CLASE TEÓRICA 8 / 12

ÍNDICE DE CONTENIDO

8.1 Tecnologías del lado del servidor

Internet es uno de los medios fundamentales en los que puede residir un aplicación actual. Uno de los problemas de la creación de aplicaciones TCP/IP del lado del cliente es que el cliente debe poseer software adaptado a esa tecnología. Si no, la aplicación no carga correctamente.

Esto ocurre con cualquier aplicación del lado del cliente. Así una simple página web HTML requiere por parte del cliente un navegador compatible con los códigos HTML originales; si se usa JavaScript, el navegador del cliente debe poseer la capacidad de interpretar código JavaScript compatible con el original; si se usa Flash el navegador debe tener instalado el plugin, al igual que lo que ocurriría con ActiveX.

El cliente puede desactivar todas estas tecnologías o incluso no tenerlas instaladas. La única solución es hacer que el cliente las instale (algo que no suele funcionar muy bien debido a la desconfianza que tiene el usuario ante la solicitud de instalación de algún software).

Por si fuera poco, los usuarios siempre van por detrás de las innovaciones tecnológicas, por lo que sus versiones de software distan de ser las últimas.

Para evitar estos problemas se idearon técnicas de creación de aplicaciones para la web del lado del servidor. En las que la interpretación se realiza en el propio servidor y no en el cliente.

8.1.1 CGI

Common Gateway Interface, o interfaz de pasarela común (CGI) es la tecnología de servidor más veterana. Apareció debido a las limitaciones de HTML para crear verdaderas aplicaciones de red.

CGI define una serie de características que permiten comunicar a una página con una aplicación residente en un servidor. La aplicación puede estar escrita casi en cualquier lenguaje (aunque el más utilizado es el lenguaje Perl) lo único que tiene conseguir es que su salida y entrada ha de ser pensada para comunicarse con la web de forma que el usuario no necesite ningún software adicional (los datos de salida suelen prepararse en formato HTML).

El servidor en el que reside la aplicación CGI debe tener implementado un compilador compatible con el lenguaje utilizado para escribir la aplicación.

8.1.2 ASP y ASP.NET

ASP parte de simplificar la idea de la tecnología de servidor. Se trata de páginas HTML que poseen etiquetas especiales (marcadas con los símbolos <% y %>) que marcan instrucciones (en diversos lenguajes, sobre todo VBScript) que debe ejecutar el servidor.

El servidor interpreta esas instrucciones y obtiene una página HTML (que es la que llega al cliente) resultado del código ASP. Es una tecnología muy exitosa gracias a la cantidad de programadores Visual Basic.

El problema es que está pensada únicamente para servidores web IIS (Internet Information Server los

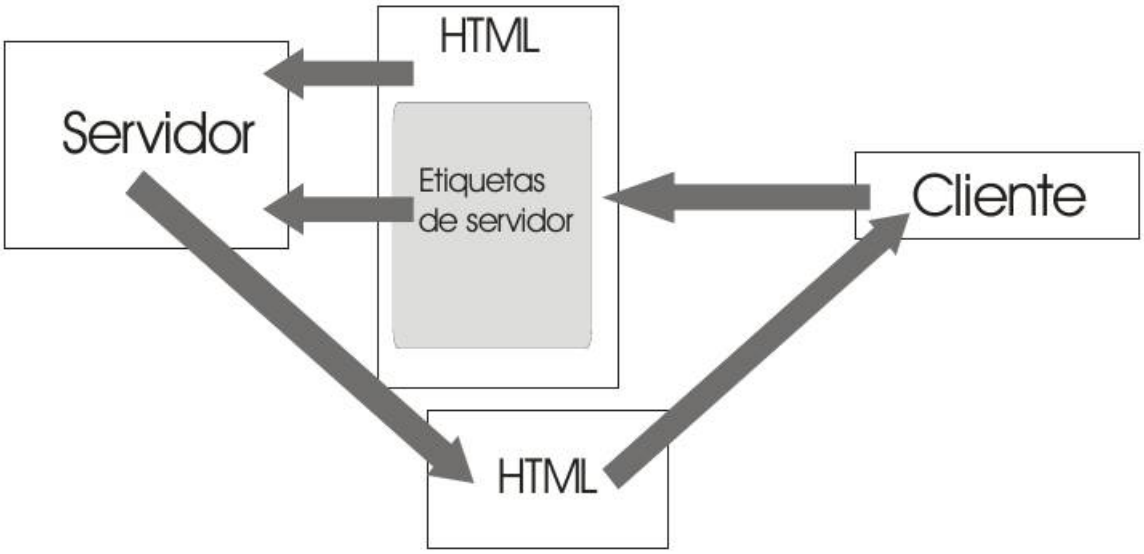
servidores web de Microsoft).

.NET es la nueva implementación de la tecnología de servidores de Microsoft que incorpora diversos lenguajes bajo una interfaz común para crear aplicaciones web en los servidores IIS. Se pueden utilizar varios tipos de lenguajes (especialmente C# y VBScript) combinados en páginas ASP.NET con directrices de servidor y posibilidad de conexión a bases de datos utilizando ADO (plataforma de conexión abierta de Microsoft para acceder a bases de datos, sucesora de ODBC).

8.1.3 PHP

Es una tecnología similar a ASP. Se trata de una página HTML que posee etiquetas especiales; en este caso son las etiquetas <?, que encierran comandos para el servidor escritos en un lenguaje script especial (es un lenguaje con muchas similitudes con Perl).

La diferencia con ASP es que es una plataforma de código abierto, compatible con Apache y que posee soporte para muchos de los gestores de base de datos más populares (Oracle, MySQL, etc.).



8.1.4 Servlets y JSP

Son las tecnologías planteadas por JAVA para crear aplicaciones cuyas tecnologías residan en el lado del servidor. JSP es similar a ASP y PHP. Los servlets forman parte de lo que se conocía como JavaEE (JAVA Enterprise Edition) y hoy en día se conoce como Jakarta EE. Una aplicación JSP o Servlet se conoce como aplicación web.

8.2 JavaEE / Jakarta EE

Se trata de una plataforma para construir aplicaciones completas, un conjunto de especificaciones estándares, escritas, revisadas y aprobadas por una comunidad de expertos, que en su conjunto definen

que servicios debería proveer y como se debería programar en una plataforma para aplicaciones empresariales. Se trata de una serie de tecnologías que permiten escribir aplicaciones en el lado del servidor para proporcionar servicios desde redes TCP/IP.

Mantienen el paradigma JAVA de la portabilidad incluso en el caso de cambiar el sistema operativo

del servidor. Sus APIs están en el paquete **`javax`** y desde que paso a ser mantenido por la Fundación Eclipse el paquete es nombrado **`jackarta`**. Las fundamentales son:

- Servlets
- JSP
- JAXP (API de procesamiento de documentos XML)
- EJB (Enterprise Java Beans)

Para ello hace falta ejecutar la aplicación JakartaEE en servidores web compatibles que posean un servidor de aplicaciones compatibles, o contenedor web, por ejemplo:

- Oracle WebLogic https://www.oracle.com/ar/java/weblogic/
- IBM WebSphere Application Server: https://www.ibm.com/ar-es/cloud/websphere-application-server
- Payara Jakarta Server: https://www.payara.fish/

De forma gratuita y de código abierto, open source (implementaciones parciales):

- Apache Tomcat: https://tomcat.apache.org/
- Eclipse Jetty: https://projects.eclipse.org/projects/rt.jetty

- WildFly: https://www.wildfly.org/
- GlassFish: https://javaee.github.io/glassfish/
- RedHat JBoss: https://developers.redhat.com/products/eap/overview?referrer=jbd

8.3 HTTP

El protocolo de transferencia de hipertexto es el encargado de transmitir páginas web por las redes TCP/IP. Toda la programación de aplicaciones se basa en el uso de este protocolo. Mediante HTTP el proceso de petición y respuesta de datos sería:

1. Un cliente establece conexión por un puerto (normalmente el 80) con un servidor web. En formato de texto envía una petición
2. El servidor analiza la petición y localiza el recurso solicitado
3. El servidor envía una copia del recurso al cliente
4. El servidor cierra la conexión

Los servidores web no recuerdan ningún dato de las peticiones anteriores, cada petición es independiente. Esto supone un serio problema al programar aplicaciones mediante este protocolo. Para evitar este problema se puede hacer que el servidor nos de un número de sesión que el cliente almacenará y enviará junto a las siguientes peticiones para que el servidor recuerde.

8.3.1 Peticiones HTTP

Las peticiones mediante http pueden utilizar los siguientes comandos:

- **`GET`**. Permite obtener un recurso simple mediante su URL en el servidor
- **`HEAD`**. Idéntico al anterior sólo que obtiene sólo las cabeceras del recurso.
- **`POST`**. Petición mediante la cual se hace que el servidor acepte datos desde el cliente.
- **`PUT`**. Permite modificar los datos de recursos en el servidor.
- **`DELETE`**. Permite eliminar recursos en el servidor.
- **`OPTIONS`**. Petición de información sobre los métodos de petición que soporta el servidor.
- **`TRACE`**. Pide la petición HTTP y las cabeceras enviadas por el cliente datos de la petición HTTP

8.3.2 Datos de la petición HTTP

Además del comando de petición, se debe indicar un segundo parámetro que es la línea de lo que se pide en el servidor. Esta línea es la ruta URL al documento pero sin las palabras http:// ni el nombre del servidor.

Es decir la URL http://www.profmatiasgarcia.com.ar/index.html, se pasa como /index.html.

Finalmente se indica la especificación http de la petición. Puede ser HTTP/1.0, HTTP/1.1 o HTTP/2.

8.3.3 Cabeceras de petición

Se ponen tras la línea de petición. Son líneas que incluyen una clave y su valor (separado por dos puntos). Mediante estas cabeceras el cliente indica sus capacidades e informaciones adicionales (navegador, idioma, tipo de contenido...).

