GET y ejecutará `session.invalidate()`

5. Acceso a espacios restringidos

¿ cómo impedimos que un cliente que no ha hecho login acceda a páginas que no debe ? Hay una forma sencilla si estamos usando sesiones. Al poner como atributo de sesión al Usuario basta con chequear que ese atributo está o no establecido Veamos en cualquier formulario JSP:

```
<%  
//acceso únicamente si el usuario se ha establecido para esta sesión  
String usuario = null;  
if(session.getAttribute("usuario") == null){  
    response.sendRedirect("login.html");  
}else  
  
    usuario = (String) session.getAttribute("usuario");
```

Observar que el usuario sin login nunca llega a ver nada del JSP porque es redirigido a la página de login. En el caso que ya se haya logueado obtenemos la información del usuario de la sesión por si hay que personalizar la vista de los datos según la información de ese usuario

¿ E impedir que un usuario sin privilegios acceda a una página que debe tenerlos ?

Si por ejemplo, el usuario: admin es el único que tiene permiso para la página JSP que se está visualizando se agregaría al código anterior lo siguiente:

```
<%  
if( !usuario.getNombre().equals("admin") ){  
    out.println("no tiene privilegios para esta página");
```

```
}else{  
  
    //aquí el código de la página  
  
}
```

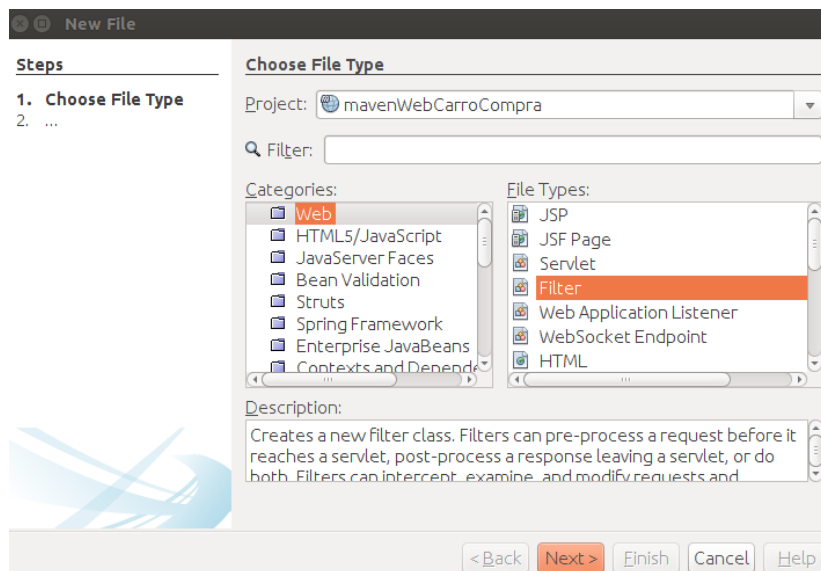
Ahora bien, la solución anterior implica chequear en cada página individualmente y Java tiene estrategias mejores: Los Filtros