8.4 Servlets

Un servlet es un programa que se ejecuta en el contenedor web de un servidor de aplicaciones. Los clientes pueden invocarlo utilizando el protocolo HTTP para luego recibir la respuesta del servlet cuando es cargado y ejecutado por el contenedor web.

Aparecieron en 1997 como respuesta a las aplicaciones CGI. Sus ventajas son:

- Mejora del rendimiento. Con las CGI lo que ocurría era que había que lanzar la aplicación con cada nueva petición de servicio. Las Servlets usan la misma aplicación y para cada petición lanzan un nuevo hilo (al estilo de los Sockets), lo que reduce el uso de memoria del servidor y el tiempo de respuesta.
- Simplicidad. Quizá la clave de su éxito. El cliente sólo necesita un navegador HTTP. El resto lo hace el servidor.
- Control de sesiones. Se pueden almacenar datos sobre las sesiones del usuario (una de las taras más importantes de http).
- Acceso a la tecnología JAVA. Lógicamente esta tecnología abre sus puertas a todas las posibilidades de JAVA (JDBC, Threads, etc.) y ser independiente de la plataforma.

8.4.1 Creación de Servlets

En principio, los servlets podrían comunicarse a través de cualquier protocolo cliente-servidor, pero se utilizan con mayor frecuencia con HTTP. Los Servlets se deben compilar en la carpeta classes de la aplicación web. Se trata de una clase normal pero que deriva de **`jakarta.servlet.HttpServlet`**. Hay una clase llamada **`jakarta.servlet.GenericServlet`** que crea Servlets genéricos, y una clase llamada **`jakarta.servlet.http.HttpServlet`** que es la encarga de crear servlets accesibles con el protocolo HTTP.

Para implementar y ejecutar un servlet, se debe utilizar un servidor de aplicaciones web o contenedor web. Un contenedor web (también conocido como contenedor de servlets) es esencialmente el componente de un servidor web que interactúa con los servlets. El contenedor web es responsable de administrar el ciclo de vida de los servlets, mapear una URL a un servlet en particular y garantizar que el solicitante de la URL tenga los derechos de acceso correctos.

8.4.2 Ciclo de vida

Un servlet genérico posee el siguiente ciclo de vida:

1. Cuando se cursó la primera petición al servlet, el servidor carga el Servlet.
2. Se ejecuta el método *`init()`* del servlet pasando un objeto que implementa la interfaz **`jakarta.servlet.ServletConfig`**. Este objeto de configuración se ejecuta una sola vez y permite que el servlet acceda a los parámetros de inicialización de nombre-valor desde la aplicación web.

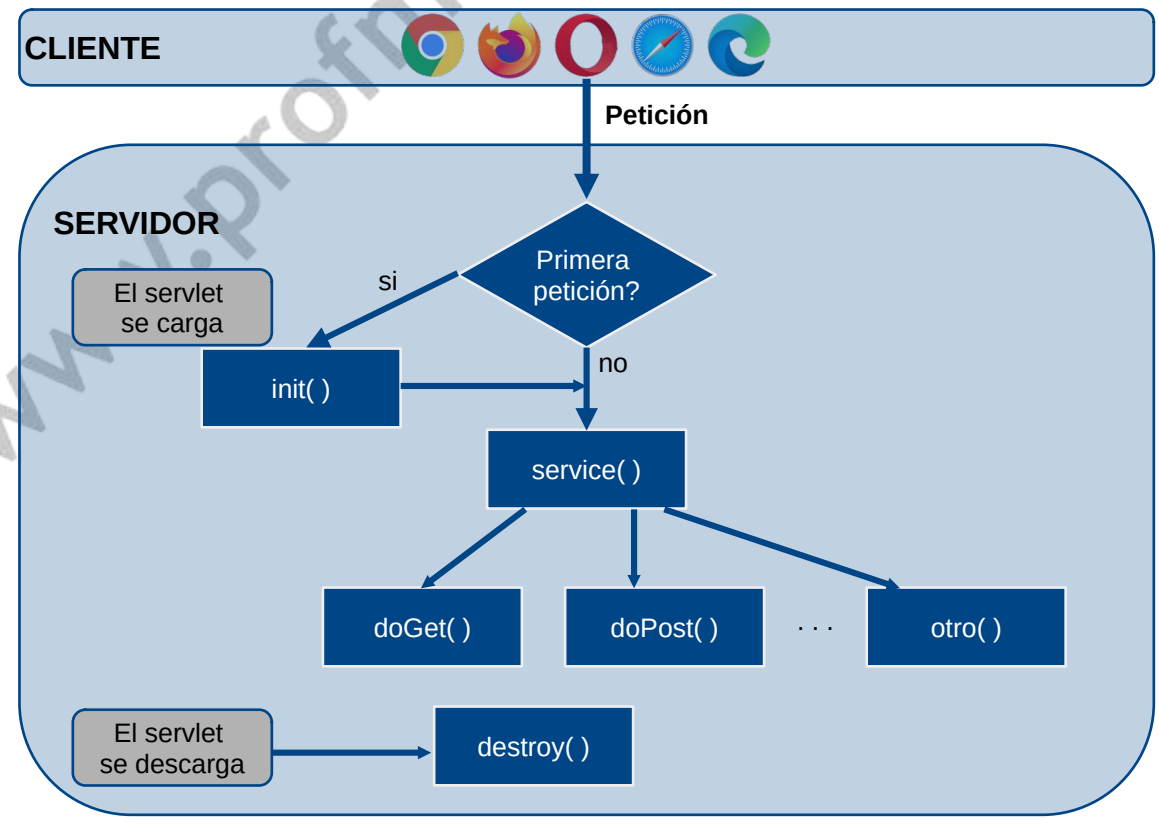
```
public void init(ServletConfig config) throws ServletException {
```

3. Después de la inicialización, la instancia de servlet puede atender las solicitudes de los clientes. Cada solicitud se atiende en su propio hilo independiente o thread. El contenedor web llama al método *`service()`* del servlet para cada solicitud. El *`service()`* determina el tipo de solicitud que se realiza y la envía a un método apropiado para manejar la solicitud. El desarrollador del servlet debe proporcionar una implementación para estos métodos. Si se realiza una solicitud para un método que no está implementado por el servlet, se llama al método de la clase principal, lo que generalmente resulta en un error que se devuelve al solicitante.

```
public abstract void service(ServletRequest request, ServletResponse response)
throws ServletException, IOException {
```

`request` y `response` son los objetos que permiten la comunicación con el cliente.

4. El método *`destroy()`* es llamado si se va a cerrar el servlet. Su función es liberar recursos. Llamar a *`destroy()`* no corta el servlet, esto sólo lo puede realizar el servidor.



Pero para servidores web se utiliza la clase **HttpServlet** cuyo ciclo difiere un poco ya que no se utiliza el método *service()*, sino que lo sustituye de forma que puede manejar diferentes tipos de solicitudes HTTP: **DELETE**, **GET**, **OPTIONS**, **POST**, **PUT**, y **TRACE**. Por cada uno de estos tipos de solicitud, la clase **HttpServlet** proporciona su correspondiente método *doXXX()*, por ejemplo *doGet()* o *doPost()*. De hecho el proceso para el método service es :

1. El método *service(request, response)* heredado de **GenericServlet** transforma estos objetos en sus equivalentes HTTP.
2. Se llama al nuevo método *service()* que posee dos parámetros.
    1. Uno es de tipo **HttpServletRequest** que encapsula los datos enviados por el cliente al servidor. Para acceder a los datos este objeto proporciona métodos como ***getParameter()***, que retorna el valor del parámetro especificado, ***getParameterValues***, que retorna una matriz de tipo **String** con todos los valores que el parametro tiene, entre otros.
    2. El otro es un objeto de tipo **HttpServletResponse** que encapsula los dato enviados por el servidor (por el servlet) al cliente. Este objeto proporciona, entre otros, el método ***getWriter*** que devuelve un objeto **FileWriter**.
3. El método anterior llama a *doGet()*, *doPost()* (que recibirán los mismos parámetros) u otro método programado, dependiendo del tipo de llamada HTTP realizada por el cliente (si es GET se llama a *doGet*, si es POST se llama a *doPost*, etc.)

Lo normal al crear un Servlet es crear una clase derivada de **HttpServlet** y redefinir el método *doGet()* o *doPost()* (o ambos) dependiendo de cómo se desee recibir la información. Es más, se suele crear un método común que es llamado por *doGet()* y d*oPost()*, ya que lo normal es que las llamadas **GET** o **POST** produzcan el mismo resultado, por ejemplo el método `processRequest(HttpServletRequest request, HttpServletResponse response)`

8.4.3 SALIDA DE DATOS DESDE LOS SERVLETS

Tanto el método *doGet()* como el método *doPost()* reciben como parámetros un objeto **HttpServletResponse** y un objeto **HttpServletRequest**. El primero sirve para que el servlet pueda escribir datos. Esos datos tienen que ,necesariamente, utilizar el lenguaje HTML. Para escribir datos se utiliza el método ***getWriter()*** de los objetos **HttpServletResponse**, el cual devuelve un objeto **PrintWriter** con el que es muy sencillo escribir, ya que posee el método *println()*.

Ejemplo:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<title>Escribiendo html</title>");
...
```

8.4.4 *INTERACCIÓN CON FORMULARIOS*

La virtud de un servlet es el hecho de que sea dinámico, es decir, que interactúe con el usuario. Una forma muy cómoda de hacer eso es enviar parámetros al servlet. Los servlets (como los CGI y cualquier otra tecnología de servidor) pueden tomar parámetros de dos formas:

- Insertando los parámetros en una URL con apartado de consulta
- Mediante formularios en las páginas web

El primer método sólo vale para peticiones **`GET`**. Consiste en colocar tras la ruta al servlet el símbolo ***`?`*** seguido del nombre del primer parámetro, el signo '**`=`**' y el valor del parámetro (se entiende que siempre es texto). Si hay más parámetros, los parámetros se separan con el símbolo '***`&`***'. Ejemplo:

```
https://www.google.com.ar/search?q=prof+matias+garcia&ie=UTF-8&hl=es&meta=
```

En este caso se llama al programa *`search`* y se le pasan cuatro parámetros: el parámetro *`q`* con valor *`prof+matias+garcia`*, el parámetro *`ie`* con valor *`UTF-8`*, el parámetro *`hl`* con valor *`es`* y el parámetro *`meta`* que no tiene valor alguno.

En la segunda forma se pueden utilizar peticiones **`GET`** o **`POST`**; la diferencia es que con **`GET`** se genera una dirección URL que incluye los parámetros (igual que en el ejemplo anterior), con **`POST`** los parámetros se envían de forma oculta.

Los formularios son etiquetas especiales que se colocan en los documentos HTML a fin de que el usuario se pueda comunicar con una determinada aplicación. Ejemplo:

```
<!DOCTYPE html>
<html>
<head>
   <title>Pagina inicial</title>
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
   <h1>Primer Web App</h1>
   <h2>Ingrese sus datos:</h2>
   <form action="SaludoServlet" method="POST">
         Ingrese su Nombre: <input type="text" name="nombre" size="20"> <br>
         Ingrese su Apellido: <input type="text" name="apellido" size="20"> <br>
<br>
         <input type="submit" value="Enviar">
   </form>
</body>
</html>
```

El método (`method`) puede ser *`"GET"`* o *`"POST"`*, basta con indicarlo. En este código HTML, se coloca un botón de tipo `type`=*`"submit"`* (*`"Enviar"`*) y dos cuadros de texto en el que el usuario rellena sus datos. El atributo `name` indica el nombre que se le dará a los parámetros que se enviarán al servlet. El parámetro tendrá como valor, lo que el usuario introduzca.

El parámetro es recogido desde el Servlet por el método ***`getParameter()`*** del objeto **`HttpServletRequest`** de los métodos *`doGet()`* o *`doPost()`*:

```
import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class SaludoServlet extends HttpServlet {
...
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    String txtNombre = request.getParameter("nombre");
    String txtApellido = request.getParameter("apellido");
    PrintWriter out = response.getWriter();
    // out.println("Nombre = " + txtNombre + " Apellido = " + txtApellido);
    response.setContentType("text/html");
    out.println("<html>");
    out.println("<title>Primer servlet</title>");
    out.println("<body>");
    out.println("<h3>Hola, su Nombre es " + txtNombre + "y su Apellido es " +
txtApellido + "</h3>");
    out.println("</body>");
    out.println("</html>");
    //processRequest(request, response);
    }
...
}
```

8.4.5 Métodos de HttpServletRequest

El objeto **HttpServletRequest** tiene métodos interesantes que permiten obtener información de los datos enviados en la petición realizada por un cliente:

- Parámetros de usuario
- Cabeceras http (pares nombre-valor)
- Otras informaciones http
- Manejo de cookies
- Obtención de la ruta de acceso de la petición
- Identificación de sesión

Método	Uso
`String getAuthType()`	Devuelve el nombre del sistema de autentificación del servidor (si no usa ninguno, devuelve null)
`Cookie[] getCookies()`	Obtiene un array con todos los objetos Cookie que ha enviado el cliente
`String getContentLength ()`	Devuelve el tamaño en bytes del contenido solicitado
`String getContentType ()`	Devuelve la cadena Content Type de la petición. El resultado es el tipo MIME que indica la petición (por ejemplo text/html)
`String getContextPath()`	Devuelve la porción de la URL referida a la URL del servlet
`Enumeration getHeaderNames()`	Devuelve una enumeración de todas las cabeceras http presentes
`Enumeration getHeader(String nombre)`	Devuelve el contenido de la cabecera http cuyo nombre es el indicado

`String getMethod()`	Devuelve el método de llamada a la aplicación que utilizó el cliente (GET, PUT o POST por ejemplo).
`String getParameter(String parámetro)`	Obtiene el contenido del parámetro cuyo nombre se pasa entre comillas
`Enumeration getParameterNames()`	Obtiene una lista con los nombres de los parámetros
`String[] getParameterValues(String nombreParámetro)`	Devuelve un array de cadenas con los valores correspondientes al parámetro indicado o null si no tenía parámetro
`Map getParameterMap()`	Devuelve un objeto map con los valores y parámetros de la petición.
`String getPathInfo()`	Obtiene la porción de ruta de la URL que sigue al nombre del servlet
`String getQueryString()`	Obtiene la cadena de la URL que sigue al carácter “?"
`String getRequestURL()`	Obtiene la URL completa empleada para la petición de página (incluida la zona ?)
`String getRemoteUser()`	Obtiene el nombre de usuario del cliente, si hay posibilidad de autentificarle.
`HttpSession getSession(boolean crear)`	Devuelve el objeto actual HttpSession si no hay, devuelve null (se creará uno nuevo si crear vale `true`).
`boolean isRequestedSessionIdFromCookie()`	Indica si el identificador de sesión se obtuvo de una Cookie
`boolean isRequestedSessionIdValid()`	`true` si el indicador de sesión es válido
`boolean isUserInRole(String rol)`	Devuelve `true` si el usuario está asociado al rol indicado.

8.4.6 Métodos de HttpServletResponse

Permite enviar información al cliente. En los servlets HTTP, esa información se pasa en formato HTML. Para ello se usa el método `setContentType` con valor `"text/html"`.

Método	Uso
`void addCookie(Cookie cookie)`	Añade una cabecera Set-Cookie para la cookie indicada.
`void addHeader(String nombre, String valor)`	Establece el valor indicado para la cabecera http nombre.
`void addIntHeader(String nombre, int valor)`	Establece el valor indicado para la cabecera http nombre. El valor se coloca con valor int
`boolean containsHeader(String nombre)`	Indica si la salida ya posee la cabecera indicada
`String encodeRedirectURL(String url)`	Soporta el seguimiento de sesiones utilizando el identificador de sesión como parámetro de URL que se enviará mediante sendRedirect(). Si el cliente soporta cookies, esto no es necesario.
`String getContentType()`	Obtiene la cadena MIME con el tipo de contenido de la respuesta
`void sendError(int estado) throws IOException`	Establece el código de estado http en el valor indicado.
`void sendError(int estado, String msg) throws IOException`	Lo mismo, pero además fija el mensaje de estado especificado.
`void sendRedirect(String location) throws IOException`	Coloca el estado http en 302 (movido provisionalmente) y se lanza al navegador a la nueva localización.
`void setCharacterSet(String tipoMIME)`	Indica que codificación se envía en las respuestas. La codificación se debe indicar en formato estándar (el definido por la IANA), por ejemplo:

	text/html;charset=UTF-8 Codificación de Estados Unidos
`void setContentType(String tipoMIME)`	Establece el tipo de contenido que se enviará en la respuesta
`void setHeader(String nombre, String valor)`	Coloca el valor indicado a la propiedad http cuyo nombre se indica
`void setIntHeader(String nombre, int valor)`	Coloca el valor indicado a la propiedad http cuyo nombre se indica. El valor se coloca como int
`void setStatus(int código)`	Establece el código de respuesta, sólo se debe utilizar si la respuesta no es un error, sino, se debe utilizar sendError

Para los códigos de respuesta utilizados por varias funciones (`setStatus` y `sendError` por ejemplo), esta interfaz define una serie de constantes estática que empiezan por la palabra SC. El valor de las constantes se corresponde con el valor estándar de respuesta.

Por ejemplo la constante `HttpServletResponse.SC_NOT_FOUND` vale `404` (código de error de "página no encontrada").

8.4.7 Contexto del Servlet

Interfaz proporcionada por el servidor de aplicaciones para entregar servicios a una aplicación web. Se obtiene el objeto `ServletContext` mediante el método `getServletContext` del objeto `ServletConfig`. Este objeto se puede obtener en el método `init`, ya que lo recibe como parámetro. Mediante el contexto se accede a información interesante como:

- Atributos cuyos valores persisten entre invocaciones al servlet
- Peticiones a otros servlets
- Capacidades del servlet (posibilidad de acceder a bases de datos, lectura de ficheros, etc.)
- Mensajes de error y de otros tipos

Método	Uso
`Object getAttribute(String nombre)`	Obtiene el atributo del servidor de aplicaciones cuyo nombre sea el indicado. Se necesita conocer el servidor de aplicaciones para saber qué atributos posee.
`Enumeration getAttributeNames()`	Obtiene una enumeración con todos los atributos del contexto de servlet
`void setAttribute(String nombre, Object valor)`	Establece el atributo de servidor con nombre indicado dándole un determinado valor.
`ServletContext getContext(String ruta)`	Obtiene el contexto de servlet del servlet cuya ruta se indica. La ruta debe empezar con el símbolo "/" (es decir, debe de ser absoluta) y el servlet debe estar en el mismo servidor.
`int getMajorVersion()`	Devuelve el número mayor de la versión de Servlet que el servidor de aplicaciones es capaz de ejecutar.
`int getMinorVersion()`	Devuelve el número menor de la versión de Servlet que el servidor de aplicaciones es capaz de ejecutar.
`String getMimeType(String archivo)`	Obtiene el tipo MIME del archivo cuya ruta en el servidor se indica
`String getRealPath(String ruta)`	Obtiene la ruta absoluta de la ruta interna al servidor que se indica

`URL getURL(String recurso)`	Devuelve un objeto URL correspondiente a la ruta de recurso en el servidor indicada
`String getServerInfo()`	Obtiene el nombre del servidor de aplicaciones que se utiliza como motor de Servlet
`void removeAttribute(String nombre)`	Elimina el atributo indicado del contexto del servlet.

8.4.8 Sesiones HTTP

La navegación mediante el protocolo HTTP, dista mucho de parecerse a una comunicación cliente—servidor típica. Cuando el cliente pide un recurso, el servidor se lo da y punto, ya que HTTP es un protocolo sin estado; esto es, cada petición es tratada de forma independiente por el servidor. Los navegadores hacen nuevas peticiones para cada elemento de la página que haya que descargar. Si el usuario hace clic se hace una petición para el enlace. El tema es que el servidor no sabe si una serie de peticiones provienen del mismo o de diferentes clientes y si, además, están relacionadas entre sí.

En definitiva, el servidor se olvida del cliente en cuanto resuelve su petición. Pero esto provoca problemas cuando se desea ir almacenando información sobre el cliente (listas de productos elegidos, datos del usuario, etc.).

Para recordar datos de usuario hay varias técnicas:

- Cookies. Un archivo que se graba en el ordenador del cliente con información sobre el mismo.
- Añadir un ID de sesión como parámetro. Se añade este número bien como parámetro normal en la dirección (**`GET`**) o como parámetro oculto (**`POST`**). Lo malo es que el número de sesión hay que añadirle continuamente mientras el usuario navegue.

La interfaz **`HttpSession`** permite utilizar un número de sesión, que el cliente guarda y utiliza en posteriores peticiones. Se puede utilizar un manejador de sesiones utilizando el método *`getSession`* del objeto **`HttpServletRequest`** (`request`) utilizado. Se pueden utilizar los métodos *`getAttribute`* para obtener objetos asociados al usuario con esa sesión y *`setAttribute`* para almacenar valores.

Método	Uso
`Object getAttribute(String nombre)`	Obtiene un atributo para la sesión actual. Si el nombre del atributo no existe, devuelve null
`Enumeration getAttributeNames()`	Obtiene una enumeración con todos los atributos de la sesión
`void setAttribute(String nombre, Object valor)`	Establece un atributo para la sesión con el nombre indicado, al que se asociará un determinado valor.
`void removeAttribute(String nombre)`	Elimina el atributo indicado de la sesión
`int setMaxInactiveInterval(int segundos)`	Establece el número de segundos que la sesión podrá estar sin que el cliente efectúe ninguna operación.
`int getMaxInactiveInterval()`	Obtiene el valor actual del intervalo comentado antes
`long getLastAccessedTime()`	Obtiene la fecha y hora en la que el usuario realizó su última petición. Devuelve el número de milisegundos de esa fecha, el formato es el mismo que el de la clase Date
`void invalidate()`	Anula la sesión actual, eliminando todos los objetos relacionados
`boolean isNew()`	Con `true` indica que el usuario aún no ha establecido sesión.

Como un cliente HTTP no puede notificar la necesidad de finalizar una sesión, cada sesión tiene un "timeout" asociado de modo que los recursos puedan ser liberados "reclaimed". El período de "timeout" puede ser accesado a través de los métodos `getMaxInactiveInterval` o `setMaxInactiveInterval` del objeto.

8.5 JSP (JAVA SERVER PAGES)

JSP (JAVA Server Pages) ofrece no sólo la independencia de operar en diferentes plataformas y servidores de páginas Web, sino que además combina el poder de la tecnología JAVA en el servidor con la facilidad de visualizar el contenido de las páginas HTML.

Una página JSP es una página web normal (sólo que con extensión `.jsp`) a la que se la puede añadir código JAVA utilizando unas etiquetas especiales dentro del código de la página. Estas etiquetas son traducidas por el servidor de aplicaciones al igual que traduce el código de un servlet. Las etiquetas JSP comienzan por `<%` y terminan por `%>`. JSP permite apreciar mejor la distinción entre el contenido de la información y su presentación.

Una página JSP típica es una mezcla de texto en HTML con elementos JSP. Realmente en la práctica todo JSP se convierte en un servlet, por lo que se necesita el mismo sistema de archivos (carpeta `WEB-INF`, `web.xml`, `classes`, `lib`, etc.). Por lo que las páginas JSP se almacenan en la raíz de la aplicación y sus librerías comprimidas como `jar` en la carpeta `lib` (todas las librerías deben almacenarse ahí, las del kit de JAVA no porque el servidor de aplicaciones ya las incorporará).