Filtros

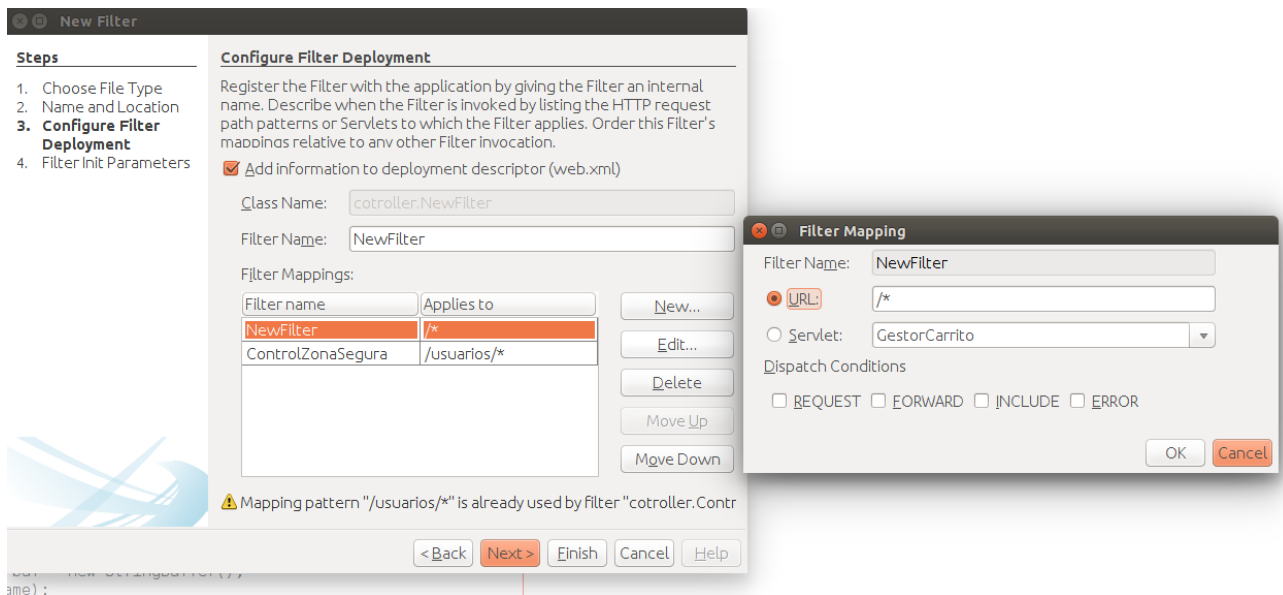
Un filtro de java se encarga de añadir una nueva funcionalidad a la aplicación colocandose entre el usuario y las páginas

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

```
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;
```

Nosotros donde vamos a trabajar 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.getContextPath();  
    res.sendRedirect(path+"/prueba.jsp");  
}
```

● **Práctica 22:** 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 23:** 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 24:** 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

Ahora continuemos con nuestra aplicación:

5.1 Para impedir que se acceda a páginas y servlet que son protegidas haremos uso de un filtro Por ejemplo podríamos tener un filtro para la ruta: /usuarios/*

El servlet que antes creamos para “Grabar” un carrito-pedido debe ser interceptado por ese filtro. No debe ocurrir que un cliente que no sea usuario registrado pueda crear un carrito en la base de datos.

Veamos las siguientes líneas de web.xml:

```
<servlet>
  <servlet-name>Grabar</servlet-name>
  <servlet-class>cotroller.Grabar</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>Grabar</servlet-name>
  <url-pattern>/usuarios/grabar</url-pattern>
</servlet-mapping>
```

El acceso al servlet se realiza mediante la url: /usuarios/grabar

y por tanto quedaría protegida por el filtro.

- **Práctica 25:** Crear en la aplicación del carrito el filtro descrito que redirigirá los accesos a la página: login.jsp si el cliente no se ha autenticado al tratar de grabar un carrito. Si el cliente se ha autenticado grabará el carrito

- **Práctica 26:** Usar un filtro para que sea case insensitive las peticiones del usuario, para la parte de subpath. Por ejemplo si nuestra aplicación está alojada en:
<http://localhost:8080/miProyectoWeb>

Esa parte se dejaría sin tocar pero los subpath serían case insensitive:

<http://localhost:8080/miProyectoWeb/hoMe.jsp>

<http://localhost:8080/miProyectoWeb/HoME.jsp>

<http://localhost:8080/miProyectoWeb/hoMe.JSP>

Para esta actividad se recomienda usar: request.getRequestURI() modificar lo que corresponda de la url recibida y hacer un: getRequestDispatcher() de la nueva url.

Nota: observar que para el dispatcher hay que pasarle únicamente la parte de la url después de: miProyectoWeb porque de otra forma lo va a agregar él de nuevo generando error (le pasamos rutas relativas)

Páginado de resultados

En ocasiones queremos paginar las salidas de nuestra web. En JPA tenemos una opción cómoda para obtener un subconjunto del conjunto de resultados. Por ejemplo si queremos 7 productos a partir del 21:

```
List<Producto> datos = em.createNamedQuery("Producto.findAll", Producto.class)
    .setFirstResult(21)
    .setMaxResults(7)
    .getResultList();
```

No es difícil observar que se acomodaría a la 4ª página si queremos 7 productos por página

- **Práctica 27:** Páginar los productos en nuestra aplicación de Carrito a razón de 4 productos por página