```
<%-- Esta página da la fecha y hora cuando fue solicitada así como el número de
accesos --%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<%@ page import="java.util.*"%>
<%!Date loadDate = new Date();
   int loadCounter = 1;%>
<!DOCTYPE html>
<html>
<head>
   <title>Ejemplo JSP</title>
</head>
<body>
   <h1>Bienvenid@!</h1>
   <h2>Desde <%=loadDate%>, ésta página ha sido accesada <%=loadCounter%>
         <%
         if (loadCounter++ == 1)
               out.println(" vez.<br>");
         else
               out.println(" veces.<br>");
         %>
         <%
         out.println("Regrese pronto");
         %>
   </h2>
</body>
```

```
</html>
```

8.5.1 Ciclo de vida de una página JSP

Toda página JSP atraviesa una serie de etapas:

1. Una página JSP comienza con el código nativo HTML-JSP. Es el código escrito por el programador, mezcla de HTML clásico e instrucciones Java.
2. La página JSP se transforma en el Servlet equivalente. Esto ocurre tras la petición, request, de la página por parte del cliente. Se genera pues un archivo class dispuesto a ser ejecutado cuando haga falta, el servidor lo almacena para su posterior uso, por lo que en peticiones siguientes ya no se compilará: a no ser que se detecten cambios en el código original
3. Se carga la clase para cada petición entrante, se le pasa la petición al servlet (ya compilado y creado en memoria). Si la página se ha modificado desde la última compilación, el servidor se da cuenta, genera el nuevo servlet, lo compila y lo carga de nuevo.
4. Finalmente la ejecución del servlet da lugar al código HTML que es el que recibe el cliente.

8.5.2 Ámbito y visibilidad de objetos JSP

Los objetos JSP pueden crearse:

- implícitamente por directivas JSP
- explícitamente a través de acciones
- excepcionalmente usando fragmentos de código

El ámbito del objeto establece su duración desde su creación hasta su destrucción; su visibilidad indica los lugares de la página donde puede usarse el objeto.

Ámbito	Categoría	Descripción
4 (mayor)	Aplicación	Pertenecen a la misma aplicación
3	Sesión	Pertenecen a la misma sesión
2	Petición	Provienen de la petición que es Atendida
1 (menor)	Página	Pertenecen a la página en que fueron creados

En general hay que intentar utilizar el ámbito mas especifico posible al contexto en el que estemos trabajando. Esto es, si un dato es utilizado solo en una pagina, se utilizará el ámbito de página, si es usado a través de múltiples peticiones va en el ámbito de sesión y si es usado en distintas sesiones se utilizará el ámbito de aplicación.

```
PageContext.setAttribute(nombre, valor, ámbito);
```

8.5.3 COMPONENTES DE LAS PÁGINAS JSP

DIRECTIVAS

Instrucciones dirigidas al servidor web que contiene la página indicando qué tipo de código se ha de generar. Sirven para importar clases o incluir archivos HTML. Formato:

```
<%@ nombreDirectiva atributo1="valor" atributo2="valor" %>
```

Directivas page

Es la directiva más importante. Permite usar diversos atributos muy importantes. Además se puede utilizar varias etiquetas page en el mismo archivo. Sus atributos más importantes son:

- `import`. Lista de uno o más paquetes de clases. Los paquetes *`java.lang.*`*, *`java.servlet.*`*, *`java.servlet.jsp.*`* y *`java.servlet.http.*`* se incluyen automáticamente.

  Si se incluye más de un paquete, se deben de separar con comas. Además se pueden colocar varias directivas page con el atributo import (es la única directiva que se puede repetir). Ejemplo:

  ```
  <%@ page import="java.io.*, java.sql.*, java.util.*" %>
  ```

- `session`. Para usar una página JSP el cliente debe establecer una sesión HTTP. Si ésta es `false` no se pueden usar sesiones en una página JSP. Por defecto es `true`.

- `contentType`. Especifica el tipo MIME de la página (normalmente *`text/html`*) y, opcionalmente su codificación (*`"text/html; charset=iso-8859-1"`*) es lo que se suele indicar en el caso de Europa occidental. Ejemplo:

  ```
  <%@ page contentType="text/html;ISO-8859-1" %>
  ```

- `pageEncoding`. Indica la codificación de caracteres de la página (por ejemplo *`UTF-8`*). No es necesaria si se utiliza el formato completo del `contentType`.

- `errorPage`. Permite pasar el control a otra página cuando se genera en el código una excepción sin capturar. Normalmente cuando esto ocurre, es el servidor de aplicaciones el que saca la página de error. De este modo se personalizan los mensajes de error. Las páginas de error deben colocar la directiva `page` con atributo `isErrorPage` a *`true`*. La página de error puede obtener información sobre el error ocurrido mediante la variable implícita `exception`. Esta variable está muy relacionada con los objetos **`Exception`** del JAVA tradicional y por ello posee los métodos *`getMessage`* y *`printStacktrace`* para acceder a la información de la excepción y así manipularla.

- `isErrorPage`. En caso de ser *`true`* indica que esta página es la versión de error de otra.

- `autoflush`. En valor *`true`* indica que el buffer de escritura se vacía automáticamente (valor por defecto).

- `buffer`. Se utiliza en combinación con el anterior para indicar la forma de enviar datos del Servlet. Se puede indicar el texto no que indica que no se usa buffer, y por tanto los datos se envían tan pronto son generados (si el `autoflush` está a *`true`*). Indicando un tamaño (que

indicará bytes) hace que se almacenen en el buffer hasta que se llena (con `autoflush` a *`true`* tan pronto se llena el buffer, se envía al cliente; con valor *`false`* en el `autoflush` se genera un excepción si se desborda el buffer).

- `info`. Equivalente al resultado del método *`getServletInfo`* de la página. Permite indicar un texto para describir el archivo JSP.
- `isThreadSafe`. *`true`* si la página acepta varias peticiones simultáneas. En *`false`* el servlet generado está ante la situación provocada por la interfaz **`SingleThreadModle`**.
- `language`. Indica el lenguaje de programación utilizado para programar en la página JSP. De forma predeterminada y recomendable se utiliza el valor *`"java"`*. Hay servidores que admiten que el lenguaje sea JavaScript e incluso otros, pero su portabilidad seria limitada.
- `extends`. Permite indicar una clase padre a la página JSP. Esto sólo debería hacerse si se posee una clase padre con una funcionalidad muy probada y eficiente, ya que todos los servidores de aplicaciones proporcionan una clase padre por defecto los suficientemente poderosa para evitar esta sentencia.

En cualquier caso la clase padre debe implementar la interfaz **`jakarta.servlet.jsp.HttpJspPage`**, que define diversos métodos que, obligatoriamente, habría que definir en la clase padre.

Directivas `include`

La directiva `include`, al estilo de la directiva `#include` del lenguaje C, permite añadir a la página código incluido en otro archivo cuyo nombre se indica. Pueden haber varias directivas `include`.

```
<%@ include file="/cabecera.html" %>
<%@ include="estilos.css" %>
```

Directivas `taglib`

Permite utilizar etiquetas personales cuya definición se encuentre en un descriptor de etiquetas (archivo TLD) cuya ruta se indica. Hay un segundo parámetro llamado prefix que permite elegir que prefijo poseen esas etiquetas. Ejemplo:

```
<%@ taglib uri="/descrip/misEtiquetas.tld" prefix="jsa" %>
```

Una etiqueta personal en el código sería:

```
<jsa:bandera>texto</jsa:bandera>
```

COMENTARIOS

Hay dos tipos:

- Propios de JSP. Comienzan por `<%--` y terminan por `--%>`. Sólo son visibles en el código original JSP
- Propios de HTML. Comienzan por `<!--`y terminan por `-->` Son visibles en el código HTML generado por el servidor

EXPRESIONES

Comienzan por <%= y terminan por %> Las expresiones en JSP permiten que el resultado de la evaluación de una expresión JAVA se convierta a una cadena de caracteres que será incluida en la página generada. Es decir, lo que se coloca como expresión es directamente traducible como HTML. Las expresiones pueden incluirse en gran variedad de contextos y no deben terminarse por puntos y comas.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
   <meta charset="UTF-8">
   <title>Ejemplo JSP</title>
</head>
<!-- Comienza cuerpo de página -->
<body>
   <h1>
         Bienvenido
         <%="Prof Matias Garcia"%>
   </h1>
</body>
</html>
```

INSTRUCCIONES

También llamadas scriptlets, van entre <% y %> y son sentencias puras JAVA. El servidor las codifica como parte del método service del servlet resultante. Se puede escribir cualquier fragmento de código JAVA, extendido por elementos HTML y no limitado a una línea. Permite escribir JAVA puro, utilizar objetos implícitos de JSP y variables, métodos y clases declaradas.

```
   <% for (int i = 1; i < 5; i++) { %>
         <h<%=i%>>Prof Matias Garcia</h<%=i%>>
   <% } %>
```

Siempre que sea posible, debemos evitar usar scriptlets JSP mientras que las librerías de etiquetas proporcionen una funcionalidad similar. Esto hace que las páginas JSP sean más fáciles de leer y de mantener, ayuda a separar la lógica de negocios de la lógica de presentación, y hará que nuestras páginas evoluciones más fácilmente. El código de JAVA dentro de scriptlets JSP no pueden ser reutilizados por otros JSP, por lo tanto la lógica común termina siendo re-implementado en múltiples páginas.

DECLARACIONES

Van entre <%! y %> y sirven para declarar variables de instancia, métodos o clases internas a nivel de página y deben seguir la sintaxis de JAVA. No pueden utilizar objetos implícitos.

Las variables declaradas pueden inicializarse y deben estar terminadas por punto y coma. Estas son diferentes para cada instancia (al igual que los métodos). En definitiva, los métodos, variables o clases de las declaraciones son globales a todo el archivo JSP. Si se desea una variable o método común a todas las instancias de la página (es decir, común a todas las sesiones del archivo), entonces se pueden declarar con **static**. Ejemplo:

```
<%!public int factorial(int n) {
      int resultado = 1;
      for (int i = 1; i <= n; i++)
            resultado *= n;
      return resultado;
}%>
```

OBJETOS IMPLÍCITOS

Se necesitan algunos objetos predefinidos para poder realizar algunas operaciones complejas.

request

Representa el objeto **`HttpServletRequest`** de los servlets, necesario para obtener información.

- **`String`** `getParameter(`**`String`** `name)` Devuelve el valor de un parámetro.
- **`Enumeration`** `getParameterNames()` Devuelve una enumeración con los nombres de todos los parámetros de la petición.
- **`String`**`[] getParameterValues(`**`String`** `name)` Los parámetros pueden tener valor múltiple, con esta función recuperamos un array con todos los valores para un nombre dado.
- **`String`** `getRemoteAddr()` Devuelve la IP del host desde el que se realiza la petición
- **`String`** `getRemoteHost()` Devuelve el nombre del host desde el que se realiza la petición.

response

Representa el objeto **`HttpServletResponse`** de los servlets. Permite escribir datos en la página, asiste al servlet en su generación de la respuesta para el cliente, contiene funciones para manejo de cabeceras, códigos de estado, cookies y transferencia de control, etc. Se obtiene por `getResponse()`.

pageContext

Para obtener datos sobre el contexto de la página. Dependientes de la implementación, espacios de nombres y otras facilidades como permitir almacenar información localmente a la página.

Para guardar y recuperar valores:

```
Object pageContext.getAttribute("clave");
void pageContext.setAttribute("clave", Object objeto);
```

session

Objeto **`HttpSession`**. Nos permite acceder a la sesión asociada a la petición. A través de este objeto podemos, entre otras cosas, guardar objetos que serán accesibles desde cualquier JSP de la sesión o invalidarla.

Para guardar y recuperar información usaremos:

```
Object session.getAttribute("clave");
void session.setAttribute("clave", Object objeto);
```

Y para invalidar o cerrar la sesión: `void session.invalidate();`

application

Es un objeto de la clase **`ServletContext`**. Este objeto es común para toda la aplicación web y, entre otras cosas, nos permite almacenar información que será accesible desde todas las páginas de la aplicación web, independientemente de la sesión.

Para guardar y recuperar valores:

```
Object application.getAttribute("clave");
void application.setAttribute("clave", Object objeto);
```

out

Representa el flujo de salida hacia HTML. Equivalente a lo que devuelve el método ***`getWriter`*** de la clase **`HttpServletResponse`**. Permite acceder a la salida del navegador desde los scriplet.

config

Objeto **`ServletConfig`** de esta aplicación. Permite acceder a parámetros de inicialización del servlet y a su contexto.

page

Referencia a la página JSP (es un objeto **`HttpServlet`**), es un sinónimo de **`this`**, no tiene utilidad en el estado actual de la especificación.

exception

Para captura de errores sólo válido en páginas de errores.

8.5.4 *Concurrencia*

La directiva `page` con el atributo `isThreadSafe` ofrece una forma de evitar los problemas que ocasiona la concurrencia. El problema con este enfoque es que no es escalable, reduciendo considerablemente el desempeño de sistemas con muchos procesos.

Una alternativa es usar regiones críticas en scriptlets para sincronizar el acceso a objetos compartidos.

```
<%@page import="java.util.*"%>
<%
synchronized (application) {
   Properties list = application.getAttribute("users");
   list.add("ana", new User("ana"));
   list.setAttribute("users", list);
}
%>
```

8.5.5 *Ruta real para acceder a los archivos*

Cuando se trabaja con JSPs y servlets, se usan las rutas relativas para hacer referencia a los archivos, por ejemplo: *`"/WEB-INF/Promedios.txt"`*. Sin embargo para leer o guardar datos en un archivo, es

necesario tener la ruta completa, es decir la ruta real del archivo. El servlet Context (contexto del servlet) maneja la información a nivel de toda la aplicación Web. La clase **`ServletContext`** contiene métodos que sirven para que un servlet se comunique con su contenedor. Todos los servlets de una aplicación tienen el mismo **`ServletContext`**.

El **`ServletContext`** contiene un método que sirve para obtener la ruta real de un archivo que está dentro del proyecto de la aplicación. Entonces, para obtener la ruta real de *`"/WEB-INF/Promedios.txt"`* hacemos:

```
ServletContext sc = this.getServletContext();
String path = sc.getRealPath("/WEB-INF/Promedios.txt");
```

La ruta expresada con diagonales invertidas: "\" es un problema en los sistemas operativos que requieren la diagonal normal "/". Se puede utilizar la siguiente instrucción para solucionar el problema: `path = path.replace('\\','/');`

8.6 Colaboración entre Servlets y JSPs

Uno de los principales problemas en el manejo de Servlets y JSPs, es el hecho de que a veces sea necesario que un servlet (o una página JSP) llame a otro Servlet o JSP, pero haciendo que la respuesta de este último dependa de una condición del primero. Es decir un Servlet delega la respuesta a una página a otro Servlet.

Hay tres métodos para hacer ésta colaboración:

1. Encadenado. Se utilizaba al principio, antes de la aparición de JavaEE, y su manejo es complejo e insuficiente para la mayoría de los problemas.
2. Lanzamiento de solicitudes. Permite lanzar una petición a otro Servlet o JSP. El lanzamiento de solicitudes permite que un Servlet (o un JSP) llame a otro recurso utilizando los objetos `request` y `response` ya creados. Una interfaz llamada **`RequestDispatcher`** permite esta posibilidad
3. Petición desde JavaScript tipo **`GET`**. Ésta es la menos elegante, pero soluciona fácilmente pequeños problema. La instrucción JavaScript `location=URL` permite modificar la página web actual sustituyéndola por la página indicada por su URL. Si en esa URL se incluyen parámetros (por ejemplo (*`?nombre=pepe&edad=28&nacionalidad=francia`*), entonces resulta que se llamará al Servlet o JSP indicados pasando esos parámetros.

La instrucción desde un Servlet sería:

```
out.println("<script language='JavaScript'>" +
"location='/servlet/form?nombre=pepe&edad=28" +
"&nacionalidad=francia';</script>");
```

8.6.1 Interfaz RequestDispatcher

Permite referenciar a un recurso web para recibir peticiones desde un Servlet. Permite dos métodos:

- `public void forward(ServletRequest request, ServletResponse response) throws ServletException, IOException{`

Permite enviar una solicitud a otro Servlet o página JSP. Utiliza los objetos `request` y `response`. De modo que estos objetos poseerán los datos (por ejemplo los parámetros) tal cual llegaron al Servlet.

- `public void include(ServletRequest request, ServletResponse response) throws ServletException, IOException{`

Permite incluir en el Servlet actual, la respuesta producida por otro Servlet o JSP.

El objeto **`RequestDispatcher`** necesario para poder realizar peticiones a otros recursos, se obtiene utilizando el método *`getRequestDispatcher`* del objeto `request`. Este método requiere una cadena con la dirección del recurso llamado. Esa dirección parte desde el contexto raíz. Es decir si estamos en el Servlet `/apli/servicios/servicio1` y queremos obtener el servlet `/aplic/servicios/servicio2` la cadena a utilizar es *`"/servicios/servicio2"`*.

Finalmente para poder pasar valores de un Servlet (o JSP) a otro, se puede utilizar el método *`setAttribute`*. Este método utiliza dos parámetros: el primero es el nombre del atributo que se desea pasar (es una cadena), el segundo es un objeto que permite colocar valor al atributo.

El Servlet (o JSP) destinatario de la petición puede obtener el valor del atributo utilizando el método *`getAttribute`*, que tiene como único parámetro el nombre del atributo que se desea obtener.

Los nombres de atributos cumplen las mismas reglas que los paquetes (por ejemplo un nombre de atributo sería com.miEmpresa.nombre).

8.6.2 *Proceso de lanzamiento de peticiones*

1. Desde el Servlet o página JSP, utilizar el objeto `request` para invocar al método *`getRequestDispatcher`*. A este método se le pasa la ruta del Servlet/JSP que será invocado
2. Configurar el objeto `request` para su uso en el Servlet o JSP invocado. En especial se suelen configurar atributos mediante la función *`setAttribute`* de los objetos `request`.
3. Utilizar el método *`forward`* de la clase **`RequestDispatcher`**. Este método llama al Servlet, JSP o página HTML referida en el método *`getDispatcherRequest`* pasando como parámetros los objetos `request` y `response`, de los que podrá obtener parámetros, atributos y cualquier otra propiedad

Las dos instrucciones que se usan para enviar datos de un servlet a una JSP son

```
request.setAttribute("nombreAtributo", objetoAEnviar);
request.getRequestDispatcher("nombre.jsp").forward(request,response);
```

El objeto `request` tiene los siguientes métodos para enviar información:

***`setAttribute`*`(String nombre, Object o)` → Sirve para guardar un objeto en un atributo al que se le da un nombre.**

`getAttribute(String nombre)` → Sirve para recuperar el objeto guardado en el atributo nombre.

`getRequestDispatcher(String ruta)` → regresa un objeto **`RequestDispatcher`** para la ruta especificada.

`forward(request, response)` → re-envia los objetos request y response a otro recurso en el servidor, que normalmente es una JSP o un servlet.

8.6.3 *Redirección de mensajes*

Transferencia de control con el objeto response También existe una forma de transferir a otra URL, y es con el siguiente método del objeto response:

`sendRedirect(String ruta)` → se envía al cliente a la ruta especificada.

Este método no transfiere los objetos request y response, por lo tanto, sólo se utiliza para re-direccionar a una URL específica, usualmente fuera de la aplicación actual.

Una petición se puede redirigir hacia otra página JSP, servlet o página HTML estática pasándole el contexto de la página que hace la invocación. El procesamiento continúa hasta el punto en el que ocurre la redirección para entonces detenerse por completo en esa página.

El elemento `<jsp:forward page="anotherPage.jsp"/>` redirige el procesamiento a la página `anotherPage.jsp` conservando todo el procesamiento realizado en la página actual.

Un elemento `<jsp:forward>` también puede tener parámetros para proporcionar valores a la página invocada:

```
<jsp:forward page="anotherPage.jsp">
      <jsp:param name="aName1" value="aValue1" />
      <jsp:param name="aName2" value="aValue2" />
</jsp:forward>
```

Ejemplo

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html>
<head>
   <title>Ejemplo redirección JSP</title>
</head>
<body>
   <%
   double memLibre = Runtime.getRuntime().freeMemory();
   double memTotal = Runtime.getRuntime().totalMemory();
   double percent = memLibre / memTotal;
   if (percent < 0.5) { %>
   <jsp:forward page="/jsp/forward/one.jsp" />
   <% } else { %>
   <jsp:forward page="two.html" />
   <% } %>
</body>
```

```
</html>
```

```
<!-- one.jsp -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP one</title>
</head>
<body bgcolor="white">
   <font color="red"> Uso de memoria por la VM < 50%. </font>
</body>
</html>
```

```
<!-- two.jsp -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP two</title>
</head>
<body bgcolor="red">
   <font color="white"> Uso de memoria por la VM > 50%. </font>
</body>
</html>
```

8.6.4 Comunicación entre servlets y clases Java

Los servlets constituyen el controlador de la aplicación, éstos utilizan las clases JAVA para procesar la información. Estas clases conforman el modelo. Una vez procesada la información, los servlets la mandan desplegar a la vista.

El intercambio de información entre servlets y clases JAVA es trivial. Sólo es necesario importar en el servlet, la o las clases con las que trabaja. Para enviar información a una clase, se instancia un objeto en el servlet y se envían los parámetros en el método constructor. El resultado del procesamiento se obtiene con los métodos de la clase diseñados para proporcionar datos y resultados.

8.7 Manejo de excepciones

JSP posee un mecanismo apropiado para el manejo de errores que ocurren a tiempo de ejecución.

El atributo `errorPage` de la directiva page se usa para indicar que la página recibe las excepciones lanzadas por algún scriptlet.

Además, el atributo errorPage de la misma directiva indica la página JSP a la cual se redirigirá el objeto implícito **`exception`** (de tipo **`Throwable`**) que describe el problema.

Ejemplo de manejo de excepciones

• Para manejar errores a tiempo de ejecución, se puede usar una página que indique la naturaleza del

error.

• Dicha página debe incluir entre sus directivas la siguiente:

```
<%@page isErrorPage="false" errorPage="Error.jsp"%>
```

• la página redirige a su vez el objeto exception a la página *"Error.jsp"*.

• La página *"Error.jsp"* debe contener a su vez, la directiva:

```
<%@page isErrorPage="true" %>
```

Esta indicación permite acceder al objeto exception en la página *"Error.jsp"*.

8.8 *Establecer sesiones*

Se dice que el protocolo HTTP es un protocolo sin estado, ya que las llamadas a este protocolo son independientes unas de otras. Los protocolos con estado permiten que las llamadas sean dependientes unas de otras. Esto es fundamental para realizar multitud de acciones que sería imposibles si el estado es independiente.

Temas como venta en línea, transacciones comerciales o páginas que se adapten a los criterios del usuario, no se podrían realizar sin conseguir una dependencia entre las llamadas. Por ello necesitamos establecer una sesión. Una sesión es una serie de solicitudes que forman una tarea completa de trabajo en la que se distinga a un cliente de otro.

El estado se consigue haciendo que el servidor recuerde información relacionada con las sesiones anteriores. Como HTTP cierra la conexión tras resolver la solicitud, no parece posible realizar estas operaciones. Para resolver este dilema, en el API de los Servlets (y por tanto en los JSP) se han incluido herramientas que solucionan el problema. Estas son:

- Reescritura de URLs. Se utiliza un identificador que se añade a la dirección URL utilizada en la petición HTTP. Cuando el cliente utiliza este identificador, el servidor le recoge para poder transmitirle de nuevo. Por ejemplo una llamada *http://www.miserver.com/servlet1;jsessionid=278282* generaría ese número de sesión. jsessionid es el la forma JAVA de indicar el número de sesión en la URL.
- Campos ocultos de formulario. Muy semejante al anterior, salvo que en lugar de reescribir el código, el servidor utiliza campos de formulario ocultos para permitir capturar el identificador de sesión. Los servlets no utilizan este enfoque
- Cookies. Son datos que se intercambian en la lectura y escritura y que se almacenan en el ordenador del cliente. Es un texto que el servidor le envía al cliente junto con la petición y que éste almacena en su ordenador. En cada petición el cliente enviará al servidor el cookie.
- SSL, Secure Socket Layer. Utiliza una tecnología de cifrado sobre TCP/IP (especialmente bajo http). El protocolo HTTPS se define con esta tecnología. Ser generan claves cifradas entre cliente

y servidor llamadas claves de sesión.

8.8.1 *Reescritura del URL*

Reescribir un URL significa añadir al final del mismo alguna información extra que identifique la sesión para que el servidor asocie ese identificador con los datos que ha almacenado sobre la sesión.

```
http://www.miservidor.com/catalogo/index.html;idsesionj=1234
```

Esta forma permite el funcionamiento en navegadores que no soportan cookies o han sido desactivadas. Tiene las mismas problemáticas y, además, hay que poner especial cuidado en añadir esa información extra por cada enlace en la respuesta del servlet. También si el usuario deja la sesión y vuelve mediante un enlace en su lista de favoritos, la información de sesión puede perderse.

8.8.2 *Campos ocultos en formularios*

Un parámetro o campo oculto es un control de entrada de tipo *"hidden"*. En este caso no se muestra ningún campo de entrada de datos al usuario, pero el par *variable valor* especificado es enviado junto con el formulario.

```
<input type="hidden" name="variable" value="valor">
```

Se suelen utilizar para mantener datos durante una sesión. Inconveniente: que solo funciona cuando las paginas se generan dinámicamente, ya que cada sesión tiene un único identificador.

8.8.3 *Sesión con cookies*

Una cookie es una pequeña cantidad de información (no más de 4KB) que un servlet puede crear y almacenar en la maquina del cliente y, posteriormente, consultar a través de la API de cookies de los servlets.

Existen ciertos aspectos que deben ser controlados a la hora de su implementación:

- Extraer la cookie que almacena el identificador de sesión entre todas las cookies, ya que puede haber varias.
- Seleccionar un tiempo de expiración apropiado para la cookie.
- Asociar la información del servidor con el identificador de sesión (cierta información que puede ser peligrosamente manipulada, como los números de tarjetas de crédito, nunca debe almacenarse en cookies).

El API de servlets permite enviar cookies al navegador del usuario

Una cookie tiene un nombre y un valor asociado (cadena de caracteres)

```
Cookie micookie = new Cookie("nombre-cookie", "valor-asociado");
```

Cada navegador debería soportar alrededor de 20 cookies por cada sitio web al que está conectado, 300 en total y puede limitar el tamaño de cada cookie a 4 KB con un espacio de almacenamiento de

2MB.

Para enviar una o varias cookies al navegador se incluyen en el objeto **`HttpServletResponse`** pasándolo como argumento.

```
response.addCookie(micookie);
```

Al enviar cookies puede hacer que se eliminen otras cookies por la limitación de espacio de almacenamiento del navegador. La fecha de expiración de una cookie es solo una sugerencia; es el navegador el que decide cuando eliminarlas.

Cada vez que el navegador hace una petición, todas las cookies relativas a esa aplicación web llegan en la request en una matriz.

```
Cookie[] cookies = request.getCookies();
if (cookies == null)
   out.println("no hay cookies");
```

Las cookies que un cliente almacena para un servidor solo pueden ser devueltas a ese mismo servidor. Por lo tanto, los servlets que se ejecutan dentro de un servidor comparten las cookies.

El servlet puede recorrer la matriz de cookies y recuperar los atributos mediante los métodos *`getName`* y *`getValue`*.

```
for (int i = 0; i < cookies.length; i++) {
   nombre = cookies[i].getName();
   valor = cookies[i].getValue();
   // Operaciones
}
```

TIEMPO DE VIDA DE UNA COOKIE

```
micookie.setMaxAge(seconds);
```

- seconds > 0 => la cookie se almacenará persistentemente en el navegador durante ese número de segundos. Los navegadores suelen almacenar este tipo de cookies en ficheros locales
- seconds == 0 => eliminar la cookie
- seconds < 0 => la cookie no se almacenará persistentemente y dejará de existir cuando el navegador termine su ejecución. Los navegadores mantienen este tipo de cookies en memoria.

Las cookies no son un mecanismo seguro para el usuario. Si un usuario tiene acceso al fichero de cookies de otra persona, puede copiar las cookies *`loginName`* y *`password`* a su fichero de cookies, y por tanto, entrar en la aplicación web con su identidad. Los sitios web que tienen la opción de “recordar mi password" suelen advertir del problema.

Si el navegador acepta cookies, cuando se crea una sesión, el servidor de aplicaciones web le envía al navegador la cookie `jsessionid`

Si el navegador no acepta cookies, y se quiere hacer uso de sesiones, es preciso que el programador codifique todas las URLs para que lleven incrustado `jsessionid`

```
../index.jsp;jsessionid=9fab993aca36c8a3af423ba
```

Cuando el servidor recibe una petición HTTP parsea la URL y utiliza el valor de `jsessionid` para asociar la petición con la sesión

Conclusión: para que una aplicación que usa sesiones funcione tanto si el usuario acepta cookies como si no, debe aplicar URL rewriting a todas las URLs que ve el navegador. Si el navegador acepta cookies, `response.encodeURL` y `response.encodeRedirectURL` no modifican la URL.

Las cookie se obtienen de la petición HTTP (objeto request) y se mandan a la respuesta HTTP (objeto response) siempre que el cliente acepte las cookies. Una vez obtenida una cookie de la petición, ésta se puede leer, modificar y alterar su fecha de expiración (cambiando su edad máxima). Además, en cualquier momento se puede agregar nuevas cookies a las ya existentes.

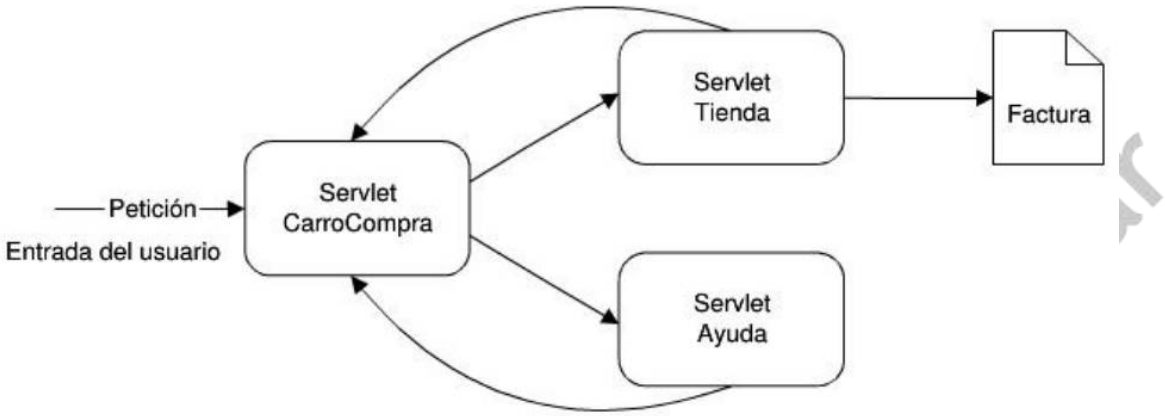
```
<!-- Cookies.jsp: Este programa puede comportarse distinto en diferentes
browsers -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP con Cookies</title>
</head>
<body>
   Sesion: <%=session.getId()%>
   <%
      Cookie[] cookies = request.getCookies();
      for (int i = 0; i < cookies.length; i++) {
         %>
            Cookie con nombre: <%=cookies[i].getName()%> <br>
            Valor: <%=cookies[i].getValue()%> <br>
            Duración máxima en segundos: <%=cookies[i].getMaxAge()%> <br>
         <%
            cookies[i].setMaxAge(5);
         %>
         Nueva duración en segundos: <%=cookies[i].getMaxAge()%> <br>
   <%
     }
   %>
   <%!int count = 0;
      int dcount = 0;%>
   <%
      response.addCookie(new Cookie("Matias" + count++, "pass" + dcount++));
   %>
</body>
</html>
```

IDENTIFICAR AL CLIENTE

En ocasiones será necesario identificar al cliente desde el que un usuario hacer la petición. Las cookies también pueden utilizarse para almacenar información que permita identificar a un usuario. Por ejemplo si el usuario esta realizando una compra online se puede registrar en una cookie su identificador de cliente, con el fin de identificarlo en el sitio y llevar así un seguimiento de sus compras hasta finalizar.

Por ejemplo en la aplicación “Carro de compra" podría constar de tres servlets, *CarroCompra*,

Tienda y *Ayuda*, y de un documento HTML para emitir la factura, *pagar.html*.



El servlet *CarroCompra* realizaría las siguientes operaciones:

- obtendría el identificador de sesión actual buscando entre las cookies recibidas. Si el identificador de sesión no hubiera sido enviado por el cliente, generaría uno. Después tendría que asegurarse de enviárselo al cliente con la respuesta (a través de response).
- Para poder el servlet generar un identificador de sesión único utiliza un objeto de la clase `UID`. Para enviarlo en las cabeceras de respuesta o de petición, hay que convertirlo en un objeto `String` y codificar cualquier carácter especial para lo cual se utiliza el método `envode` de la clase **`java.net.URLEncoder`**.
- Asociaría los artículos con el identificador de sesión. De esta forma, entre petición y petición de un mismo usuario podrían recuperarse los artículos comprados hasta el momento por este, para a continuación continuar con la compra.
- Generaría un formulario para mostrar, por cada petición, los artículos actualmente comprados, y para dar la opción de continuar con la compra o finalizarla solicitando la factura.
- Otro Servlet proporcionaría ayuda en linea relacionada con esta aplicación, carro de la compra, o con otras.

8.8.4 Sesiones con el API Java Servlet

Para permitir dar seguimiento a las sesiones, los servlets proporcionan la API **`HttpSession`**. Cuando un usuario accede por primera vez a un sitio web, le es asignado un objeto **`HttpSession`** nuevo (el cual esta asociado al objeto **`HttpServletRequest`**) y un identificador único ID utilizado para emparejar el ID (usuario) y el objeto **`HttpSession`** en sucesivas peticiones.

Esta interfaz se ha construido sobre las cookies y la reescritura URL. De hecho, muchos servidores usan cookies si el navegador web las soporta, pero cuando las cookies no son soportadas o están desactivadas, pasan automáticamente a reescribir el URL. Lo importante es que el desarrollador de servlets no tiene que manipular explícitamente las cookies o la información añadida al URL.

Para realizar el seguimiento de una sesión deberemos:

1. Obtener la sesión (un objeto **`HttpSession`**) correspondiente a un usuario determinado
2. Almacenar datos en, u obtener datos de, el objeto **`HttpSession`**.
3. Opcionalmente, cancelar la sesión.

Para obtener la sesión correspondiente a un usuario o crearla si no existiera:

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

El objeto **`HttpSession`** tiene una estructura de datos que permite almacenar un numero de claves y sus valores asociados. Para leer el valor de un atributo se utilizara ***`getAttribute`*** y para asignar un valor ***`setAttribute`***. Para recuperar el nombre de todos los atributos en un objeto **`Enumeration`** se utilizara el método ***`getAttributeNames`***.

Hay otra versión del método ***`getSession`*** que admite un parámetro que, con valor `false`, no establece la nueva sesión si ésta no existía.

En las páginas JSP el objeto implícito session permite acceder a la sesión actual (es un objeto de tipo **`HttpSession`**). Si no se desearán registrar sesiones, entonces habría que usar la directiva: `<%@ page session="false"%>`

La sesión mantiene referencias a todos los objetos almacenados pero tan pronto como la sesión es invalidada o el plazo de los objetos expira, los objetos se marcan para su recolección como basura.

Los pasos para utilizar sesiones suelen ser:

1. La sesión se obtiene mediante el método ***`getSession`*** del objeto `request` (en JSP ya hay creado un objeto llamado session). Realmente lo que ocurre es que se creará
2. Se pueden utilizar los métodos informativos de la clase **`HttpSession`** para obtener información de la sesión (***`getId`***, ***`getCreationTime`***, ***`getLastAccessedTime`***, ***`isNew`***,...)
3. Se utiliza el método ***`setAttribute`*** para establecer atributos de la sesión y ***`getAttribute`*** para recuperar atributos de la sesión. Esto permite grabar información en la sesión y recuperarla mientras la sesión esté activa.
4. El método ***`removeAttribute`*** permite eliminar un atributo.
5. La sesión se puede destruir de dos formas:

- La llamada al método ***`invalidate`*** de la sesión; que elimina la sesión actual.
- El cliente sobrepasa el límite de tiempo de inactividad. El intervalo de espera máxima se establece en el archivo de despliegue *`web.xml`* de la aplicación web:

```
<web-app>
...
<session-config>
   <session-timeout>30</session-timeout>
</session-config>
```

```
...
</web-app>
```

El intervalo se establece en minutos (en el ejemplo el tiempo de espera máximo será de 30 minutos).

Hay que tener en cuenta que normalmente esta gestión se realiza por cookies. Pero esto causa un problema: qué ocurre si el usuario desactiva las cookies. En ese caso se podría detectar y avisar (habría que comprobar si se graban los datos o no), o mejor utilizar paso de datos sin usar cookies.

Esto se consigue utilizando el método uso de sesiones por URL. Para ello al llamar a las direcciones, en lugar de poner su URL sin más, hay que utilizar el método *encodeURL* del objeto response. Ese método recibe una dirección URL y devuelve la URL incluyendo la sesión. Ejemplo:

```
out.println("<a href=\"/app1/prueba.jsp\">...");
```

El código anterior colocaría en la página un enlace a la dirección /app1/prueba.jsp. Si el navegador acepta cookies, además grabará los datos de la sesión.

```
out.println("<a href=" + response.encodeUrl(\"/app1/prueba.jsp\") + ">.....");
```

Este código hace lo mismo pero incluye en la URL la sesión, por lo que se permitirá usar los datos de la sesión haya cookies o no.

CANCELAR UNA SESIÓN

Una sesión normalmente expira automáticamente después de un tiempo de inactividad, o, manualmente cuando es explícitamente cancelada por un servlet. En este instante, todos los datos mantenidos por el objeto session son eliminados del sistema.

Con el método *invalidate()* se finaliza de inmediato una sesión, destruyendo todos los objetos que han sido almacenados.

8.8.5 *EVENTOS DE SESIÓN*

Las sesiones pueden producir eventos que pueden ser escuchados mediante clases que implementen alguna de estas interfaces.

interfaz **`HttpSessionListener`**

Sirve para escuchar eventos de sesión. Ocurren estos eventos en la creación o en la destrucción (o invalidación) de la sesión. Los métodos que define son:

- **`void sessionCreated(HttpSessionEvent e)`**. Ocurre cuando se crea una nueva sesión en la aplicación.
- **`void sessionDestroyed(HttpSessionEvent e)`**. Ocurre cuando se elimina una sesión.

Cualquier objeto podría implementar esta interfaz, pero se necesita indicar este objeto en el archivo web.xml

```
<web-app>
...
   <listener>
         <listener-class>CLaseEscuchadora</listener-class>
   </listener>
...
</web-app>
```

La clase escuchadora es la encargada de gestionar las acciones necesarias en la creación o eliminación de una sesión.

interfaz **`HttpSessionActivationListener`**

Sirve para escuchar eventos de activación de la sesión. Los métodos que define son:

- **`void sessionDidActivate(HttpSessionEvent e)`**. Ocurre cuando la sesión se activa.
- **`void sessionWillPassivate(HttpSessionEvent e)`**. Ocurre cuando la sesión pasa a estado de pasiva (no activa)

clase **`HttpSessionEvent`**

Representa los eventos de sesión. Deriva de la clase **`java.util.EventObject`** a la que añade el método getSession que obtiene el objeto **`HttpSession`** que lanzó el evento.

interfaz **`HttpSessionBindingListener`**

Sirve para escuchar eventos de adición o eliminación de atributos en la sesión. Métodos:

- **`void valueBound(HttpSessionBindingEvent e).`** Ocurre cuando se está añadiendo un atributo
- **`void valueUnbound(HttpSessionBindingEvent e)`**. Ocurre cuando un atributo se está eliminando de la sesión.

Son los objetos que se utilizan como atributos de la sesión los que pueden implementar esta interfaz.

interfaz **`HttpSessionAttributeListener`**

Similar a la anterior. Permite escuchar cuándo el estado de la sesión cambia.

- **`void attributeAdded(HttpSessionBindingEvent e)`**. Ocurre cuando se ha añadido un atributo a una sesión
- **`void attributeRemoved(HttpSessionBindingEvent e)`**. Ocurre cuando un atributo es eliminado de la sesión.
- **`void attributeReplaced(HttpSessionBindingEvent e)`**. Ocurre cuando se reemplazó un atributo por otro. Esto sucede si se utiliza *`setAttirbute`* con el mismo atributo pero diferente valor.

clase **`HttpSessionBindingEvent`**

Clase que define los objetos capturados por las dos interfaces anteriores. Deriva de **`java.util.EventObject`** y añade estos métodos:

- **`String`** *`getName()`*. Devuelve el nombre del atributo implicado en el evento.
- **`Object`** *`getValue()`*. Devuelve el nombre del objeto implicado en el evento.
- **`HttpSession`** *`getSession()`*. Devuelve el nombre de la sesión relacionada con el evento.

```
<!-- session.jsp: Ejemplo de una sesion -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP con session</title>
</head>
<body>
   <h1> Sesion: <%=session.getId()%></h1>
   <p>La session fue creada el<%=session.getCreationTime()%> </p>
   <p>Duración usual de la session:<%=session.getMaxInactiveInterval()%> </p>
   <p>Reducción de la duración de la session a 5 segundos <
%session.setMaxInactiveInterval(5);%> </p>
   <p>Nueva duración de la session:<%=session.getMaxInactiveInterval()%> </p>
   <p>Depositar un objeto nuevo
   <%
   session.setAttribute("clave", new String("G4rc14"));
   %> </p>
   <p>Mostrar el objeto depositado:<%=session.getAttribute("clave")%> </p>
</body>
</html>
```

```
<!-- session1.jsp: Ejemplo de una sesion con invalidación -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP con session 1</title>
</head>
<body>
   <h1>Sesion: <%=session.getId()%> </h1>
   <%
   if (session.getAttribute("clave") == null) {
         session.setAttribute("clave", new String("G4rc14"));
         out.println("<p>Escribir nueva clave:" + session.getAttribute("clave")
+ "</p>");
   } else {
         out.println("<p>Recuperar clave:" + session.getAttribute("clave") +
"</p>");
   }
   %>
   <form type="POST" action="session2.jsp">
         <input type="submit" name="invalidar" value="invalidar">
   </form>
   <form type="POST" action="session3.jsp">
         <input type="submit" name="conservar" value="conservar">
   </form>
</body>
</html>
```

Los archivos session2.jsp y session3.jsp son llamados por session1.jsp.

```
<!-- session2.jsp: Ejemplo de una sesion con invalidación -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP con session 2</title>
</head>
<body>
   <h1>Sesion: <%=session.getId()%> </h1>
   <p>Recuperar la clave: <%=session.getValue("clave")%> </p>
   <p>Invalidar la session <% session.invalidate(); %> </p>
   <p>Recuperar la clave: <%=session.getValue("clave")%> </p>
   <form type="POST" action="session1.jsp">
         <input type="submit" name="submit" value="regresar">
   </form>
</body>
</html>
```

```
<!-- session3.jsp: Ejemplo de una sesion con invalidación -->
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<!DOCTYPE html>
<html>
<head>
   <title>JSP con session 3</title>
</head>
<body>
   <h1>Sesion: <%=session.getId()%> </h1>
   <p>Recuperar la clave: <%= session.getValue("clave") %> </p>
   <form type="POST" action="session1.jsp">
         <input type="submit" name="submit" value="regresar">
   </form>
</body>
</html>
```

BIBLIOGRAFÍA

- Ceballos, Fco. Javier, “JAVA Interfaces Gráficas y Aplicaciones para Internet" 4ta Ed. (Ra-Ma 2015)
- Kuhn, Monica, “Apuntes de Programación II" INSPT/UTN (2014)
- Sanchez, Jorge, “JAVA 2" (2004)
- Gómez Fuentes, María del Carmen y Cervantes Ojeda, Jorge, “Introducción a la Programación Web con Java: JSP y Servlets, JavaServer Faces" (2017)

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar