Escuela Superior de Ingenieros Industriales
Industri Injineruen Goimailako Eskola

UNIVERSIDAD DE NAVARRA - NAFARROAKO UNIBERTSITATEA

Aprenda Informática ...

Aprenda Servlets de Java

como si estuviera en segundo

San Sebastián, Abril 1999

Javier García de Jalón • José Ignacio Rodríguez • Aitor Imaz

Aprenda Java

como si estuviera en primero

Javier García de Jalón

José Ignacio Rodríguez

Aitor Imaz

Perteneciente a la colección : *"Aprenda …, como si estuviera en primero"*

ÍNDICE

1 INTRODUCCIÓN

1.1 INTRODUCCIÓN A INTERNET/INTRANET

1.1.1 Introducción histórica

La red ***Internet*** es hoy día la red de ordenadores más extensa del planeta. Para ser más precisos, ***Internet*** es una red que enlaza centenares de miles de redes locales heterogéneas**.**

En 1990, ***Tim Berners-Lee***, un joven estudiante del Laboratorio Europeo de Física de Partículas (***CERN***) situado en Suiza, desarrolló un nuevo sistema de distribución de información en ***Internet*** basado en páginas ***hipertexto***, al que denominó ***World Wide Web*** (La "telaraña mundial"). La revolución de la ***Web*** había comenzado.

Realmente, el concepto de documento ***hipertexto*** no es nuevo: fue introducido por ***Ted Nelson*** en 1965 y básicamente se puede definir como *texto de recorrido no secuencial*. Clicando en las palabras con ***enlaces*** (links) se puede acceder al documento al que apuntan, que normalmente contiene una información más detallada sobre el concepto representado por las palabras del enlace. De ordinario, las palabras del enlace aparecen subrayadas y de un color diferente al del resto del documento, para que puedan diferenciarse fácilmente. Una vez que han sido clicadas cambian de color, para indicar que el documento al que apuntan ya ha sido visitado. Lo realmente novedoso de la ***Web*** es la aplicación del concepto de ***hipertexto*** a la inmensa base de información accesible a través de ***Internet***.

Por otra parte, lo que inicialmente se había concebido como un sistema de páginas ***hipertexto***, se ha convertido posteriormente en un verdadero sistema ***hipermedia***, en el que las páginas permiten acceder a imágenes, sonidos, videos, etc. Ello ha incrementado aún más el atractivo de la ***Web***.

Además de ***Internet***, existen en la actualidad numerosas ***Intranets***, es decir redes basadas en los mismos concepto de ***hipertexto*** e ***hipermedia*** y en las mismas tecnologías que ***Internet***, pero con un ámbito mucho más limitado. Por lo general, las ***Intranets*** se reducen al marco de una empresa, de una institución, de un centro educativo, etc. En general carecen de interés para otros usuarios del exterior, por el tipo de información que ofrecen. Por ejemplo, la ***Web*** de alumnos de la ESIISS puede ser considerada como una ***Intranet***: la información sobre asignaturas, horarios, exámenes, etc. no tiene gran interés para usuarios que no sean alumnos de la Escuela. De ahí que esté orientada a un uso interno. Esto no quiere decir que su acceso esté prohibido o restringido para usuarios externos: de hecho no lo está. Sin embargo, a diferencia de las ***Intranets*** universitarias, las ***Intranets*** empresariales sí que suelen tener limitados los accesos externos.

1.1.2 Redes de ordenadores

Una ***red*** es una ***agrupación de computadores***. Mediante una red, se posibilita el intercambio de información entre ordenadores de un modo eficiente y transparente. Una red permite ver los discos de otros ordenadores como si fueran discos locales. Según sea la estructura de dicha agrupación, o según el número de ordenadores integrados en ella se pueden establecer diferentes clasificaciones:

- ***Red Local*** (LAN: Local Area Network). De ordinario es una red dentro de un mismo edificio, como por ejemplo las redes de alumnos o de profesores de la ESIISS.

- ***Red de campus*** (CAN: Campus Area Network). Es una red que une distintos edificios dentro de una zona geográfica limitada, por ejemplo el campus de una universidad. De ordinario todos los cables por los que circula la información son privados.
- ***Red de ciudad*** (MAN: Metropolitan Area Network). Se trata de una red que une distintos edificios dentro de un área urbana. En la transmisión de la información interviene ya una empresa de telecomunicaciones, que podría ser de ámbito local o regional.
- **Red de área extensa** (WAN: Wide Area Network). En este caso la red puede unir centros dispersos en una zona geográfica muy amplia, en ocasiones por todo el mundo. Es la red típica de las empresas multinacionales. En la transmisión de la información deberán intervenir múltiples empresas de telecomunicaciones, como por ejemplo Euskaltel, Telefónica, BT, ATT, etc. ***Internet*** puede ser considerada como la WAN más conocida y extensa que existe en la actualidad.

Hay que mencionar la *jerarquía* y *estructuración* existente en las redes: unos ordenadores poseen unos derechos que otros no poseen (tienen accesos a archivos a los que otros no pueden acceder, los ordenadores con más jerarquía pueden controlar a los de menor rango, etc.).

1.1.3 Protocolo TCP/IP

Lo que permite que ordenadores remotos con procesadores y sistemas operativos diferentes se entiendan y en definitiva que ***Internet*** funcione como lo hace en la actualidad, es un conjunto de instrucciones o reglas conocidas con el nombre de ***protocolo***. La ***Internet*** utiliza varios protocolos, pero los que están en la base de todos los demás son el ***Transport Control Protocol*** (**TCP**) y el llamado ***Internet Protocol*** (***IP***), o en definitiva ***TCP/IP*** para abreviar. Se trata de una serie de reglas para mover de un ordenador a otro los datos electrónicos descompuestos en ***paquetes***, asegurándose de que todos los paquetes llegan y son ensamblados correctamente en su destino. Todos los ordenadores en ***Internet*** utilizan el protocolo ***TCP/IP***, y gracias a ello se consigue eliminar la barrera de la heterogeneidad de los ordenadores y resolver los problemas de direccionamiento.

1.1.4 Servicios

Sobre la base la infraestructura de transporte de datos que proporciona el protocolo ***TCP/IP*** se han construido otros protocolos más específicos que permiten por ejemplo enviar correo electrónico (***SMTP***), establecer conexiones y ejecutar comandos en máquinas remotas (***TELNET***), acceder a foros de discusión o *news* (***NNTP***), transmitir ficheros (***FTP***), conectarse con un servidor web (***HTTP***), etc. A estas capacidades de ***Internet*** se les llama ***servicios***. A continuación se revisan los más conocidos.

1.1.4.1 Correo electrónico

El correo electrónico o ***e-mail*** permite mantener correspondencia con usuarios en cualquier parte del mundo. Respecto al correo tradicional tiene la ventaja de que es mucho más rápido y sencillo de utilizar: es una manera muy fácil de enviar o recibir mensajes y ficheros, con el consiguiente ahorro de papel, lo que también supone una ventaja ecológica.

El correo electrónico tiene también ventajas económicas: es más barato que los servicios comerciales y carece de sobrecargas por larga distancia, siendo a su vez rápido y efectivo en el coste. El protocolo que se utiliza para el correo es el llamado **SMTP** (***Simple Mail Transfer Protocol***).

1.1.4.2 Ejecutar comandos en ordenadores remotos (Telnet)

Mediante ***Telnet*** es posible conectarse a un ordenador remoto en el que se tiene una cuenta de usuario o simplemente que está abierto a cualquier usuario. Tradicionalmente ***Telnet*** se ha utilizado para acceder a servicios de bases de datos y catálogos de bibliotecas. ***Telnet*** abre la posibilidad de conectarse a una cuenta remota gracias a ***Internet***. El servicio ***Telnet*** hace que se pueda estar conectado a un servidor remoto mediante una consola Unió (en cierta forma similar a la de MS-DOS), de igual manera que si la conexión se realizara en el propio ordenador. Todo lo que se escribe desde un teclado es redireccionado al ordenador remoto. De igual manera todo lo que el ordenador remoto devuelva como respuesta es redireccionado al monitor del usuario. No importa la distancia que haya entre ambos.

A diferencia del e-mail, ***Telnet*** establece una conexión ***permanente*** y ***síncrona*** entre los ordenadores cliente y servidor, conexión que permanece hasta que explícitamente es cortada por una de las dos partes.

1.1.4.3 Transferencia de ficheros (Ftp)

El servicio ***Ftp*** (***File Transfer Protocol***) es una parte importante de ***Internet***. ***Ftp*** permite transferir bidireccionalmente cualquier tipo de archivos con cualquiera de los miles de ordenadores remotos que tengan un servidor ***Ftp***. Se pueden transferir archivos ejecutables, de gráficos, sonido, vídeo o cualquier otro tipo. Al igual que ***Telnet***, ***Ftp*** establece conexiones síncronas y permanentes. Para utilizar el servicio ***Ftp*** suele ser necesario proporcionar un ***nombre de usuario*** y un ***password***.

Es muy frecuente encontrar servidores ***Ftp*** abiertos a todo el mundo y que permiten sólo lectura de ficheros (no escritura). Muchas empresas como ***Microsoft***, ***Sun***, ***Netscape***, etc. utilizan este sistema para distribuir software y utilidades gratuitas. En ocasiones, para conectarse a este tipo de servicio hay que dar como nombre de usuario la palabra ***anonymous***, y como password la propia dirección de correo electrónico.

1.1.4.4 World Wide Web

La ***World Wide Web***, o simplemente ***Web***, es el sistema de información más completo y actual, que une tanto elementos multimedia como hipertexto. De hecho, tomando el todo por la parte, con mucha frecuencia la ***Web*** se utiliza como sinónimo de ***Internet***.

El ***World Wide Web (WWW)*** es el resultado de cuatro ideas o factores:

1. La idea de ***Internet*** y los protocolos de transporte de información en que está basada.
2. La concepción de Ted Nelson de un sistema de ***hipertexto***, extendida a la red.
3. La idea de programas ***cliente*** que interaccionan con programas ***servidores*** capaces de enviar la información en ellos almacenada. Para la ***Web***, esto se hace mediante el protocolo ***HTTP*** (***HyperText Transfer Protocol***).
4. El concepto de *lenguaje anotado* (*Markup language*) y más en concreto del lenguaje ***HTML*** (***HyperText Markup Language***), que no se explicará en este documento, pero del que conviene tener una cierta idea (ver *http://jgjalon.ceit.es/Ayudainf/CursoHTML/Curso01.htm*).

HTML es una herramienta fundamental de ***Internet***. Gracias al ***hipertexto***, desde una página ***Web*** se puede acceder a cualquier otra página ***Web*** almacenada en un servidor ***HTTP*** situado en cualquier parte del mundo. Todo este tipo de operaciones se hacen mediante un programa llamado ***browser*** o ***navegador***, que básicamente es un programa que reconoce el lenguaje HTML, lo procesa y lo representa en pantalla con el formato más adecuado posible..

Hoy en día pueden encontrarse ***Webs*** relacionadas con cualquier área de la sociedad: educación, empresa, negocios, política, música, ocio, deportes, etcétera.

1.1.4.5 Grupos de discusión (News)

Los ***news groups*** o grupos de discusión son foros globales para la discusión de temas específicos. Son utilizados con el fin de discutir (en el buen sentido de la palabra) e intercambiar información, que versa sobre gran riqueza y variedad de temas. Estas discusiones suelen ser públicas, es decir, accesibles por personas de todo el mundo interesadas en el tema. Las discusiones pueden ser ***libres*** (cada usuario que desea intervenir lo hace sin limitación alguna) o ***moderadas*** (un moderador decide si las intervenciones se incluyen o no).

1.2 PROTOCOLO HTTP Y LENGUAJE HTML

Anteriormente se ha visto lo que son los protocolos de ***Internet*** y algunos de sus servicios. Por lo visto hasta el momento, se pueden enviar/recibir ficheros de cualquier tipo entre ordenadores conectados a ***Internet***, se puede enviar correo electrónico, se puede conectar a un servidor remoto y ejecutar comandos, etc. Sin embargo, ninguno de esos servicios permiten la posibilidad de colaborar en la creación de un entorno hipertexto e hipermedia, es decir, no se pueden pedir datos a un ordenador remoto para visualizarlos localmente utilizando **TCP/IP**. Es por ello que en 1991 se creó el protocolo llamado ***HTTP*** (***HyperText Transport Protocol***).

Una de las características del protocolo ***HTTP*** es que ***no es permanente***, es decir, una vez que el servidor ha respondido a la petición del cliente la conexión se pierde y el servidor queda en espera, al contrario de lo que ocurre con los servicios de ***ftp*** o ***telnet***, en los cuales la conexión es permanente hasta que el usuario o el servidor transmite la orden de desconexión. La ***conexión no permanente*** tiene la ventaja de que es más difícil que el servidor se colapse o sature, y el inconveniente de que no permite saber que es un mismo usuario el que está realizando diversas conexiones (esto complica la seguridad cuando los accesos se hacen con ***password***, pues no se puede pedir el password cada vez que se realiza una conexión para pedir una nueva página; el password sólo se debería pedir la primera vez que un usuario se conecta).

Se llama ***mantener la sesión*** a la capacidad de un servidor ***HTTP*** y de sus programas asociados para reconocer que una determinada solicitud de un servicio pertenece a un usuario que ya había sido identificado y autorizado. Esta es una característica muy importante en todos los programas de comercio electrónico.

La ventaja del protocolo ***HTTP*** es que se pueden crear recursos multimedia localmente, transferirlos fácilmente a un servidor remoto y visionarlos desde donde se han enviado o desde cualquier otro ordenador conectado a la red. El protocolo ***HTTP*** es una herramienta muy poderosa, que constituye la esencia del ***World Wide Web***.

Ya se ha dicho que para la creación de las páginas ***Web*** en ***Internet*** se utiliza el lenguaje ***HTML*** (***HyperText Markup Language***). Es un lenguaje muy simple, cuyo código se puede escribir con cualquier editor de texto como ***Notepad***, ***Wordpad*** o ***Word***. Se basa en comandos o tags reconocibles por el browser y que van entre los símbolos '<' y '>'. Como tutorial introductorio puede utilizarse el contenido en la dirección *http://jqjalon.ceit.es/Ayudainf/CursoHTML/Curso01.htm*.

El lenguaje ***HTML*** es tan importante que se han creado muchos editores especiales, entre los que destaca ***Microsoft FrontPage 98***. Además, las aplicaciones más habituales (tales como ***Word***, ***Excel*** y ***PowerPoint***) tienen posibilidad de exportar ficheros ***HTML***. No es pues nada difícil aprender a crear páginas ***HTML***.

1.3 URL (UNIFORM RESOURCE LOCATOR)

Todo ordenador en ***Internet*** y toda persona que use ***Internet*** tiene su propia dirección electrónica (***IP address***). Todas estas direcciones siguen un mismo formato. Para el ayudante Pedro Gómez, de la Escuela Superior de Ingenieros Industriales de San Sebastián, su dirección podría ser:

```
pgomez@gaviota.ceit.es
```

donde ***pgomez*** es el identificador ID o nombre de usuario que Pedro utiliza para conectarse a la red. Es así como el ordenador le conoce. La parte de la dirección que sigue al símbolo de *arroba* (@) identifica al ordenador en el que está el servidor de correo electrónico. Consta de dos partes: el nombre del ordenador o ***host***, y un identificador de la red local de la institución, llamado ***dominio***. En este caso el ordenador se llama ***gaviota*** y el dominio es ***ceit.es***, que identifica a las redes de la ESIISS y del Centro de Estudios e Investigaciones Técnicas de Guipúzcoa (ceit), en España (es). Nunca hay espacios en blanco en una dirección de ***Internet***.

El nombre del servidor o ***IP address*** está dividido en este caso en tres campos que se leen de derecha a izquierda. El primer campo por la derecha es el identificador del país o, en el caso de EEUU, del tipo de institución. Entre los posibles valores de este campo se pueden encontrar los siguientes (utilizados como se ha dicho en Estados Unidos): ***com*** (organizaciones y empresas comerciales), ***gov*** (gobierno), ***int*** (organización internacional), ***mil*** (militar), ***net*** (organización de redes) y ***org*** (organizaciones sin ánimo de lucro).

Fuera de Estados Unidos el campo de la derecha se refiere al estado o país al que pertenece el servidor, como por ejemplo:

at:	Austria	au:	Australia
ca:	Canadá	ch:	Suiza
de:	Alemania	dk:	Dinamarca
es:	España	fi:	Finlandia
fr:	Francia	gr:	Grecia
jp:	Japón	uk:	Reino Unido

En realidad los ordenadores no se identifican mediante un nombre, sino mediante un número: el llamado número o ***dirección IP***, que es lo que el ordenador realmente entiende. Los nombres son para facilitar la tarea a los usuarios, ya que son más fáciles de recordar y de relacionar con la institución. Por ejemplo el número que corresponde al servidor ***gaviota.ceit.es*** es 193.145.249.23. Es evidente que es más fácil recordar el nombre que la ***dirección IP***. En ***Internet*** existen unos servidores especiales, llamados servidores de nombres o de direcciones, que mantienen unas tablas mediante las que se puede determinar la ***dirección IP*** a partir del nombre.

Así pues, ¿***qué es exactamente un URL***? Pues podría concebirse como la extensión del concepto de nombre completo de un archivo (path). Mediante un ***URL*** no sólo puede apuntarse a un archivo en un directorio en un disco local, sino que además tal archivo y tal directorio pueden estar localizados de hecho en cualquier ordenador de la red, con el mismo o con distinto sistema operativo. Las ***URLs*** posibilitan el direccionamiento de personas, ficheros y de una gran variedad de información, disponible según los distintos protocolos o servicios de ***Internet***. El protocolo más conocido es el ***HTTP***, pero ***FTP*** y las direcciones de ***e-mail*** también pueden ser referidas con un ***URL***. En definitiva, un ***URL*** es como la ***dirección completa*** de un determinado servicio: proporciona todos los datos necesarios para localizar el recurso o la información deseada.

En resumen, un ***URL*** es una manera conveniente y sucinta de referirse a un archivo o a cualquier otro recurso electrónico.

La sintaxis genérica de los ***URLs*** es la que se muestra a continuación:

```
método://servidor.dominio/ruta-completa-del-fichero
```

donde ***método*** es una de las palabras que describen el servicio: ***http, ftp, news, …*** Enseguida se verá la sintaxis específica de cada método, pero antes conviene añadir unas breves observaciones:

- En ocasiones el **URL** empleado tiene una sintaxis como la mostrada, pero acabada con una barra (/). Esto quiere decir que no se apunta a un archivo, sino a un ***directorio***. Según como esté configurado, el servidor devolverá el índice por defecto de ese directorio (un listado de archivos y subdirectorios de ese directorio para poder acceder al que se desee), un archivo por defecto que el servidor busca automáticamente en el directorio (de ordinario llamado *Index.htm* o *Index.html*) o quizás impida el acceso si no se conoce exactamente el nombre del fichero al que se quiere acceder (como medida de seguridad).
- ¿Cómo presentar un ***URL*** a otros usuarios? Se suele recomendar hacerlo de la siguiente manera:

  ```
  <URL: método://ordenador.dominio/ruta-completa-del-fichero>
  ```

  para distinguir así los ***URLs*** de los ***URIs*** (***Uniform Resource Identification***), que representan un concepto similar pero no idéntico.

A continuación se muestran las distintas formas de construir los ***URLs*** según los distintos servicios de ***Internet***:

1.3.1 URLs del protocolo HTTP

Como ya se ha dicho, ***HTTP*** es el protocolo específicamente diseñado para la ***World Wide Web***. Su sintaxis es la siguiente:

```
http://<host>:<puerto>/<ruta>
```

donde ***host*** es la dirección del servidor ***WWW***, el ***puerto*** indica a través de que "entrada" el servidor atiende los requerimientos ***HTTP*** (puede ser omitido, en cuyo caso se utiliza el valor por defecto, 80), y la ***ruta*** indica al servidor el ***path*** del fichero que se desea cargar (el ***path*** es relativo a un directorio raíz indicado en el ***servidor HTTP***).

Así, por ejemplo, *http://www.msn.com/index/prev/welcome.htm* accede a la ***Web*** de ***Microsoft Network***, en concreto al archivo ***welcome.htm*** (cuya ruta de acceso es ***index/prev***).

1.3.2 URLs del protocolo FTP

La sintaxis específica del protocolo ***ftp*** es la siguiente:

```
ftp://<usuario>:<password>@<host>:<puerto>/<cwd1>/<cwd2>/…/<cwdN>/<nombre>
```

Los campos ***usuario*** y ***password*** sólo son necesarios si el servidor los requiere para autorizar el acceso; en otro caso pueden ser omitidos; ***host*** es la dirección del ordenador en el que se está ejecutando el servidor ***ftp***; el ***puerto***, como antes, es una información que puede ser omitida (por defecto suele ser el “21”); la serie de argumentos *<cwd1>/…/<cwdN>* son los comandos que el cliente debe ejecutar para moverse hasta el directorio en el que reside el documento; ***nombre*** es el nombre del documento que se desea obtener.

Así, por ejemplo, *ftp://www.msn.com/index/prev/welcome.htm* traerá el fichero ***welcome.htm*** (cuya ruta de acceso es ***index/prev***) del servidor ***ftp*** de ***Microsoft Network***.

1.3.3 URLs del protocolo correo electrónico (mailto)

La sintaxis del correo electrónico difiere bastante de las anteriores, y es como sigue:

```
mailto:<cuenta@dominio>
```

siendo *cuenta@dominio* la dirección de correo electrónico a la cual se desea enviar un mensaje. Por ejemplo, ***mailto:pgomez@ceit.es*** manda un mensaje a Pedro Gómez, en el CEIT, en España. En este caso se ha omitido el nombre del ordenador que contiene el servidor de correo electrónico. Una institución puede definir un servidor de correo electrónico por defecto, y en ese caso no hace falta incluir su nombre. Si en esa institución hay más servidores de correo electrónico habrá que especificar el nombre del ordenador (por ejemplo, ***pgomez@gaviota.ceit.es***).

Cuando en las páginas ***HTML*** de una ***web*** se quiere introducir una dirección de correo electrónico se utiliza este tipo de ***URL*** (ver por ejemplo las portadas o páginas de entrada de las ***webs*** de ***Informática 1*** y ***2***).

1.3.4 URLs del protocolo News (NNTP)

Para acceder a un ***grupo de noticias*** o *news*, se utiliza una dirección de un servidor según el esquema siguiente:

```
news:<grupo_de_news-nombre>
```

donde ***grupo_de_news-nombre*** es el nombre del grupo de ***news***, de ***Usenet*** (por ejemplo, ***comp.infosystems.www.providers***), que generalmente indicará al lector de ***news*** (los browsers como ***Netscape Navigator*** suelen tener un lector de ***news*** incorporado) los artículos disponibles en ese grupo de ***news***. Si este parámetro fuese un asterisco "*", el ***URL*** se referiría a todos los ***newsgroups*** que estén operativos. Hay que tener en cuenta que un cliente o lector de ***news*** debe estar bien configurado para saber dónde obtener artículos o ***grupos de news***, (normalmente de un servidor específico ***NNTP*****, tal** como ***news.cti.unav.es***; en Netscape Navigator esta configuración se hace en el cuadro de diálogo que se abre con ***Edit/Preferences/Mail&NewsGroups***).

1.3.5 URLs del protocolo Telnet

El ***URL*** necesario para crear una sesión *Telnet* en un servidor remoto en ***Internet***, se define mediante el protocolo *Telnet*. Su sintaxis es la siguiente:

```
telnet://<usuario>:<password>@<host>:<puerto>/
```

donde los parámetros ***user*** y ***password*** pueden ser omitidos si el servidor no los requiere, ***host*** identificar al ordenador remoto con el que se va a realizar la conexión y ***port*** define el puerto por el que dicho servidor atiende los requerimientos de servicios ***Telnet*** (puede ser igualmente omitido, siendo su valor por defecto "23").

1.3.6 Nombres específicos de ficheros

Es `posible acceder también directamente a un fichero concreto mediante el ***URL*** del método ***file***, que supone que dicho fichero puede ser obtenido por un cliente. Esto suele confundirse con el servicio ***ftp***. La diferencia radica en que ***ftp*** es un servicio específico para la transmisión de ficheros, mientras que ***file*** deja la forma de traer los ficheros a la elección del cliente, que en algunas circunstancias, bien podría ser el método ***ftp***. La sintaxis para el método ***file*** es la que se muestra a continuación:

```
file://<host>/<ruta-de-acceso>
```

Como siempre, ***host*** es la dirección del servidor y ***ruta-de-acceso*** es el camino jerárquico para acceder al documento (con una estructura directorio/directorio/…/nombre del archivo). Si el parámetro ***host*** se deja en blanco, se supondrá por defecto ***localhost***, es decir, que se van a traer ficheros del propio ordenador.

1.4 CLIENTES Y SERVIDORES

1.4.1 Clientes (clients)

Por su versatilidad y potencialidad, en la actualidad la mayoría de los usuarios de ***Internet*** utilizan en sus comunicaciones con los servidores de datos, los ***browsers*** o ***navegadores***. Esto no significa que no puedan emplearse otro tipo de programas como clientes *e-mail*, *news*, etc. para aplicaciones más específicas. De hecho, los browsers más utilizados incorporan lectores de mail y de news.

En la actualidad los browsers más extendidos son ***Netscape Communicator*** y ***Microsoft Internet Explorer***. En el momento de escribir estas notas (abril 1999), el primero está en la versión 4.51 y el segundo en la 5.0. Ambos acaparan una cuota de mercado que cubre prácticamente a todos los usuarios.

A pesar de que ambos cumplen con la mayoría de los estándares aceptados en la ***Internet***, cada uno de ellos proporciona soluciones adicionales a problemas más específicos. Por este motivo, muchas veces será necesario tener en cuenta qué tipo de browser se va a comunicar con un servidor, pues el resultado puede ser distinto dependiendo del browser empleado, lo cual puede dar lugar a errores.

Ambos browsers soportan ***Java***, lo cual implica que disponen de una ***Java Virtual Machine*** en la que se ejecutan los ficheros ****.class*** de las ***Applets*** que traen a través de ***Internet***. Netscape es más fiel al estándar de ***Java*** tal y como lo define ***Sun***, pero ambos tienen la posibilidad de sustituir la ***Java Virtual Machine*** por medio de un mecanismo definido por ***Sun***, que se llama ***Java Plug-in*** (los ***plug-ins*** son aplicaciones que se ejecutan controladas por los browsers y que permiten extender sus capacidades, por ejemplo para soportar nuevos formatos de audio o video).

1.4.2 Servidores (servers)

Los ***servidores*** son programas que se encuentran permanentemente esperando a que algún otro ordenador realice una solicitud de conexión. En un mismo ordenador es posible tener simultáneamente servidores de los distintos servicios anteriormente mencionados (***HTTP***, ***FTP***, ***TELNET***, etc.). Cuando a dicho ordenador llega un requerimiento de servicio enviado por otro ordenador de la red, se interpreta el tipo de llamada, y se pasa el control de la conexión al ***servidor*** correspondiente a dicho requerimiento. En caso de no tener el ***servidor*** adecuado para responder a la comunicación, está será rechazada. Un ejemplo de rechazo ocurre cuando se quiere conectar a través de ***TELNET*** (típico de los sistemas ***UNIX***) con un ordenador que utilice ***Windows 95/98***.

Como ya se ha apuntado, no todos los servicios actúan de igual manera. Algunos, como ***TELNET*** y ***FTP***, una vez establecida la conexión, la mantienen hasta que el cliente o el servidor explícitamente la cortan. Por ejemplo, cuando se establece una conexión con un servidor de ***FTP***, los dos ordenadores se mantienen en contacto hasta que el cliente cierre la conexión mediante el comando correspondiente (***quit***, ***exit***, …) o pase un tiempo establecido en la configuración del servidor ***FTP*** o del propio cliente, sin ninguna actividad entre ambos.

La comunicación a través del protocolo ***HTTP*** es diferente, ya que es necesario establecer una comunicación o conexión distinta para cada elemento que se desea leer. Esto significa que en un documento ***HTML*** con 10 imágenes son necesarias 11 conexiones distintas con el servidor ***HTTP***, esto es, una para el texto del documento ***HTML*** con las *tags* y las otras 10 para traer las imágenes referenciadas en el documento ***HTML***.

La mayoría de los usuarios de ***Internet*** son ***clientes*** que acceden mediante un ***browser*** a los distintos ***servidores WWW*** presentes en la red. El servidor no permite acceder indiscriminadamente a todos sus ficheros, sino únicamente a determinados directorios y documentos previamente establecidos por el *administrador* de dicho servidor.

1.5 TENDENCIAS ACTUALES PARA LAS APLICACIONES EN INTERNET

En la actualidad, la mayoría de aplicaciones que se utilizan en entornos empresariales están construidos en torno a una arquitectura ***cliente-servidor***, en la cual uno o varios computadores (generalmente de una potencia considerable) son los ***servidores***, que proporcionan servicios a un número mucho más grande de ***clientes*** conectados a través de la red. Los ***clientes*** suelen ser PCs de propósito general, de ordinario menos potentes y más orientados al usuario final. A veces los servidores son intermediarios entre los clientes y otros servidores más especializados (por ejemplo los grandes servidores de bases de datos corporativos basados en ***mainframes*** y/o sistemas ***Unix***. En esta caso se habla se *aplicaciones de varias capas*).

Con el auge de ***Internet***, la arquitectura ***cliente-servidor*** ha adquirido una mayor relevancia, ya que la misma es el principio básico de funcionamiento de la ***World Wide Web***: un usuario que mediante un ***browser*** (***cliente***) solicita un servicio (páginas ***HTML***, etc.) a un computador que hace las veces de ***servidor***. En su concepción más tradicional, los servidores ***HTTP*** se limitaban a enviar una página ***HTML*** cuando el usuario la requería directamente o clicaba sobre un enlace. La interactividad de este proceso era mínima, ya que el usuario podía pedir ficheros, pero no enviar sus datos personales de modo que fueran almacenados en el servidor u obtuviera una respuesta personalizada. La Figura 1 representa gráficamente este concepto.

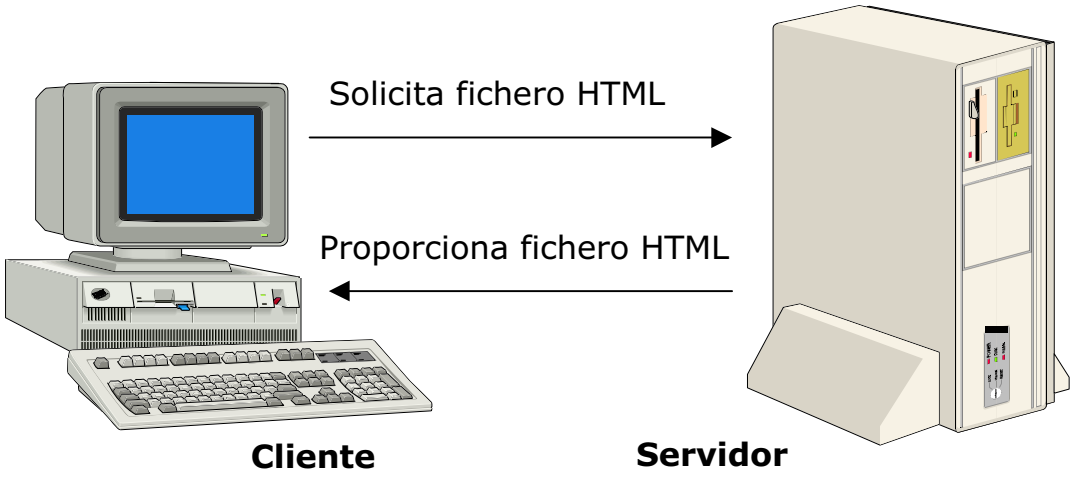

Figura 1. Arquitectura cliente-servidor tradicional.

Desde esa primera concepción del servidor ***HTTP*** como mero servidor de ficheros ***HTML*** el concepto ha ido evolucionando en dos direcciones complementarias:

1. Añadir más inteligencia en el ***servidor***, y
2. Añadir más inteligencia en el ***cliente***.

Las formas más extendidas de añadir inteligencia a los clientes (a las páginas ***HTML***) han sido ***Javascript*** y las ***applets de Java***. ***Javascript*** es un lenguaje relativamente sencillo, interpretado, cuyo código fuente se introduce en la página ***HTML*** por medio de los tags <SCRIPT> … </SCRIPT>; su nombre deriva de una cierta similitud sintáctica con ***Java***. Las ***applets de Java*** tienen u mucha más capacidad de añadir inteligencia a las páginas ***HTML*** que se visualizan en el browser, ya que son verdaderas clases de ***Java*** (ficheros ****.class***) que se cargan y se ejecutan en el cliente. Sobre las posibilidades de las ***applets de Java*** puede consultarse el manual "*Aprenda Java como si estuviera en Primero*".

De cara a estos apuntes tienen mucho más interés los caminos seguidos para añadir más inteligencia en el servidor ***HTTP***. La primera y más empleada tecnología ha sido la de los ***programas CGI*** (***Common Gateway Interface***), unida a los ***formularios HTML***.

Los ***formularios HTML*** permiten de alguna manera invertir el sentido del flujo de la información. Cumplimentando algunos campos con cajas de texto, botones de opción y de selección, el usuario puede definir sus preferencias o enviar sus datos al servidor. Un ejemplo de formulario bien conocido por los alumnos de la ESIISS es la ficha electrónica de ***Informática 1*** (*http://www1.ceit.es/Asignaturas/Informat1/Curso9899/General/FichaInf1.htm*). Cuando en un formulario ***HTML*** se pulsa en el botón ***Enviar*** (o nombre equivalente, como ***Submit***) los datos tecleados por el cliente se envían al servidor para su procesamiento.

¿Cómo recibe el servidor los datos de un formulario y qué hace con ellos? Éste es el problema que tradicionalmente han resuelto los ***programas CGI***. Cada formulario lleva incluido un campo llamado ***Action*** con el que se asocia el nombre de programa en el servidor. El servidor arranca dicho programa y le pasa los datos que han llegado con el formulario. Existen dos formas principales de pasar los datos del formulario al ***programa CGI***:

1. Por medio de una variable de entorno del sistema operativo del servidor, de tipo String (método ***GET***)
2. Por medio de un flujo de caracteres que llega a través de la entrada estándar (*stdin* o *System.in*), que de ordinario está asociada al teclado (método ***POST***).

En ambos casos, la información introducida por el usuario en el formulario llega en la forma de una única cadena de caracteres en la que el nombre de cada campo del formulario se asocia con el valor asignado por el usuario, y en la que los blancos y ciertos caracteres especiales se han sustituido por secuencias de caracteres de acuerdo con una determinada codificación. Más adelante se verán con más detenimiento las reglas que gobiernan esta transmisión de información. En cualquier caso, lo primero que tiene que hacer el ***programa CGI*** es decodificar esta información y separar los valores de los distintos campos. Después ya puede realizar su tarea específica: escribir en un fichero o en una base de datos, realizar una búsqueda de la información solicitada, realizar comprobaciones, etc. De ordinario, el ***programa CGI*** termina enviando al cliente (el navegador desde el que se envió el formulario) una página ***HTML*** en la que le informa de las tareas realizadas, le avisa de si se ha producido alguna dificultad, le reclama algún dato pendiente o mal cumplimentado, etc. La forma de enviar esta página ***HTML*** al cliente es a través de la salida estándar (*stduot* o *System.out*), que de ordinario suele estar asociada a la pantalla. La página ***HTML*** tiene que ser construida elemento a elemento, de acuerdo con las reglas de este lenguaje. No basta enviar el contenido: hay que enviar también todas y cada una de las ***tags***. En un próximo apartado se verá un ejemplo completo.

En principio, los ***programas CGI*** pueden estar escritos en cualquier lenguaje de programación, aunque en la práctica se han utilizado principalmente los lenguajes ***Perl***[1] y ***C/C++***. Un claro ejemplo de un ***programa CGI*** sería el de un formulario en el que el usuario introdujera sus datos personales para registrarse en un sitio web. El ***programa CGI*** recibiría los datos del usuario, introduciéndolos en la base de datos correspondiente y devolviendo al usuario una página ***HTML*** donde se le informaría de que sus datos habían sido registrados. La Figura 2 muestra el esquema básico de funcionamiento de los ***programas CGI***.

Figura 2. Arquitectura cliente-servidor interactiva para la WEB.

Es importante resaltar que estos procesos tienen lugar en el servidor. Esto a su vez puede resultar un problema, ya que al tener múltiples clientes conectados al servidor, el ***programa CGI*** puede estar siendo llamado simultáneamente por varios clientes, con el riesgo de que el servidor se llegue a saturar. Téngase en cuenta que cada vez que se recibe un requerimiento se arranca una nueva copia del ***programa CGI***. Existen otros riesgos adicionales que se estudiarán más adelante.

El objetivo de este capítulo es el estudio de la alternativa que ***Java*** ofrece a los ***programas CGI***: los ***servlets,*** que son a los servidores lo que los ***applets*** a los browsers. Se podría definir un ***servlet*** como ***un programa escrito en Java que se ejecuta en el marco de un servicio de red, (un servidor HTTP, por ejemplo), y que recibe y responde a las peticiones de uno o más clientes***.

En adelante, se supondrá que el lector está ya algo familiarizado con los conceptos básicos de la ***World Wide Web***. En caso de no ser así, referirse a la ayuda disponible en le página web de ***Informática 2***.

2 DIFERENCIAS ENTRE LAS TECNOLOGÍAS CGI Y SERVLET

La tecnología ***Servlet*** proporciona las mismas ventajas del lenguaje ***Java*** en cuanto a ***portabilidad*** (“*write once, run anywhere*”) y ***seguridad***, ya que un ***servlet*** es una ***clase*** de ***Java*** igual que cualquier otra, y por tanto tiene en ese sentido todas las características del lenguaje. Esto es algo de lo que carecen los ***programas CGI***, ya que hay que compilarlos para el sistema operativo del

[1] PERL es un lenguaje interpretado procedente del entorno Unix (aunque también existe en Windows NT), con grandes capacidades para manejar texto y cadenas de caracteres.

servidor y no disponen en muchos casos de técnicas de comprobación dinámica de errores en tiempo de ejecución.

Otra de las principales ventajas de los ***servlets*** con respecto a los ***programas CGI***, es la del rendimiento, y esto a pesar de que ***Java*** no es un lenguaje particularmente rápido. Mientras que los es necesario cargar los ***programas CGI*** tantas veces como peticiones de servicio existan por parte de los clientes, los ***servlets***, una vez que son llamados por primera vez, ***quedan activos en la memoria del servidor hasta que el programa que controla el servidor los desactiva***. De esta manera se minimiza en gran medida el tiempo de respuesta.

Además, los ***servlets*** se benefician de la gran capacidad de ***Java*** para ejecutar métodos en ordenadores remotos, para conectar con bases de datos, para la seguridad en la información, etc. Se podría decir que las ***clases estándar de Java*** ofrecen resueltos mucho problemas que con otros lenguajes tiene que resolver el programador.

3 CARACTERÍSTICAS DE LOS *SERVLETS*

Además de las características indicadas en el apartado anterior, los ***servlets*** tienen las siguientes características:

1. Son independientes del servidor utilizado y de su sistema operativo, lo que quiere decir que a pesar de estar escritos en ***Java***, el servidor puede estar escrito en cualquier lenguaje de programación, obteniéndose exactamente el mismo resultado que si lo estuviera en ***Java***.
2. Los ***servlets*** pueden llamar a otros ***servlets***, e incluso a métodos concretos de otros ***servlets***. De esta forma se puede distribuir de forma más eficiente el trabajo a realizar. Por ejemplo, se podría tener un ***servlet*** encargado de la interacción con los clientes y que llamara a otro ***servlet*** para que a su vez se encargara de la comunicación con una base de datos. De igual forma, los ***servlets*** permiten *redireccionar* peticiones de servicios a otros ***servlets*** (en la misma máquina o en una máquina remota).
3. Los ***servlets*** pueden obtener fácilmente información acerca del ***cliente*** (la permitida por el protocolo ***HTTP***), tal como su dirección ***IP***, el ***puerto*** que se utiliza en la llamada, el método utilizado (***GET***, ***POST,*** ...), etc.
4. Permiten además la utilización de ***cookies*** y ***sesiones***, de forma que se puede guardar información específica acerca de un usuario determinado, personalizando de esta forma la interacción cliente-servidor. Una clara aplicación es ***mantener la sesión*** con un cliente.
5. Los ***servlets*** pueden actuar como enlace entre el cliente y una o varias ***bases de datos*** en arquitecturas *cliente-servidor de 3 capas* (si la base de datos está en un servidor distinto).
6. Asimismo, pueden realizar tareas de ***proxy*** para un ***applet***. Debido a las restricciones de seguridad, un ***applet*** no puede acceder directamente por ejemplo a un servidor de datos localizado en cualquier máquina remota, pero el ***servlet*** sí puede hacerlo de su parte.
7. Al igual que los ***programas CGI***, los ***servlets*** permiten la generación dinámica de código ***HTML*** dentro de una propia página ***HTML***. Así, pueden emplearse ***servlets*** para la creación de contadores, banners, etc.

4 JSDK 2.0

El ***JSDK*** (***Java Servlet Developer Kit***), distribuido gratuitamente por ***Sun***, proporciona el conjunto de herramientas necesarias para el desarrollo de ***servlets***. En el momento en el que estas notas han sido escritas (abril 1999), la última versión disponible es la ***2.1***, pero al no estar todavía soportada por la mayor parte de los servidores ***HTTP***, en adelante se utilizará únicamente a la versión ***2.0*** del ***JSDK***, salvo donde se especifique lo contrario. El ***JSDK 2.0*** se encuentra disponible en la dirección de ***Internet*** http://java.sun.com/products/servlet/index.html**.** Se trata de un fichero de 950 Kbytes, llamado ***jsdk20-Win32.exe***, que está disponible en el directorio ***Q:\Infor2\Servlet*** de la red de Alumnos de la ESIISS, y que puede transportarse al propio ordenador en un simple disquete. El ***JSDK*** consta básicamente de 3 partes:

1. El ***API*** del ***JSDK***, que se encuentra diseñada como una ***extensión*** del ***JDK*** propiamente dicho. Consta de dos ***packages*** cuyo funcionamiento será estudiado en detalle en apartados posteriores, y que se encuentran contenidos en ***javax.servlet*** y ***javax.servlet.http***. Este último es una particularización del primero para el caso del protocolo **HTTP**, que es el que será utilizado en este manual, al ser el más extendido en la actualidad. Mediante este diseño lo que se consigue es que se mantenga una puerta abierta a la utilización de otros protocolos que existen en la actualidad ***(FTP, POP, SMTP,*** etc.***)***, o vayan siendo utilizados en el futuro. Estos ***packages*** están almacenados en un fichero ***JAR (\lib\jsdk.jar)***.
2. La ***documentación*** propiamente dicha del ***API*** y el código fuente de las clases (similar a la de los ***JDK 1.1*** y ***1.2***).
3. La aplicación ***servletrunner***, que es una simple utilidad que permite probar los ***servlets*** creados sin necesidad de hacer complejas instalaciones de servidores ***HTTP***.. Es similar en concepción al ***appletviewer*** del ***JDK***. Su utilización será descrita en un apartado posterior.

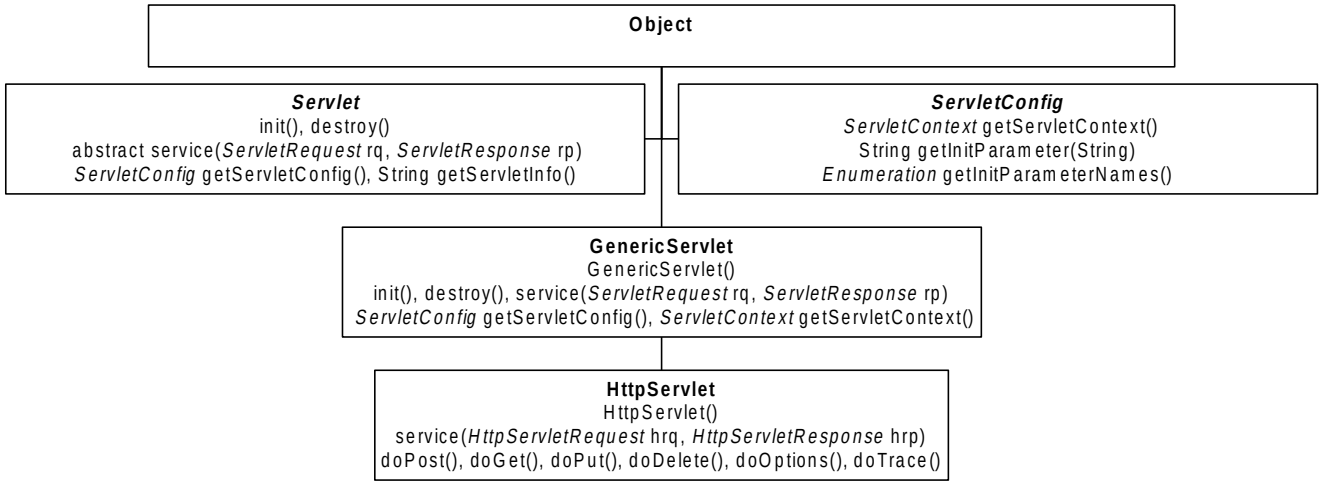

Figura 3. Jerarquía y métodos de las principales clases para crear servlets.

4.1 VISIÓN GENERAL DEL API DE JSDK 2.0

Es importante adquirir cuanto antes una visión general del ***API*** (*Application Programming Interface*) del ***Java Servlet Development Kit 2.0***, de qué clases e interfaces la constituyen y de cuál es la relación entre ellas.

El ***JSDK 2.0*** contiene dos packages: ***javax.servlet*** y ***javax.servlet.http***. Todas las clases e interfaces que hay que utilizar en la programación de ***servlets*** están en estos dos packages.

La relación entre las clases e interfaces de ***Java***, muy determinada por el concepto de ***herencia***, se entiende mucho mejor mediante una representación gráfica tal como la que puede verse en la Figura 3. En dicha figura se representan las ***clases*** con letra normal y las ***interfaces*** con *cursiva*.

La clase ***GenericServlet*** es una clase ***abstract*** puesto que su método ***service()*** es ***abstract***. Esta clase implementa dos interfaces, de las cuales la más importante es la interface ***Servlet***.

La interface ***Servlet*** declara los métodos más importantes de cara a la vida de un servlet: ***init()*** que se ejecuta sólo al arrancar el *servlet*; ***destroy()*** que se ejecuta cuando va a ser destruido y ***service()*** que se ejecutará cada vez que el *servlet* deba atender una solicitud de servicio.

Cualquier clase que derive de ***GenericServlet*** deberá definir el método ***service()***. Es muy interesante observar los dos argumentos que recibe este método, correspondientes a las interfaces ***ServletRequest*** y ***ServletResponse***. La primera de ellas referencia a un objeto que describe por completo la solicitud de servicio que se le envía al servlet. Si la solicitud de servicio viene de un formulario HTML, por medio de ese objeto se puede acceder a los nombres de los campos y a los valores introducidos por el usuario; puede también obtenerse cierta información sobre el cliente (ordenador y browser). El segundo argumento es un objeto con una referencia de la interface ***ServletResponse***, que constituye el camino mediante el cual el método ***service()*** se conecta de nuevo con el cliente y le comunica el resultado de su solicitud. Además, dicho método deberá realizar cuantas operaciones sean necesarias para desempeñar su cometido: escribir y/o leer datos de un fichero, comunicarse con una base de datos, etc. El método ***service()*** es realmente el corazón del servlet.

En la práctica, salvo para desarrollos muy especializados, todos los servlets deberán construirse a partir de la clase ***HttpServlet***, sub-clase de ***GenericServlet***.

La clase ***HttpServlet*** ya no es ***abstract*** y dispone de una implementación o definición del método ***service()***. Dicha implementación detecta el tipo de servicio o método ***HTTP*** que le ha sido solicitado desde el browser y llama al método adecuado de esa misma clase (***doPost()***, ***doGet()***, etc.). Cuando el programador crea una sub-clase de ***HttpServlet***, por lo general no tiene que redefinir el método ***service()***, sino uno de los métodos más especializados (normalmente ***doPost()***), que tienen los mismos argumentos que ***service()***: dos objetos referenciados por las interfaces ***ServletRequest*** y ***ServletResponse***.

En la Figura 3 aparecen también algunas otras ***interfaces***, cuyo papel se resume a continuación.

1. La interface ***ServletContext*** permite a los ***servlets*** acceder a información sobre el entorno en que se están ejecutando.
2. La interface ***ServletConfig*** define métodos que permiten pasar al ***servlet*** información sobre sus parámetros de inicialización.
3. La interface ***ServletRequest*** permite al método ***service()*** de ***GenericServlet*** obtener información sobre una petición de servicio recibida de un cliente. Algunos de los datos proporcionados por ***GenericServlet*** son los nombres y valores de los parámetros enviados por el formulario HTML y una ***input stream***.
4. La interface ***ServletResponse*** permite al método ***service()*** de ***GenericServlet*** enviar su respuesta al cliente que ha solicitado el servicio. Esta interface dispone de métodos para

obtener un ***output stream*** o un ***writer*** con los que enviar al cliente datos binarios o caracteres, respectivamente.

5. La interface ***HttpServletRequest*** deriva de ***ServletRequest***. Esta interface permite a los métodos ***service()***, ***doPost()***, ***doGet()***, etc. de la clase ***HttpServlet*** recibir una petición de servicio ***HTTP***. Esta interface permite obtener información del header de la petición de servicio ***HTTP***.
6. La interface ***HttpServletResponse*** extiende ***ServletResponse***. A través de esta interface los métodos de ***HttpServlet*** envían información a los clientes que les han pedido algún servicio.

El ***API*** del ***JSDK 2.0*** dispone de clases e interfaces adicionales, no citadas en este apartado. Algunas de estas clases e interfaces serán consideradas en apartados posteriores.

4.2 LA APLICACIÓN SERVLETRUNNER

Servletrunner es la utilidad que proporciona ***Sun*** conjuntamente con el ***JSDK***. Es a los ***servlets*** lo que el ***appletviewer*** a los ***applets***. Sin embargo, es mucho más útil que ***appletviewer***, porque mientras es muy fácil disponer de un ***browser*** en el que comprobar las ***applets***, no es tan sencillo instalar y disponer de un ***servidor HTTP*** en el que comprobar los ***servlets***. Por esta razón la aplicación ***servletrunner***, a pesar de ser bastante básica y poco configurable, es una herramienta muy útil para el desarrollo de ***servlets***, pues se ejecuta desde la línea de comandos del ***MS-DOS***. Como es natural, una vez que se haya probado debidamente el funcionamiento de los ***servlets***, para una aplicación en una empresa real sería preciso emplear ***servidores HTTP*** profesionales.

Además, ***servletrunner*** es ***multithread***, lo que le permite gestionar múltiples peticiones a la vez. Gracias a ello es posible ejecutar distintos ***servlets*** simultáneamente o probar ***servlets*** que llaman a su vez a otros ***servlets***.

Una advertencia: ***servletrunner*** no carga de nuevo de modo automático los ***servlets*** que hayan sido actualizados externamente; es decir, si se cambia algo en el código de un ***servlet*** y se vuelve a compilar, al hacer una nueva llamada al mismo ***servletrunner*** utiliza la copia de la anterior versión del ***servlet*** que tiene cargada. Para que cargue la nueva es necesario cerrar el ***servletrunner*** (***Ctrl+C***) y reiniciarlo otra vez. Esta operación habrá que realizarla cada vez que se modifique el ***servlet***.

Para asegurarse de que ***servletrunner*** tiene acceso a los packages del ***Servlet API***, será necesario comprobar que la variable de entorno ***CLASSPATH*** contiene la ruta de acceso del fichero ***jsdk.jar*** en el directorio ***lib*** (en el ***Servlet API 2.1*** están situados en el fichero ***servlet-2.1.0.jar***). En la plataforma ***Java 2*** es más sencillo simplemente copiar el ***JAR*** al directorio ***ext*** que se encuentra en ***\jre\lib***. Esto hace que los ***packages*** sean tratados como extensiones estándar de ***Java***. También es necesario cambiar la variable ***PATH*** para que se encuentre la aplicación ***servletrunner.exe***. Otra posibilidad es copiar esta aplicación al directorio donde están los demás ejecutables de ***Java*** (por ejemplo ***c:\jdk117\bin***).

4.3 FICHEROS DE PROPIEDADES

Servletrunner permite la utilización de ficheros que contienen las propiedades (***properties***) utilizadas en la configuración, creación e inicialización de los ***servlets***. Las propiedades son pares del tipo ***clave/valor***. Por ejemplo, `servlet.catalogo.codigo=ServletCatalogo` es una propiedad cuya "clave" es `servlet.catalogo.codigo` y cuyo "valor" es `ServletCatalogo`.

Existen ***dos propiedades*** muy importantes para los ***servlets***:

1. `servlet.`*`nombre`*`.code`
2. `servlet.`*`nombre`*`.initargs`

La propiedad ***servlet.nombre.code*** debe contener el nombre completo de la clase del ***servlet***, incluyendo su ***package***. Por ejemplo, la propiedad,

```
servlet.libros.code=basededatos.ServletLibros
```

asocia el nombre ***libros*** con la clase ***basededatos.ServletLibros***.

La propiedad ***initargs*** contiene los parámetros de inicialización del ***servlet***. El valor de un único parámetro se establece en la forma ***`nombreDeParametro=valorDeParametro`***. Es posible establecer el valor de varios parámetros a la vez, pero el conjunto de la propiedad debe ser una única línea lógica. Por tanto, para una mayor legibilidad será preciso emplear el carácter ***barra invertida*** (****) para emplear varias líneas del fichero. Así, por ejemplo:

```
servlet.librodb.initArgs=\
   fichero=servlets/Datos,\
   usuario=administrador,\
...
```

Obsérvese que los distintos parámetros se encuentran separados por ***comas*** (,). El último de los parámetros no necesitará ninguna coma al final.

Todas estas propiedades estarán almacenadas en un fichero que por defecto tiene el nombre ***servlet.properties*** (se puede especificar otro nombre en la línea de comandos de ***servletrunner*** tal y como se verá más adelante). Se pueden incluir ***líneas de comentario***, que deberán comenzar por el carácter (**#**). Por defecto, este fichero debe estar en el mismo directorio que el ***servlet***, pero al ejecutar ***servletrunner*** puede especificarse un nombre de fichero de propiedades con un ***path*** diferente.

4.4 EJECUCIÓN DE LA APLICACIÓN SERVLETRUNNER

La aplicación ***servletrunner*** se ejecuta desde la línea de comandos de ***MS-DOS*** y admite los siguientes parámetros (aparecen tecleando en la consola "***servletrunner ?***"):

```
-p  puerto al que escuchar
-m  número máximo de conexiones
-t  tiempo de desconexión en milisegundos
-d  directorio en el que están los servlets
-s  nombre del fichero de propiedades
```

Así por ejemplo, si se tuviera un ***servlet*** en el directorio ***c:\programas***, el fichero de propiedades se llamara ***ServletEjemplo.prop*** y se quisiera que el ***servletrunner*** estuviera escuchando el ***puerto*** 8000, habría que escribir lo siguiente en la línea de comandos:

```
C:\servletrunner -p 8000 -d c:\programas -s ServletEjemplo.prop
```

5 EJEMPLO INTRODUCTORIO

Para poder hacerse una idea del funcionamiento de un ***servlet*** y del aspecto que tienen los mismos, lo mejor es estudiar un ejemplo sencillo. Imagínese que en una página web se desea recabar la opinión de un visitante así como algunos de sus datos personales, con el fin de realizar un estudio estadístico. Dicha información podría ser almacenada en una base de datos para su posterior estudio.

La primera tarea sería diseñar un formulario en el que el visitante pudiera introducir los datos. Este paso es idéntico a lo que se haría al escribir un ***programa CGI***, ya que bastará con utilizar los

tags que proporciona el lenguaje ***HTML*** (**<FORM>, <ACTION>, <TYPE>**, etc.). Para una mayor información acerca del mismo se recomienda la lectura de la información complementaria que está en la web de ***Informática 2.*** En adelante se supondrá que el lector se encuentra suficientemente familiarizado con dicho lenguaje.

5.1 INSTALACIÓN DEL JAVA SERVLET DEVELOPMENT KIT (JSDK 2.0)

Para poder ejecutar este ejemplo es necesario que el ***JSDK 2.0*** esté correctamente instalado, bien en el propio ordenador, bien en uno de los ordenadores de las Salas de PCs de la ESIISS. Para realizar esta instalación en un ordenador propio se pueden seguir los siguientes pasos:

1. En primer lugar se debe conseguir el fichero de instalación, llamado ***jsdk20-win32.exe***. Este fichero se puede obtener de ***Sun*** (*http://www.javasoft.com/products/servlet/index.html*) o en el directorio ***Q:\Infor2\Servlet*** en la red de PCs de la ESIISS. Se trata de un fichero de 950 Kbytes, que puede ser transportado en un disquete sin dificultad.
2. Se copia el fichero citado al directorio ***C:\Temp*** del propio ordenador. Se clica dos veces sobre dicho fichero y comienza el proceso de instalación.
3. Se determina el directorio en el que se realizará la instalación. El programa de instalación propone el directorio ***C:\Jsdk2.0,*** que es perfectamente adecuado.
4. En el directorio ***C:\Jsdk2.0\bin*** aparece la aplicación ***servletrunner.exe***, que es muy importante como se ha visto anteriormente. Para que esta aplicación sea encontrada al teclear su nombre en la ventana de ***MS-DOS*** es necesario que el nombre de dicho directorio aparezca en la variable de entorno ***PATH***. Una posibilidad es modificar de modo acorde dicha variable y otra copiar el fichero ***servletrunner.exe*** al directorio donde están los demás ejecutables de Java (por ejemplo ***C:\Jdk1.1.7\bin***); como ese directorio ya está en el ***PATH***, la aplicación ***servletrunner.exe*** será encontrada sin dificultad. Ésta es la solución más sencilla.
5. Además de encontrar ***servletrunner.exe***, tanto para compilar los servlets como para ejecutarlos con ***servletrunner*** es necesario encontrar las clases e interfaces del ***API*** de ***JSDK 2.0***. Estas clases pueden estar por ejemplo en el archivo ***C:\Jsdk2.0\lib\jsdk.jar***. Para que este archivo pueda ser localizado, es necesario modificar la variable de entorno ***CLASSPATH***. Esto se puede hacer en la forma (antepone el path del fichero ***jsdk.jar*** al valor anterior que se tuviera en la variable ***CLASSPATH***):

   ```
   set CLASSPATH=C:\Jsdk2.0\lib\jsdk.jar;%CLASSPATH%
   ```

6. En las ***Salas de PCs*** de la Escuela se puede compilar los servlets con ***Visual J++ 6.0***, aunque no ejecutarlos ni utilizar el debugger. Para ello, después de crear el proyecto, se debe ir a ***Project/Properties/Classpath/New***, e introducir ***Q:\Infor2\Servlet\jsdk.jar***.

En las Salas de PCs de la Escuela el ***JSDK 2.0*** está instalado en el directorio ***Q:\jdk20***. La aplicación ***servletrunner*** ha sido ya copiada a los directorio ***Q:\jdk117\bin*** y ***Q:\jdk12\bin***, con lo cual está ya accesible. Para modificar el ***CLASSPATH*** se sugiere utilizar los nuevos ficheros ***jdk117s.bat*** y ***jdk12s.bat***, disponibles en el directorio ***Q:\Infor2*** y en la web de ***Informática 2*** (apartado de ***Clases y Prácticas***, semana 9).

5.2 FORMULARIO

El formulario contendrá dos campos de tipo TEXT donde el visitante introducirá su ***nombre*** y ***apellidos***. A continuación, deberá indicar la opinión que le merece la página visitada eligiendo una

entre tres posibles (***Buena***, ***Regular*** y ***Mala***),. Por último, se ofrece al usuario la posibilidad de escribir un ***comentario*** si así lo considera oportuno. En la Figura 4 puede observarse el diseño del formulario creado.

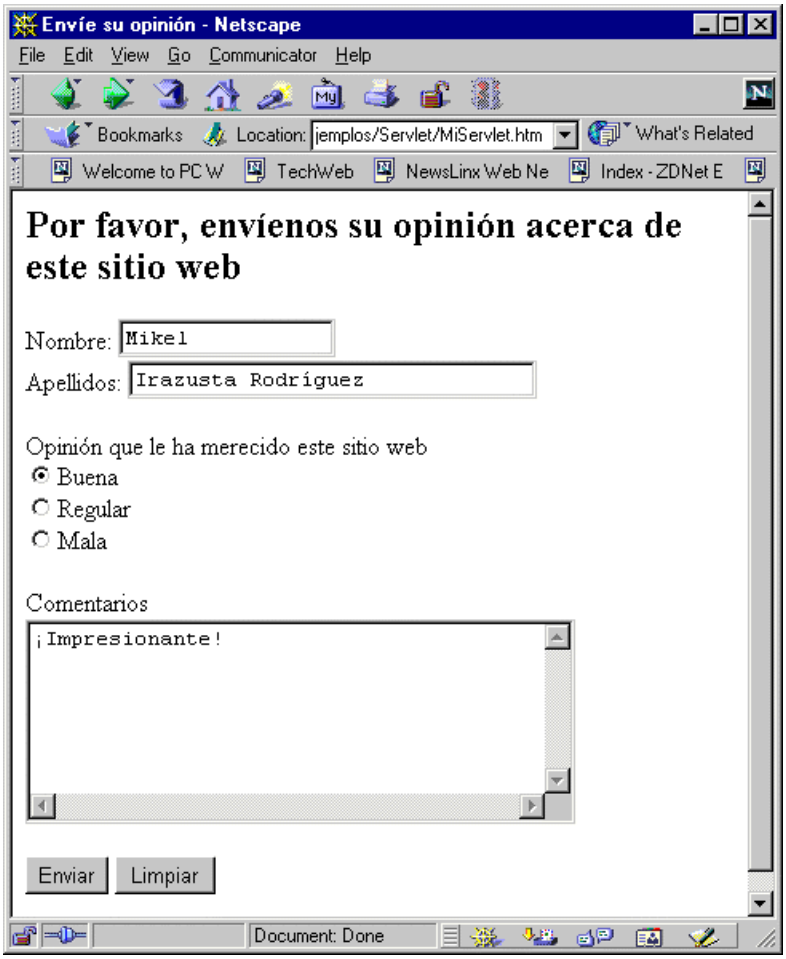

Figura 4. Diseño del formulario de adquisición de datos.

El código correspondiente a la ***página HTML*** que contiene este formulario es el siguiente (fichero ***MiServlet.htm***):

```
<HTML>
<HEAD>
    <TITLE>Envíe su opinión</TITLE>
</HEAD>

<BODY>

<H2>Por favor, envíenos su opinión acerca de este sitio web</H2>

<FORM ACTION="http://jgjalon.ceit.es:8080/servlet/ServletOpinion" METHOD="POST">

    Nombre: <INPUT TYPE="TEXT" NAME="nombre" SIZE=15><BR>
    Apellidos: <INPUT TYPE="TEXT" NAME="apellidos" SIZE=30><P>

    Opinión que le ha merecido este sitio web<BR>
    <INPUT TYPE="RADIO" CHECKED NAME="opinion" VALUE="Buena">Buena<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Regular">Regular<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Mala">Mala<P>

    Comentarios <BR>
    <TEXTAREA NAME="comentarios" ROWS=6 COLS=40>
    </TEXTAREA><P>
```

```
    <INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
    <INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">

</FORM>

</BODY>
</HTML>
```

En el código anterior, hay algunas cosas que merecen ser comentadas. En primer lugar, es necesario asignar un identificador único (es decir, un valor de la propiedad ***NAME***) a cada uno de los ***campos*** del formulario, ya que la información que reciba el ***servlet*** estará organizada en forma de ***pares de valores***, donde uno de los elementos de dicho par será un ***String*** que contendrá el ***nombre del campo***. Así, por ejemplo, si se introdujera como nombre del visitante *"Mikel"*, el ***servlet*** recibiría del browser el par ***nombre=Mikel***, que permitirá acceder de una forma sencilla al nombre introducido mediante el método ***getParameter()***, tal y como se explicará posteriormente al analizar el ***servlet*** del ejemplo introductorio. Por este motivo es importante no utilizar nombres duplicados en los elementos de los formularios.

Por otra parte puede observarse que en el *tag* <**FORM>** se han utilizado dos propiedades, ***ACTION*** y ***METHOD***. El método (***METHOD***) utilizado para la transmisión de datos es el método ***HTTP POST***. También se podría haber utilizado el método ***HTTP GET***, pero este método tiene algunas limitaciones en cuanto al volumen de datos transmisible, por lo que es recomendable la utilizar el método ***POST***. Mediante la propiedad ***ACTION*** deberá especificarse el ***URL*** del ***servlet*** que debe procesar los datos. Este ***URL*** contiene, en el ejemplo presentado, las siguientes características:

- El ***servlet*** se encuentra situado en un servidor cuyo nombre es ***miServidor*** (un ejemplo más real podría ser ***www1.ceit.es***). Este nombre dependerá del ordenador que proporcione los servicios de red. La forma de saber cuál es el nombre de dicho ordenador y su dirección ***IP*** es acudiendo al ***Panel de Control (Control Panel)*** de ***Windows***. Clicando en ***Red (Network)***, se accede a las propiedades de la red (obviamente, si se quiere utilizar ***servlets***, será necesario tener instalados los drivers de red, en concreto los drivers de ***TCP/IP*** que vienen con ***Windows 95/98/NT***). Dentro de la lengüeta ***Protocolos (Protocols)*** se escoge ***TCP/IP*** y se clica en ***Propiedades (Properties)***. Aparecerá un nuevo cuadro de diálogo en el que habrá que seleccionar la lengüeta ***DNS (Domain Name System).*** Allí se encuentra recogido el nombre del ordenador ***(Host Name)*** así como su dominio ***(Domain Name).*** En cualquier caso, para poder hacer pruebas, se puede utilizar el como nombre de servidor el *host local* o ***localhost***, cuyo número ***IP*** es ***127.0.0.1***. Por ejemplo, se podría haber escrito (aunque estos servicios no serían accesibles desde el exterior de la ESIISS):

  ```
  <FORM ACTION="http://localhost:8080/servlet/ServletOpinion" METHOD="POST">
  ```

  o de otra forma,

  ```
  <FORM ACTION="http://127.0.0.1:8080/servlet/ServletOpinion" METHOD="POST">
  ```

- El ***servidor HTTP*** está "escuchando" por el puerto el ***puerto*** 8080. Todas las llamadas utilizando dicho puerto serán procesadas por el módulo del servidor encargado de la gestión de los ***servlets***. En principio es factible la utilización de cualquier puerto libre del sistema, siempre que se indique al ***servidor HTTP*** cuál va a ser el puerto utilizado para dichas llamadas. Por diversos motivos, esto último debe ser configurado por el administrador del sistema.

- El ***servlet*** se encuentra situado en un subdirectorio (virtual) del servidor llamado ***servlet***. Este nombre es en principio opcional, aunque la mayoría de los servidores lo utilizan por defecto.

En caso de querer utilizar otro directorio, el servidor debe ser configurado por el administrador a tal fin. En concreto, la aplicación ***servletrunner*** de ***Sun*** no permite dicha modificación, por lo que el ***URL*** utilizada debe contener dicho nombre de directorio.

- El nombre del ***servlet*** empleado es ***ServletOpinion***, y es éste el que recibirá la información enviada por el cliente al servidor (el formulario en este caso), y quien se encargará de diseñar la respuesta, que pasará al servidor para que este a su vez la envíe de vuelta al cliente. Al final se ejecutará una clase llamada ***ServletOpinion.class***.

5.3 CÓDIGO DEL SERVLET

Tal y como se ha mencionado con anterioridad, el ***servlet*** que gestionará toda la información del formulario se llamará ***ServletOpinion***. Como un ***servlet*** es una clase de ***Java***, deberá por tanto encontrarse almacenado en un fichero con el nombre ***ServletOpinion.java***. En cualquier caso, por hacer lo más simple posible este ejemplo introductorio, este ***servlet*** se limitará a responder al usuario con una página ***HTML*** con la información introducida en el formulario, dejando para un posterior apartado el estudio de cómo se almacenarían dichos datos. El código fuente de la clase ***ServletOpinion*** es el siguiente:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletOpinion extends HttpServlet {

    // Declaración de variables miembro correspondientes a
    //    los campos del formulario
    private String nombre=null;
    private String apellidos=null;
    private String opinion=null;
    private String comentarios=null;


    // Este método se ejecuta una única vez (al ser inicializado el servlet)
    // Se suelen inicializar variables y realizar operaciones costosas en
    //     tiempo de ejecución (abrir ficheros, bases de datos, etc)
    public void init(ServletConfig config) throws ServletException {
        // Llamada al método init() de la superclase (GenericServlet)
        // Así se asegura una correcta inicialización del servlet
        super.init(config);
        System.out.println("Iniciando ServletOpinion...");
    } // fin del método init()


    // Este método es llamado por el servidor web al "apagarse" (al hacer
    //     shutdown). Sirve para proporcionar una correcta desconexión de una
    //     base de datos, cerrar ficheros abiertos, etc.
    public void destroy() {
        System.out.println("No hay nada que hacer...");
    } fin del método destroy()


    // Método llamado mediante un HTTP POST. Este método se llama
    //     automáticamente al ejecutar un formulario HTML
    public void doPost (HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {

        // Adquisición de los valores del formulario a través del objeto req
        nombre=req.getParameter("nombre");
        apellidos=req.getParameter("apellidos");
        opinion=req.getParameter("opinion");
        comentarios=req.getParameter("comentarios");

        // Devolver al usuario una página HTML con los valores adquiridos
```

```
            devolverPaginaHTML(resp);
        } // fin del método doPost()

        public void devolverPaginaHTML(HttpServletResponse resp) {

            // En primer lugar se establece el tipo de contenido MIME de la respuesta
            resp.setContentType("text/html");

            // Se obtiene un PrintWriter donde escribir (sólo para mandar texto)
            PrintWriter out = null;
            try {
                out=resp.getWriter();
            } catch (IOException io) {
                System.out.println("Se ha producido una excepcion");
            }

            // Se genera el contenido de la página HTML
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Valores recogidos en el formulario</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<b><font size=+2>Valores recogidos del ");
            out.println("formulario: </font></b>");
            out.println("<p><font size=+1><b>Nombre: </b>"+nombre+"</font>");
            out.println("<br><fontsize=+1><b>Apellido: </b>"
                        +apellidos+"</font><b><font size=+1></font></b>");
            out.println("<p><font size=+1> <b>Opini&oacute;n: </b><i>" + opinion +
                        "</i></font>");
            out.println("<br><font size=+1><b>Comentarios: </b>" + comentarios
                        +"</font>");
            out.println("</body>");
            out.println("</html>");

            // Se fuerza la descarga del buffer y se cierra el PrintWriter,
            //     liberando recursos de esta forma. IMPORTANTE
            out.flush();
            out.close();
        } // fin de devolverPaginaHTML()


        // Función que permite al servidor web obtener una pequeña descripción del
        // servlet, qué cometido tiene, nombre del autor, comentarios
        //     adicionales, etc.
        public String getServletInfo() {
            return "Este servlet lee los datos de un formulario" +
                   " y los muestra en pantalla";
        } // fin del método getServletInfo()
    }
```

El resultado obtenido en el browser tras la ejecución del ***servlet*** puede apreciarse en la Figura 5 mostrada a continuación:

En aras de una mayor simplicidad en esta primera aproximación a los ***servlets***, se ha evitado tratar de conseguir un código más sólido, que debería realizar las comprobaciones pertinentes (verificar que los ***String*** no son ***null*** después de leer el parámetro, excepciones que se pudieran dar, etc.) e informar al usuario acerca de posibles errores en caso de que fuera necesario.

En cualquier caso, puede observarse que el aspecto del código del ***servlet*** es muy similar al de cualquier otra clase de ***Java***. Sin embargo, cabe destacar algunos aspectos particulares:

- La clase ***ServletOpinion*** hereda de la clase ***HttpServlet***, que a su vez hereda de ***GenericServlet***. La forma más sencilla (y por tanto la que debería ser siempre empleada) de crear un ***servlet***, es heredar de la clase ***HttpServlet***. De esta forma se está identificando la clase como un ***servlet*** que se conectará con un ***servidor HTTP***. Más adelante se estudiará esto con más detalle.

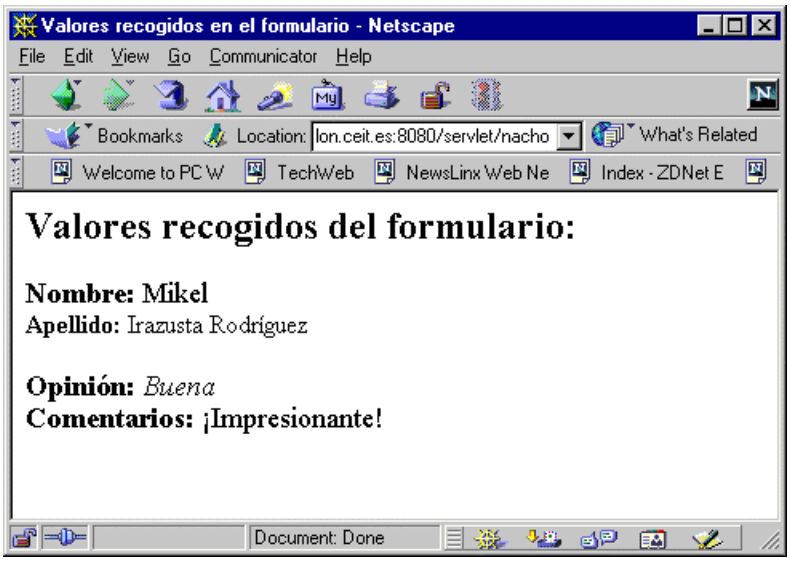

Figura 5. Página HTML devuelta por el servlet.

- El método ***init()*** es el primero en ser ejecutado. Sólo es ejecutado ***la primera vez*** que el ***servlet*** es llamado. Se llama al método ***init()*** de la super-clase ***GenericServlet*** a fin de que la inicialización sea completa y correcta. La interface ***ServletConfig*** proporciona la información que necesita el ***servlet*** para inicializarse (parámetros de inicialización, etc.).
- El método ***destroy()*** no tiene ninguna función en este ***servlet***, ya que no se ha utilizado ningún recurso adicional que necesite ser cerrado, pero tiene mucha importancia si lo que se busca es proporcionar una descarga correcta del ***servlet*** de la memoria, de forma que no queden recursos ocupados indebidamente, o haya conflictos entre recursos en uso. Tareas propias de este método son por ejemplo el cierre de las conexiones con otros ordenadores o con bases de datos.
- Como el formulario ***HTML*** utiliza el método ***HTTP POST*** para la transmisión de sus datos, habrá que redefinir el método ***doPost()***, que se encarga de procesar la respuesta y que tiene como argumentos el objeto que contiene la petición y el que contiene la respuesta (pertenecientes a las clases ***HttpServletRequest*** y ***HttpServletResponse***, respectivamente). Este método será llamado tras la inicialización del ***servlet*** (en caso de que no haya sido previamente inicializado), y contendrá el núcleo del código del ***servlet*** (llamadas a otros ***servlets***, llamadas a otros métodos, etc.).
- El método ***getServletInfo()*** proporciona datos acerca del ***servlet*** (autor, fecha de creación, funcionamiento, etc.) al servidor web. No es en ningún caso obligatoria su utilización aunque puede ser interesante cuando se tienen muchos ***servlets*** funcionando en un mismo servidor y puede resultar compleja la identificación de los mismos.
- Por último, el método ***devolverPaginaHTML()*** es el encargado de mandar los valores recogidos del cliente. En primer lugar es necesario tener un *stream* hacia el cliente (***PrintWriter*** cuando haya que mandar texto, ***ServletOutputStream*** para datos binarios). Posteriormente debe indicarse el tipo de contenido **MIME** de aquello que va dirigido al cliente (***text/html*** en el caso presentado). Estos dos pasos son necesarios para poder enviar correctamente los datos al cliente. Finalmente, mediante el método ***println()*** se va generando la página **HTML** propiamente dicha (en forma de ***String***).
- Puede sorprender la forma en que ha sido enviada la página ***HTML*** como ***String***. Podría parecer más lógico generar un ***String*** en la forma (concatenación de ***Strings***),

```
String texto=”<html><head> + ... +"<b>Nombre:</b>" + nombre + "</font>...
```

y después escribirlo en el *stream* o flujo de salida. Sin embargo, uno de los parámetros a tener en cuenta en los ***servlets*** es el tiempo de respuesta, que tiene que ser el mínimo posible. En este sentido, la creación de un ***String*** mediante concatenación es bastante costosa, pues cada vez que se concatenan dos ***Strings*** mediante el signo + se están convirtiendo a ***StringBuffers*** y a su vez creando un nuevo ***String***, lo que utilizado profusamente da lugar requiere más recursos que lo que se ha hecho en el ejemplo, donde se han escrito directamente mediante el método ***println()***.

El esquema mencionado en este ejemplo se repite en la mayoría de los ***servlets*** y es el fundamento de este tecnología. En posteriores apartados se efectuará un estudio más detallado de las clases y métodos empleados en este pequeño ejemplo.

6 EL SERVLET API 2.0

El ***Java Servlet API 2.0*** es una extensión al ***API*** de ***Java 1.1.x***, y también de ***Java 2***. Contiene los packages ***javax.servlet*** y ***javax.servlet.http***. El ***API*** proporciona soporte en cuatro áreas:

1. Control del ciclo de vida de un ***servlet***: clase ***GenericServlet***
2. Acceso al contexto del ***servlet*** (***servlet context***)
3. Clases de utilidades
4. Clases de soporte específicas para ***HTTP***: clase ***HttpServlet***

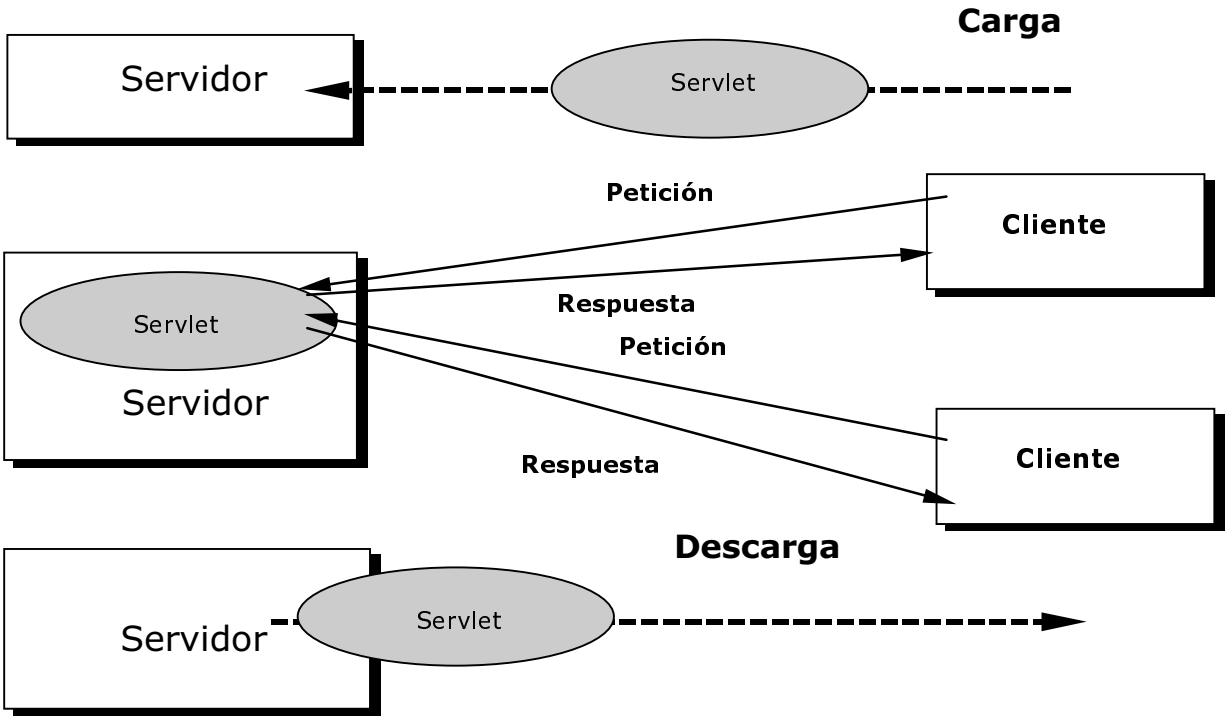

Figura 6. Ciclo de vida de un servlet.

6.1 EL CICLO DE VIDA DE UN SERVLET: CLASE GENERICSERVLET

Como se ha dicho en el Apartado 4.1, página 15, la clase ***GenericServlet*** es una clase abstract porque declara el método ***service()*** como ***abstract***. Aunque los ***servlets*** desarrollados en conexión con páginas web suelen derivar de la clase ***HttpServlet***, puede ser útil estudiar el ciclo de vida de un

servlet en relación con los métodos de la clase ***GenericServlet***. Esto es lo que se hará en los apartados siguientes. Puede ser útil recordar la Figura 3, en la página 15, que ofrece una visión general de las principales clases e interfaces en relación con los ***servlets***. Además, la clase ***HttpServlet*** hereda los métodos de ***GenericServlet*** y define el método ***service()***.

Los ***servlets*** se ejecutan en el ***servidor HTTP*** como parte integrante del propio proceso del servidor. Por este motivo, el ***servidor HTTP*** es el responsable de la inicialización, llamada y destrucción de cada objeto de un ***servlet***, tal y como puede observarse en la Figura 6.

Un ***servidor web*** se comunica con un ***servlet*** mediante la los métodos de la interface ***javax.servlet.Servlet.*** Esta interface está constituida básicamente por tres métodos principales, alguno de los cuales ya se ha utilizado en el ejemplo introductorio:

- ***init(), destroy()*** y ***service()***

y por dos métodos algo menos importantes:

- ***getServletConfig()***, ***getServletInfo()***

6.1.1 El método init() en la clase GenericServlet

Cuando un ***servlet*** es cargado por primera vez, el método ***init()*** es llamado por el ***servidor HTTP***. Este método no será llamado nunca más mientras el ***servlet*** se esté ejecutando. Esto permite al ***servlet*** efectuar cualquier operación de inicialización potencialmente costosa en términos de CPU, ya que ésta sólo se ejecutará la primera vez. Esto es una ventaja importante frente a los ***programas CGI***, que son cargados en memoria cada vez que hay una petición por parte del cliente. Por ejemplo, si en un día hay 500 consultas a una base de datos, mediante un ***CGI*** habría que abrir una conexión con la base de datos 500 veces, frente a una única apertura que sería necesaria con un ***servlet***, pues dicha conexión podría quedar abierta a la espera de recibir nuevas peticiones.

Si el servidor permite pre-cargar los ***servlets***, el método ***init()*** será llamado al iniciarse el servidor. Si el servidor no tiene esa posibilidad, será llamado la primera vez que haya una petición por parte de un cliente.

El método ***init()*** tiene un único argumento, que es una referencia a un objeto de la interface ***ServletConfig***, que proporciona los argumentos de inicialización del ***servlet***. Este objeto dispone del método ***getServletContext()*** que devuelve una referencia de la interface ***ServletContext***, que a su vez contiene información acerca del entorno en el que se está ejecutando el ***servlet***.

Siempre que se redefina el método ***init()*** de la clase base ***GenericServlet*** (o de ***HttpServlet***, que lo hereda de ***GenericServlet***), será preciso llamar al método ***init()*** de la super-clase, a fin de garantizar que la inicialización se efectúe correctamente. Por ejemplo:

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

    // Llamada al método init de la superclase
    super.init(config);
    System.out.println("Iniciando...");

    // Definición de variables
    ...
    // Apertura de conexiones, ficheros, etc.
    ...
}
...
```

El servidor garantiza que el método ***init()*** termina su ejecución antes de que sea llamado cualquier otro método del ***servlet***.

Métodos de ServletRequest	**Comentarios**
public abstract int **getContentLength()**	Devuelve el tamaño de la petición del cliente o -1 si es desconocido.
public abstract String **getContentType()**	Devuelve el tipo de contenido **MIME** de la petición o **null** si éste es desconocido.
public abstract String **getProtocol()**	Devuelve el protocolo y la versión de la petición como un ***String*** en la forma <protocolo>/<versión mayor>.<versión menor>
public abstract String **getScheme()**	Devuelve el tipo de esquema de la **URL** de la petición: http, https, ftp...
public abstract String **getServerName()**	Devuelve el nombre del host del servidor que recibió la petición..
public abstract int **getServerPort()**	Devuelve el número del puerto en el que fue recibida la petición.
public abstract String **getRemoteAddr()**	Devuelve la dirección IP del ordenador que realizó la petición.
public abstract String **getRemoteHost()**	Devuelve el nombre completo del ordenador que realizó la petición.
public abstract ServletInputStream **getInputStream()** throws IOException	Devuelve un ***InputStream*** para leer los datos binarios que vienen dentro del cuerpo de la petición.
public abstract String **getParameter(String)**	Devuelve un ***String*** que contiene el valor del parámetro especificado, o null si dicho parámetro no existe. Sólo debe emplearse cuando se está seguro de que el parámetro tiene un único valor.
public abstract String[] **getParameterValues(String)**	Devuelve los valores del parámetro especificado en forma de un array de ***Strings***, o null si el parámetro no existe. Útil cuando un parámetro puede tener más de un valor.
public abstract Enumeration **getParameterNames()**	Devuelve una enumeración en forma de ***String*** de los parámetros encapsulados en la petición. No devuelve nada si el ***InputStream*** está vacío.
public abstract BufferedReader **getReader()** throws IOException	Devuelve un ***BufferedReader*** que permite leer el texto contenido en el cuerpo de la petición.
public abstract String **getCharacterEncoding()**	Devuelve el tipo de codificación de los caracteres empleados en la petición.

Tabla 1. Métodos de la interface ***ServletRequest***.

6.1.2 El método service() en la clase GenericServlet

Este método es el núcleo fundamental del ***servlet***. Recuérdese que es ***abstract*** en ***GenericServlet***, por lo que si el ***servlet*** deriva de esta clase deberá ser definido por el programador. Cada petición por parte del cliente se traduce en una llamada al método ***service()*** del ***servlet***. El método ***service()*** lee la petición y debe producir una respuesta en base a los dos argumentos que recibe:

- Un objeto de la interface ***ServletRequest*** con datos enviados por el cliente. Estos incluyen parejas de parámetros clave/valor y un ***InputStream***. Hay diversos métodos que proporcionan información acerca del cliente y de la petición efectuado por el mismo, entre otros los mostrados en la Tabla 1.
- Un objeto de la interface ***ServletResponse***, que encapsula la respuesta del ***servlet*** al cliente. En el proceso de preparación de la respuesta, es necesario llamar al método ***setContentType()***, a fin de establecer el tipo de contenido **MIME** de la respuesta. La Tabla 2 indica los métodos de la interface ***ServletResponse***.

Puede observarse en la Tabla 1 que hay dos formas de recibir la información de un formulario HTML en un ***servlet***. La primera de ellas consiste en obtener los valores de los parámetros (métodos ***getParameterNames()*** y ***getParameterValues()***) y la segunda en recibir la información mediante un ***InputStream*** o un ***Reader*** y hacer por uno mismo su partición decodificación.

Métodos de ServletResponse	**Comentarios**
ServletOutput Stream getOutputStream()	Permite obtener un ServletOutputStream para enviar datos binarios
PrintWriter getWriter()	Permite obtener un PrintWriter para enviar caracteres
setContentType(String)	Establece el tipo MIME de la salida
setContentLength(int)	Establece el tamaño de la respuesta

Tabla 2. Métodos de la interface ***ServletResponse***.

El cometido del método ***service()*** es conceptualmente bastante simple: genera una respuesta por cada petición recibida de un cliente. Es importante tener en cuenta que puede haber múltiples respuestas que están siendo procesadas al mismo tiempo, pues los ***servlets*** son ***multithread***. Esto hace que haya que ser especialmente cuidadoso con los ***threads***, para evitar por ejemplo que haya dos objetos de un ***servlet*** escribiendo simultáneamente en un mismo campo de una base de datos.

A pesar de la importancia del método ***service()***, en general es no es aconsejable su definición (no queda más remedio que hacerlo si la clase del servlet deriva de ***GenericServlet***, pero lo lógico es que el programador derive las clases de sus servlets de ***HttpServlet***). El motivo es simple: la clase ***HttpServlet*** define ***service()*** de una forma más que adecuada, llamando a otros métodos (***doPost()***, ***doGet()***, etc.) que son los que tiene que redefinir el programador. La forma de esta redefinición será estudiada en apartados posteriores.

6.1.3 El método destroy() en la clase GenericServlet: forma de terminar ordenadamente

Una buena implementación de este método debe permitir que el ***servlet*** concluya sus tareas de forma ordenada. De esta forma, es posible liberar recursos (ficheros abiertos, conexiones con bases de datos, etc.) de una forma limpia y segura. Cuando esto no es necesario o importante, no hará falta redefinir el método ***destroy()***.

Puede suceder que al llamar al método ***destroy()*** haya peticiones de servicio que estén todavía siendo ejecutadas por el método ***service()***, lo que podría provocar un fallo general del sistema. Por este motivo, es conveniente escribir el método ***destroy()*** de forma que se retrase la liberación de recursos hasta que no hayan concluido todas las llamadas al método ***service()***. A continuación se presenta una forma de lograr una correcta descarga del ***servlet***:

En primer lugar, es preciso saber si existe alguna llamada al método ***service()*** pendiente de ejecución, para lo cual se debe llevar un ***contador*** con las llamadas activas a dicho método.

Aunque en general es poco recomendable redefinir ***service()*** (en caso de tratarse de un ***servlet*** que derive de ***HttpServlet)***. Sin embargo, en este caso sí resulta conveniente su redefinición, para poder saber cuándo ha sido llamado. El método redefinido deberá llamar al método ***service()*** de su super-clase para mantener íntegra la funcionalidad del ***servlet***.

Los métodos de actualización del ***contador*** deben estar ***sincronizados***, para evitar que dicho valor sea accedido simultáneamente por dos o más ***threads***, lo que podría hacer que su valor fuera erróneo.

Además, no basta con que el ***servlet*** espere a que todos los métodos ***service()*** hayan acabado. Es preciso indicarle a dicho método que el servidor se dispone a apagarse. De otra forma, el ***servlet*** podría quedar esperando indefinidamente a que los métodos ***service()*** acabaran. Esto se consigue utilizando una variable ***boolean*** que establezca esta condición.

Todas las consideraciones anteriores se han introducido en el siguiente código:

```
public class ServletSeguro extends HttpServlet {
...
    private int contador=0;
    private boolean apagandose=false;

...
    protected synchronized void entrandoEnService() {
        contador++;
    }

    protected synchronized void saliendoDeService() {
        contador--;
    }

    protected synchronized void numeroDeServicios() {
        return contador;
    }

    protected void service(HttpServletRequest req, HttpServletResponse resp)
                          throws ServletException, IOException {

        entrandoEnService();
        try {
            super.service(req, resp);
        } finally {
            saliendoDeService();
        }

    } // fin del método service()

    protected void setApagandose(boolean flag){
        apagandose=flag;
    }

    protected boolean estaApagandose() {
        return apagandose;
    }

...

    public void destroy() {

        // Comprobar que hay servicios en ejecución y en caso afirmativo
        // ordernarles que paren la ejecución
        if(numeroDeServicios()>0)
            setApagandose(true);

        // Mientras haya servicios en ejecución, esperar
        while(numServices()>0) {
            try {
                Thread.sleep(intervalo);
            } catch(InterruptedException e) {
            } // fin del catch
        } // fin del while
    } // fin de destroy()

    ...
    // Servicio
    public void doPost(...) {
        ...
        // Comprobación de que el servidor no se está apagando
        for (i=0; ((i<numeroDeCosasAHacer)&& !estaApagandose()); i++) {
            try {
                ...
                // Aquí viene el código
            } catch(Exception e)
        } // fin del for
    } // fin de doPost()

} // fin de la clase ServletEjemplo
```

6.2 EL CONTEXTO DEL SERVLET (SERVLET CONTEXT)

Un ***servlet*** vive y muere dentro de los límites del proceso del servidor. Por este motivo, puede ser interesante en un determinado momento obtener información acerca del entorno en el que se está ejecutando el ***servlet***. Esta información incluye la disponible en el momento de inicialización del ***servlet***, la referente al propio servidor o la información contextual específica que puede contener cada petición de servicio.

6.2.1 Información durante la inicialización del servlet

Esta información es suministrada al ***servlet*** mediante el argumento ***ServletConfig*** del método ***init()***. Cada ***servidor HTTP*** tiene su propia forma de pasar información al ***servlet***. En cualquier caso, para acceder a dicha información habría que emplear un código similar al siguiente:

```
String valorParametro;
public void init(ServletConfig config) {
    valorParametro = config.getInitParameter(nombreParametro);
}
```

Como puede observarse, se ha empleado el método ***getInitParameter()*** de la interface ***ServletConfig*** (implementada por ***GenericServlet***) para obtener el valor del parámetro. Asimismo, puede obtenerse una ***enumeración*** de todos los nombres de parámetros mediante el método ***getInitParameterNames()*** de la misma interface.

6.2.2 Información contextual acerca del servidor

La información acerca del servidor está disponible en todo momento a través de un objeto de la interface ***ServletContext***. Un ***servlet*** puede obtener dicho objeto mediante el método ***getServletContext()*** aplicable a un objeto ***ServletConfig***.

La interface ***ServletContext*** define los métodos descritos en la Tabla 3:

Métodos de ServletContext	**Comentarios**
public abstract Object **getAttribute(String)**	Devuelve información acerca de determinados atributos del tipo clave/valor del servidor. Es propio de cada servidor.
public abstract Enumeration **getAttributeNames()**	Devuelve una enumeración con los nombre de atributos disponibles en el servidor. **Sólo disponible en la versión *2.1***
public abstract String **getMimeType(String)**	Devuelve el tipo **MIME** de un determinado fichero.
public abstract String **getRealPath(String)**	Traduce una ruta de acceso virtual a la ruta relativa al lugar donde se encuentra el directorio raíz de páginas ***HTML***
public abstract String **getServerInfo()**	Devuelve el nombre y la versión del servicio de red en el que está siendo ejecutado el ***servlet***.
public abstract Servlet **getServlet(String)** throws ServletException	Devuelve un objeto ***servlet*** con el nombre dado. **Deprecado en la versión *2.1***, pues es un potencial foco de errores.
public abstract Enumeration **getServletNames()**	Devuelve un enumeración con los ***servlets*** disponibles en el servidor. **Deprecado en la versión *2.1***.
public abstract void **log(String)**	Escribe información en un fichero de *log*. El nombre del mismo y su formato son propios de cada servidor.

Tabla 3. Métodos de la interface ***ServletContext***.

6.3 CLASES DE UTILIDADES (UTILITY CLASSES)

El ***Servlet API*** proporciona una serie de utilidades que se describen a continuación.

- La primera de ellas es la interface ***javax.servlet.SingleThreadModel*** que puede hacer más sencillo el desarrollo de ***servlets***. Si un ***servlet*** implementa dicha interface, el servidor sabe que nunca debe llamar al método ***service()*** mientras esté procesando una petición anterior. Es decir, el servidor procesa todas las peticiones de servicio dentro de un mismo ***thread***. Sin embargo, a pesar de que esto puede facilitar el desarrollo de ***servlets***, puede ser un gran obstáculo en cuanto al rendimiento del ***servlet***. Por ello, a veces es preciso explorar otras opciones. Por ejemplo, si un ***servlet*** accede a una base de datos para su modificación, existen dos alternativas para evitar conflictos por accesos simultáneos:
  1. ***Sincronizar los métodos*** que acceden a los recursos, con la consiguiente complejidad en el código del ***servlet***.
  2. Implementar la ya citada interface ***SingleThreadModel***, solución más sencilla pero que trae consigo un aumento en el tiempo de respuesta. En este caso, no es necesario escribir ningún código adicional, basta con implementar la interface. Es una forma de ***marcar*** aquellos ***servlets*** que deben tener ese comportamiento, de forma que el servidor pueda identificarlos.
- El ***Servlet API*** incluye dos clases de ***excepciones***:
  1. La excepción ***javax.servlet.ServletException*** puede ser empleada cuando ocurre un fallo general en el ***servlet***. Esto hace saber al servidor que hay un problema.
  2. La excepción ***javax.servlet.UnavailableException*** indica que un ***servlet*** no se encuentra disponible. Los ***servlets*** pueden notificar esta excepción en cualquier momento. Existen dos tipos de indisponibilidades:
     a) ***Permanente*****:** El ***servlet*** no podrá seguir funcionando hasta que el administrador del servidor haga algo. En este estado, el ***servlet*** debería escribir en el fichero de *log* una descripción del problema, y posibles soluciones.
     b) ***Temporal***: El ***servlet*** se ha encontrado con un problema que es potencialmente temporal, como pueda ser un disco lleno, un servidor que ha fallado, etc. El problema puede arreglarse con el tiempo o puede requerir la intervención del administrador.

6.4 CLASE HTTPSERVLET: SOPORTE ESPECÍFICO PARA EL PROTOCOLO HTTP

Los ***servlets*** que utilizan el protocolo ***HTTP*** son los más comunes. Por este motivo, ***Sun*** ha incluido un package específico para estos ***servlets*** en su ***JSDK***: ***javax.servlet.http***. Antes de estudiar dicho ***package*** en profundidad, se va a hacer una pequeña referencia al protocolo ***HTTP***.

HTTP son las siglas de ***HyperText Transfer Protocol***, que es un protocolo mediante el cual los browser y los servidores puedan comunicarse entre sí, mediante la utilización de una serie de ***métodos: GET***, ***HEAD***, ***POST***, *PUT*, *DELETE*, *TRACE*, *CONNECT* y *OPTIONS*. Para la mayoría de las aplicaciones, bastará con conocer los tres primeros.

6.4.1 **Método GET: codificación de URLs**

El método ***HTTP GET*** solicita ***información*** a un ***servidor web***. Esta información puede ser un fichero, el resultado de un programa ejecutado en el servidor (como un ***servlet***, un ***programa CGI***, ...), etc.

En la mayoría de los servidores web los ***servlets*** son accedidos mediante un ***URL*** que comienza por ***/servlet/.*** El siguiente método ***HTTP GET*** solicita el servicio del ***servlet MiServlet*** al servidor ***miServidor.com***, con lo cual petición ***GET*** tiene la siguiente forma (en **negrita** el contenido de la petición):

```
GET /servlet/MiServlet?nombre=Antonio&Apellido=Lopez%20de%20Romera HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/4.5 (
  compatible;
  MSIE 4.01;
  Windows NT)
Host: miServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg
```

El ***URL*** de esta petición ***GET*** llama a un ***servlet*** llamado ***MiServlet*** y contiene dos parámetros, ***nombre*** y ***apellido***. Cada parámetro es un par que sigue el formato ***clave=valor***. Los parámetros se especifican poniendo un ***signo de interrogación*** (**?**) tras el nombre del ***servlet***. Además, los distintos parámetros están separados entre sí por el símbolo ***ampersand*** (**&**).

Obsérvese que la secuencia de caracteres ***%20*** aparece dos veces en el apellido. Es una forma de decir que hay un ***espacio*** entre “Lopez” y “de”, y otro entre “de” y “Romera”. Esto ocurre por la forma en que se codifican los ***URL*** en el protocolo ***HTTP***. Sucede lo mismo con otros símbolos como las tildes u otros caracteres especiales. Esta codificación sigue el esquema:

%+<valor hexadecimal del código ASCII correspondiente al carácter>

Por ejemplo, el carácter **á** se escribiría como ***%E1*** (código ASCII 225). También se puede cambiar la secuencia ***%20*** por el signo **+**, obteniendo el mismo efecto.

En cualquier caso, los programadores de ***servlets*** no deben preocuparse en principio por este problema, ya que la clase ***HttpServletRequest*** se encarga de la decodificación, de forma que los valores de los parámetros sean accesibles mediante el método ***getParameter(String parametro)*** de dicha clase. Sin embargo, hay que tener cuidado con algunos caracteres a la hora de incluirlos en un ***URL***, en concreto con aquellos caracteres no pertenecientes al código ASCII y con aquellos que tienen un significado concreto para el protocolo ***HTTP.*** Más en concreto, se pueden citar los siguientes caracteres especiales y su secuencia o código equivalente:

```
" (%22), # (%23), % (%25), & (%26), + (%2B), , (%2C), / (%2F),
: (%3A), < (%3C), = (%3D), > (%3E), ? (%3F) y @ (%40).
```

Adicionalmente, ***Java*** proporciona la posibilidad de codificar un ***URL*** de forma que cumpla con las anteriores restricciones. Para ello, se puede utilizar la clase ***URLEncoder***, que se encuentra incluida en el package ***java.net***, que es un package estándar de ***Java***. Dicha clase tiene un único método, ***String encode(String)***, que se encarga de codificar el ***String*** que recibe como argumento, devolviendo otro ***String*** con el ***URL*** debidamente codificado. Así, considérese el siguiente ejemplo:

```
import java.net.*;

public class Codificar {
    public static void main(String argv[]) {
        String URLcodificada=URLEncoder.encode("/servlet/MiServlet?nombre=Antonio"+
                                               "&Apellido=López de Romera");
        System.out.println(URLcodificada);
    }
}
```

que cuando es ejecutado tiene como resultado la siguiente secuencia de caracteres:

```
%2Fservlet%2FMiServlet%3Fnombre%3DAntonio%26Apellido%3DL%A2pez+de+Romera
```

Obsérvese además que, cuando sea necesario escribir una comilla dentro del ***String*** de ***out.println(String)***, hay que precederla por el carácter ***escape*** (\). Así, la sentencia:

```
out.println("<A HREF="http://www.yahoo.com">Yahoo</A>"); // INCORRECTA
```

es incorrecta y produce errores de compilación. Deberá ser sustituida por:

```
out.println("<A HREF=\"http://www.yahoo.com\">Yahoo</A>");
```

Las peticiones ***HTTP GET*** tienen una limitación importante (recuérdese que transmiten la información a través de las variables de entorno del sistema operativo) y es un límite en la cantidad de caracteres que pueden aceptar en el ***URL***. Si se envían los datos de un formulario muy extenso mediante ***HTTP GET*** pueden producirse errores por este motivo, por lo que habría que utilizar el método ***HTTP POST***.

Se suele decir que el método ***GET*** es ***seguro*** e ***idempotente*****:**

- ***Seguro***, porque no tiene ningún efecto secundario del cual pueda considerarse al usuario responsable del mismo. Es decir, por ejemplo, una llamada del método ***GET*** no debe ser capaz en teoría de alterar una base de datos. ***GET*** debería servir únicamente para obtener información.
- ***Idempotente***, porque puede ser llamado tantas veces como se quiera de una forma segura.

  Es como si ***GET*** fuera algo así como *ver pero no tocar*.

6.4.2 Método HEAD: información de ficheros

Este método es similar al anterior. La petición del cliente tiene la misma forma que en el método ***GET***, con la salvedad de que en lugar de ***GET*** se utiliza ***HEAD***. En este caso el servidor responde a dicha petición enviando únicamente ***información acerca del fichero***, y no el fichero en sí.

El método ***HEAD*** se suele utilizar frecuentemente para comprobar lo siguiente:

- La ***fecha de modificación*** de un documento presente en el servidor.
- El ***tamaño del documento*** antes de su descarga, de forma que el browser pueda presentar información acerca del progreso de descarga.
- El ***tipo de servidor***.
- El ***tipo de documento*** solicitado, de forma que el cliente pueda saber si es capaz de soportarlo.

El método ***HEAD***, al igual que ***GET***, es ***seguro*** e ***idempotente***.

6.4.3 Método POST: el más utilizado

El método ***HTTP POST*** permite al cliente ***enviar información al servidor***. Se debe utilizar en lugar de ***GET*** en aquellos casos que requieran transferir una cantidad importante de datos (formularios).

El método ***POST*** no tiene la limitación de ***GET*** en cuanto a volumen de información transferida, pues ésta no va incluida en el ***URL*** de la petición, sino que viaja encapsulada en un ***input stream*** que llega al ***servlet*** a través de la entrada estándar.

El encabezamiento y el contenido (en **negrita**) de una petición ***POST*** tiene la siguiente forma:

```
POST /servlet/MiServlet HTTP/1.1
User-Agent: Mozilla/4.5 (
   compatible;
   MSIE 4.01;
   Windows NT)
Host: www.MiServidor.com
Accept: image/gif, image/x-bitmap, image/jpeg, image/jpeg, */
```

```
Content-type: application/x-www-form-urlencoded
Content-length: 39

nombre=Antonio&Apellido=Lopez%20de%20Romera
```

Nótese la existencia de una ***línea en blanco*** entre el encabezamiento (*header*) y el comienzo de la información extendida. Esta línea en blanco indica el final del ***header***.

A diferencia de los anteriores métodos, ***POST no*** es ni ***seguro*** ni ***idempotente***, y por tanto es conveniente su utilización en aquellas aplicaciones que requieran operaciones más complejas que las de sólo-lectura, como por ejemplo modificar bases de datos, etc.

6.4.4 Clases de soporte HTTP

Una vez que se han presentado unas ciertas nociones sobre el protocolo ***HTTP***, resulta más sencillo entender las funciones del package ***javax.servlet.http***, que facilitan de sobremanera la creación de ***servlets*** que empleen dicho protocolo.

La clase abstracta ***javax.servlet.http.HttpServlet*** implementa la interface ***javax.servlet.Servlet*** e incluye un numero de importante de funciones adicionales. La forma más sencilla de escribir un ***servlet HTTP*** es heredando de ***HttpServlet*** como puede observarse en la Figura 7:

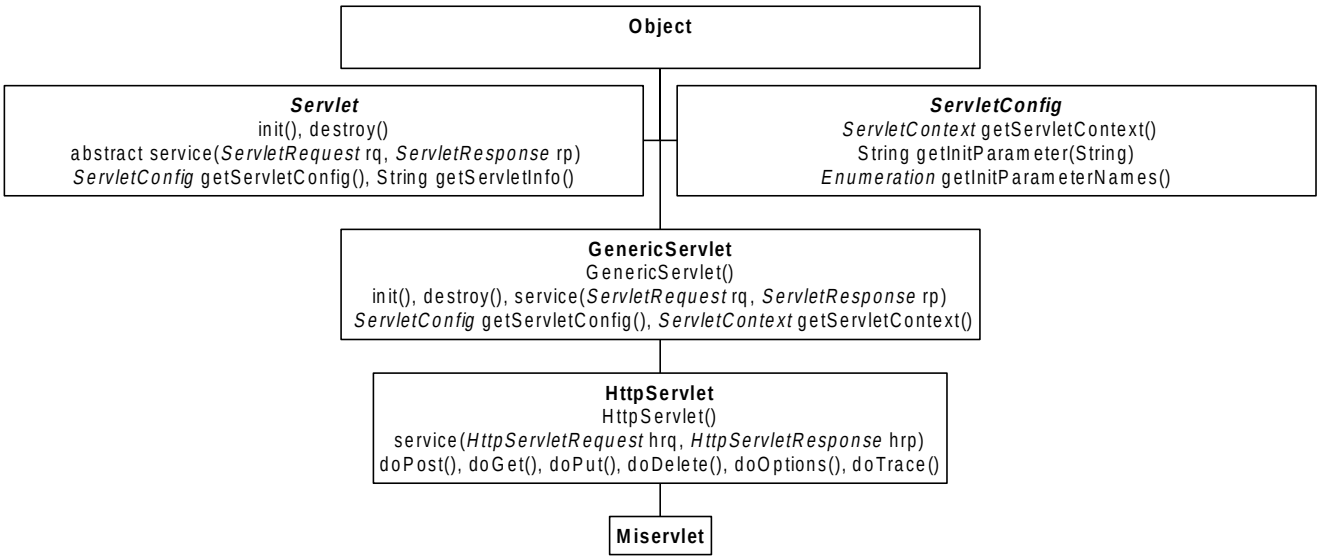

Figura 7. Jerarquía de clases en servlets.

La clase ***HttpServlet*** es también una clase ***abstract***, de modo que es necesario definir una clase que derive de ella y redefinir en la clase derivada ***al menos uno*** de sus métodos, tales como ***doGet(), doPost()***, etc.

Como ya se ha comentado, la clase ***HttpServlet*** proporciona una implementación del método ***service()*** en la que distingue qué método se ha utilizado en la petición (***GET, POST,*** etc.), llamando seguidamente al método adecuado (***doGet()***, ***doHead()***, ***doDelete()***, ***doOptions()***, ***doPost()*** y ***doTrace()***). Estos métodos e corresponden con los métodos ***HTTP*** anteriormente citados.

Así pues, la clase ***HttpServlet*** no define el método ***service()*** como ***abstract***, sino como ***protected***, al igual que los métodos ***init(), destroy(), doGet(), doPost(),*** etc., de forma que ya no es necesario escribir una implementación de ***service()*** en un ***servlet*** que herede de dicha clase. Si por algún motivo es necesario redefinir el método ***service()***, es muy conveniente llamar desde él al método ***service()*** de la super-clase (***HttpServlet***).

La clase ***HttpServlet*** es bastante "inteligente", ya que es también capaz de saber qué métodos han sido redefinidos en una sub-clase, de forma que puede comunicar al cliente qué tipos de métodos soporta el ***servlet*** en cuestión. Así, si en la clase ***MiServlet*** sólo se ha redefinido el método ***doPost()***, si el cliente realiza una petición de tipo ***HTTP GET*** el servidor lanzará automáticamente un mensaje de error similar al siguiente:

```
501 Method GET Not Supported
```

donde el número que aparece antes del mensaje es un código empleado por los ***servidores HTTP*** para indicar su estado actual. En este caso el código es el 501.

No siempre es necesario redefinir todos los métodos de la clase ***HttpServlet***. Por ejemplo, basta definir el método ***doGet()*** para que el ***servlet*** responda por sí mismo a peticiones del tipo ***HTTP HEAD*** o ***HTTP OPTIONS***.

6.4.5 Modo de empleo de la clase HttpServlet

Todos los métodos de clase ***HttpServlet*** que debe o puede redefinir el programador (***doGet()***, ***doPost()***, ***doPut()***, ***doOptions()***, etc.) reciben como argumentos un objeto ***HttpServletRequest*** y otro ***HttpServletResponse***.

Métodos de HttpServletRequest	**Comentarios**
getAuthType()	
getDateHeader(String)	
getHeader(String)	
getHeaderNames()	
getIntHeader(String)	
getMethod()	
getQueryString()	

Tabla 4. Métodos de la interface ***HttpServletRequest***.

La interface ***HttpServletRequest*** proporciona numerosos métodos para obtener información acerca de la petición del cliente (así como de la identidad del mismo, aunque estos serán estudiados en posteriores apartados). Entre otros (consultar la documentación del ***API*** para mayor información) se pueden citar los incluidos en la Tabla 4.

Métodos de HttpServletResponse	**Comentarios**
sendError(int), sendError(int, String)	
setStatus(int), setStatus(int, String)	
setHeader(String, String))	
setDateHeader(String, long)	

Tabla 5. Métodos de la interface ***HttpServletResponse***.

Por otra parte, el objeto de la interface ***HttpServletResponse*** permite enviar desde el ***servlet*** al cliente información acerca del estado del servidor (métodos ***sendError()*** y ***setStatus()***), así como establecer los valores del *header* del mensaje saliente (métodos ***setHeader(), setDateHeader()***, etc.). Algunos de estos métodos pueden verse en la Tabla 5.

Recuérdese que tanto ***HttpServletRequest*** como ***HttpServletResponse*** son interfaces que derivan de las interfaces ***ServletRequest*** y ***ServletResponse*** respectivamente, por lo que se pueden también utilizar todos los métodos declarados en estas últimas. Recuérdese que algunos métodos de

ServletRequest fueron descritos en la Tabla 1, página 27, mientras que los métodos de ***ServletResponse*** lo fueron en la Tabla 2, página 27.

Recuérdese a modo de recapitulación que el método ***doGet()*** debería :

1. Leer los datos de la solicitud, tales como los nombres de los parámetros y sus valores
2. Establecer el *header* de la respuesta (longitud, tipo y codificación)
3. Escribir la respuesta en formato HTML para enviarla al cliente.

Recuérdese que la implementación de este método debe ser ***segura*** e ***idempotente***.

El método ***doPost()*** por su parte, debería realizar las siguientes funciones:

1. Obtener input stream del cliente y leer los parámetros de la solicitud.
2. Realizar aquello para lo que está diseñado (actualización de bases de datos, etc.).
3. Informar al cliente de la finalización de dicha tarea o de posibles imprevistos. Para ello hay que establecer primero el tipo de la respuesta, obtener luego un PrintWriter y enviar a través suyo el mensaje HTML.

7 FORMAS DE SEGUIR LA TRAYECTORIA DE LOS USUARIOS (CLIENTES)

Los ***servlets*** permiten seguir la trayectoria de un cliente, es decir, obtener y mantener una determinada información acerca del cliente. De esta forma se puede ***tener identificado a un cliente*** (usuario que está utilizando un browser) durante un determinado tiempo. Esto es muy importante si se quiere disponer de aplicaciones que impliquen la ejecución de varios ***servlets*** o la ejecución repetida de un mismo ***servlet***. Un claro ejemplo de aplicación de esta técnica es el de los ***comercios vía Internet*** que permiten llevar un ***carrito de la compra*** en el que se van guardando aquellos productos solicitados por el cliente. El cliente puede ir navegando por las distintas secciones del comercio virtual, es decir realizando distintas conexiones ***HTTP*** y ejecutando diversos ***servlets***, y a pesar de ello no se pierde la información contenida en el carrito de la compra y se sabe en todo momento que es un mismo cliente quien está haciendo esas conexiones diferentes.

El mantener información sobre un cliente a lo largo de un proceso que implica múltiples conexiones se puede realizar de tres formas distintas:

1. Mediante ***cookies***
2. Mediante ***seguimiento de sesiones*** (***Session Tracking***)
3. Mediante la ***reescritura*** de URLs

7.1 COOKIES

Estrictamente hablando “cookie” significa galleta. Parece ser que dicha palabra tiene otro significado: se utilizaría también para la ficha que le dan a un cliente en un guardarropa al dejar el abrigo y que tiene que entregar para que le reconozcan y le devuelvan dicha prenda. Éste sería el sentido de la palabra ***cookie*** en el contexto de los ***servlets***: algo que se utiliza para que un ***servidor HTTP*** reconozca a un cliente como alguien que ya se había conectado anteriormente. Como era de esperar, los ***cookies*** en ***Java*** son objetos de la clase ***Cookie***, en el package ***javax.servlet.http***.

El empleo de ***cookies*** en el seguimiento de un cliente requiere que dicho cliente sea capaz de soportarlas. En el momento de escribir este documento, los browsers más extendidos tienen soporte para ***cookies***. Sin embargo, puede ocurrir que a pesar de estar disponible, dicha opción esté ***desactivada*** por el usuario, por lo que puede ser necesario emplear otras alternativas de seguimiento de clientes como la reescritura de ***URLs***. Esto es debido a que los ***servlets*** envían ***cookies*** a los clientes junto con la respuesta, y los clientes las devuelven junto con una petición. Así, si un cliente tiene activada la opción ***No cookies*** o similar en su navegador, no le llegará la ***cookie*** enviada por el ***servlet***, por lo que el seguimiento será imposible.

Cada ***cookie*** tiene un nombre que puede ser el mismo para varias ***cookies***, y se almacenan en un directorio o fichero predeterminado en el disco duro del cliente. De esta forma, puede mantenerse información acerca del cliente durante días, ya que esa información queda almacenada en el ordenador del cliente (aunque no indefinidamente, pues las ***cookies*** tienen una fecha de caducidad).

La forma en que se envían ***cookies*** es bastante sencilla en concepto. Antes ya se vio que era posible añadir campos adicionales al *header* de un mensaje ***HTTP***. De esta forma, añadiendo una ***clave*** y un ***valor*** al *header* del mensaje es posible enviar ***cookies*** al cliente, y desde éste al servidor. Adicionalmente, es posible incluir otros parámetros adicionales, tales como ***comentarios***. Sin embargo, estos no suelen ser tratados correctamente por los browsers actuales, por lo que su empleo es desaconsejable. Un servidor puede enviar más de una ***cookie*** al cliente (hasta veinte ***cookies***).

Las ***cookies*** almacenadas en el cliente son enviadas en principio sólo al servidor que las originó. Por este motivo (porque las ***cookies*** son enviadas al ***servidor HTTP*** y **no** al ***servlet***), los ***servlets*** que se ejecutan en un mismo servidor comparten las mismas ***cookies***.

La forma de implementar todo esto es relativamente simple gracias a la clase ***Cookie*** incluida en el ***Servlet API***. Para enviar una ***cookie*** es preciso:

1. Crear un objeto ***Cookie***
2. Establecer sus atributos
3. Enviar la ***cookie***

Por otra parte, para obtener información de una ***cookie***, es necesario:

1. Recoger todas las ***cookies*** de la petición del cliente
2. Encontrar la ***cookie*** precisa
3. Obtener el valor recogido en la misma

7.1.1 Crear un objeto Cookie

La clase ***javax.servlet.http.Cookie*** tiene un constructor que presenta como argumentos un ***String*** con el ***nombre*** de la ***cookie*** y otro ***String*** con su ***valor***. Es importante hacer notar que toda la información almacenada en ***cookies*** lo es en forma de ***String***, por lo que será preciso convertir cualquier valor a ***String*** antes de añadirlo a una ***cookie***.

Habrá de ser cuidadoso con los ***nombres*** empleados, ya que aquellos que contengan caracteres especiales pueden no ser válidos. Adicionalmente, aquellos que comienzan por el símbolo de dólar (**$**) no pueden emplearse, por estar reservados.

Con respecto al ***valor*** de la ***cookie***, en principio puede tener cualquier forma, aunque hay que tener cautela con el valor ***null***, que puede ser incorrectamente manejado por los browsers. Adicionalmente, si el browser empleado es ***Netscape*** deben evitarse aquellos valores que contengan un espacio en blanco o los siguientes caracteres:

```
[ ] ( ) = , " / ? @:;
```

Por último, es importante saber que es necesario crear la ***cookie*** antes de acceder al *Writer* del objeto ***HttpServletResponse***, pues como las ***cookies*** son enviadas al cliente en el *header* del mensaje, y éstas deben ser escritas antes de crear el *Writer*.

Por ejemplo, el siguiente código crea una ***cookie*** con el nombre "***Compra***" y el valor de ***IdObjetoAComprar***, que es una variable que contiene la identificación de un objeto a comprar (301):

```
...
String IdObjetoAComprar = new String("301");
if(IdObjetoAComprar!=null)
    Cookie miCookie=new Cookie("Compra", IdObjetoAComprar);
```

7.1.2 Establecer los atributos de la cookie

La clase ***Cookie*** proporciona varios métodos para establecer los valores de una ***cookie*** y sus atributos. Entre otros, los mostrados en la Tabla 6:

Métodos de la clase Cookie	**Comentarios**
public void **setComment(String)**	Si un browser presenta esta ***cookie*** al usuario, el cometido de la ***cookie*** será descrito mediante este comentario.
public void **setDomain(String)**	Establece el patrón de dominio a quien permitir el acceso a la información contenida en la ***cookie***. Por ejemplo .yahoo.com permite el acceso a la ***cookie*** al servidor www.yahoo.com pero no a a.b.yahoo.com
public void **setMaxAge(int)**	Establece el tiempo de caducidad de la ***cookie*** en segundos. Un valor -1 indica al browser que borre la ***cookie*** cuando se apague. Un valor 0 borra la ***cookie*** de inmediato.
public void **setPath(String)**	Establece la ruta de acceso del directorio de los ***servlets*** que tienen acceso a la ***cookie***. Por defecto es aquel que originó la ***cookie***.
public void **setSecure(boolean)**	Indica al browser que la ***cookie*** sólo debe ser enviada utilizando un protocolo seguro (***https***). Sólo debe utilizarse en caso de que el servidor que haya creado la ***cookie*** lo haya hecho de forma segura.
public void **setValue(String)**	Establece el valor de la ***cookie***
public void **setVersion(int)**	Establece la versión del protocolo de la ***cookie***.

Tabla 6. Métodos de la clase ***Cookie***.

Todos estos métodos tienen sus métodos ***getX()*** correspondientes incluidos en la misma clase.

Por ejemplo, se puede cambiar el valor de una ***cookie*** de la siguiente forma:

```
...
Cookie miCookie=new Cookie("Nombre", "ValorInicial");
miCookie.setValue("ValorFinal");
```

o hacer que sea eliminada al cerrar el browser:

```
miCookie.setMaxAge(-1);
...
```

7.1.3 Enviar la cookie

Las ***cookies*** son enviadas como parte del *header* de la respuesta al cliente. Por ello, tienen que ser añadidas a un objeto ***HttpServletResponse*** mediante el método ***addCookie(Cookie)***. Tal y como se ha explicado con anterioridad, esto debe realizarse antes de llamar al método ***getWriter()*** de ese mismo objeto. Sirva como ejemplo el siguiente código:

```
...
public void doGet(HttpServletRequest req, HttpServletResponse resp)
                  throws ServletException, IOException {
    ...
    Cookie miCookie=new Cookie("Nombre","Valor");
    miCookie.setMaxAge(-1);
    miCookie.setComment("Esto es un comentario");
    resp.addCookie(miCookie);

    PrintWriter out=resp.getWriter();
    ...
}
```

7.1.4 Recoger las cookies

Los clientes devuelven las ***cookies*** como parte integrante del *header* de la petición al servidor. Por este motivo, las ***cookies*** enviadas deberán recogerse del objeto ***HttpServletRequest*** mediante el método ***getCookies()***, que devuelve un array de objetos ***Cookie***. Véase el siguiente ejemplo:

```
...
Cookie miCookie = null;
Cookie[] arrayCookies = req.getCookies();
miCookie = arrayCookies[0];
...
```

El anterior ejemplo recoge la primera ***cookie*** del ***array de cookies***.

Por otra parte, habrá que tener cuidado, pues tal y como se ha mencionado con anterioridad, puede haber más de una ***cookie*** con el mismo nombre, por lo que habrá que detectar de alguna manera cuál es la ***cookie*** que se necesita.

7.1.5 Obtener el valor de la cookie

Para obtener el valor de una ***cookie*** se utiliza el método ***getValue()*** de la clase ***Cookie***. Obsérvese el siguiente ejemplo. Supóngase que se tiene una tienda virtual de libros y que un usuario ha decidido eliminar un libro del carro de la compra. Se tienen dos ***cookies*** con el mismo nombre (***compra***), pero con dos valores (***libro1***, ***libro2***). Si se quiere eliminar el valor ***libro1***:

```
...
String libroABorrar=req.getParameter("Borrar");
...
if(libroABorrar!=null) {
    Cookie[] arrayCookies=req.getCookies();
    for(i=0;i<arrayCookies.length;i++) {
        Cookie miCookie=arrayCookies[i];
        if(miCookie.getName().equals("compra")
             &&miCookie.getValue.equals("libro1") {
               miCookie.setMaxAge(0); // Elimina la cookie
        } // fin del if
    } // fin del for
} // fin del if
...
```

Tal y como se ha dicho con anterioridad, una ***cookie*** contiene como valor un ***String***. Este ***String*** debe ser tal que signifique algo para el ***servlet***. Con esto se quiere decir que es responsabilidad exclusiva del programador establecer que formato o codificación va a tener ese ***String*** que almacena la ***cookie***. Por ejemplo, si se tiene una tienda on-line, pueden establecerse tres posibles tipos de status de un producto (con referencia al interés de un cliente determinado por dicho producto): el cliente se ha solicitado información sobre el producto (**A**), el cliente lo tiene contenido en el carrito de la compra (**B**) o el cliente ya ha comprado uno anteriormente (**C**). Así, si por ejemplo el código del producto fuera el **301** y estuviera contenido en el carrito de la compra, podría enviarse una ***cookie*** con el siguiente valor:

```
Cookie miCookie = new Cookie("NombreDeCookie", "301_B");
```

El programador deberá establecer una codificación propia y ser capaz de descodificarlo posteriormente.

7.2 SESIONES (SESSION TRACKING)

Una ***sesión*** es una conexión continuada de un mismo browser a un servidor durante un tiempo prefijado de tiempo. Este tiempo depende habitualmente del servidor, aunque a partir de la versión ***2.1*** del ***Servlet API*** puede establecerse mediante el método ***setMaxInactiveInterval(int)*** de la interface ***HttpSession***. Esta interface es la que proporciona los métodos necesarios para mantener ***sesiones***.

Al igual que las ***cookies***, las ***sesiones*** son compartidas por todos los ***servlets*** de un mismo servidor. De hecho, por defecto se utilizan ***cookies*** de una forma implícita en el mantenimiento de

sesiones. Por ello, si el browser no acepta ***cookies***, habrá que emplearse las ***sesiones*** en conjunción con la reescritura de **URLs** (Ver apartado 7.3).

La forma de obtener una ***sesión*** es mediante el método ***getSession(boolean)*** de un objeto ***HttpServletRequest***. Si este ***boolean*** es ***true***, se crea una sesión nueva si es necesario mientras que si es ***false***, el método devolverá la ***sesión*** actual. Por ejemplo:

```
...
HttpSession miSesion = req.getSession(true);
...
```

crea una nueva ***sesión*** con el nombre ***miSesion***.

Una vez que se tiene un objeto ***HttpSession***, es posible mantener una colección de pares ***nombre de dato***/***valor de dato***, de forma que pueda almacenarse todo tipo de información sobre la ***sesión***. Este valor puede ser cualquier objeto de la clase ***Object*** que se desee. La forma de añadir valores a la ***sesión*** es mediante el método ***putValue(String ,Object)*** de la clase ***HttpSession*** y la de obtenerlos es mediante el método ***getValue(String , Object)*** del mismo objeto. Esto puede verse en el siguiente ejemplo:

```
...
HttpSession miSesion=req.getSesion(true);
CarritoCompras compra = (CarritoCompras)miSesion.getValue(miSesion.getId());

if(compra==null) {
    compra = new CarritoCompras();
    miSesion.putValue(miSesion.getId(), compra);
}
...
```

En este ejemplo, se supone la existencia de una clase llamada ***CarritoCompras***. En primer lugar se obtiene una nueva ***sesión*** (en caso de que fuera necesario, si no se mantendrá una creada previamente), y se trata de obtener el objeto ***CarritoCompras*** añadido a la ***sesión***. Obsérvese que para ello se hace una llamada al método ***getId()*** del objeto ***miSesion***. Cada ***sesión*** se encuentra identificada por un identificador único que la diferencia de las demás. Este método devuelve dicho identificador. Esta es una buena forma de evitar confusiones con el nombre de las ***sesiones*** y el de sus valores. En cualquier caso, al objeto ***CarritoCompras*** se le podía haber asociado cualquier otra clave. Si no se hubiera añadido previamente el objeto ***CarritoCompras*** a la ***sesión***, la llamada al método ***getValue()*** tendría como resultado ***null***. Obsérvese además, que es preciso hacer un **cast** para pasar el objeto ***Object*** a objeto ***CarritoCompras***.

En caso de que compra sea ***null***, es decir, que no existiera un objeto añadido previamente, se crea un nuevo objeto ***CarritoCompras*** y se añade a la sesión ***miSesion*** mediante el método ***putValue()***, utilizando de nuevo el identificador de la ***sesión*** como nombre.

Además de estos métodos mencionados, la interface ***HttpSession*** define los siguientes métodos:

- ***getCreationTime():*** devuelve el momento en que fue creado la ***sesión*** (en milisegundos).
- ***getLastAccessedTime():***devuelve el último momento en que el cliente realizó una petición con el identificador asignado a una determinada ***sesión*** (en milisegundos)
- ***getValueNames():*** devuelve un array con todos los nombres de los objetos asociados con la ***sesión***.
- ***invalidate():*** invalida la ***sesión*** en curso.
- ***isNew():*** devuelve un ***boolean*** indicando si la ***sesión*** es “nueva”.

- ***removeValue(String):*** elimina el objeto asociado con una determinada clave.

De todos los anteriores métodos conviene comentar dos en especial: ***invalidate()*** y ***isNew()***.

El método ***invalidate()*** invalida la sesión en curso. Tal y como se ha mencionado con anterioridad, una sesión puede ser invalidada por el propio servidor si en el transcurso de un intervalo prefijado de tiempo no ha recibido peticiones de un cliente. *Invalidar* quiere decir eliminar el objeto ***HttpSession*** y los valores asociados con él del sistema.

El método ***isNew()*** sirve para conocer si una sesión es *"nueva"*. El servidor considera que una sesión es nueva hasta que el cliente se una a la sesión. Hasta ese momento ***isNew()*** devuelve ***true***. Un valor de retorno ***true*** puede darse en las siguientes circunstancias:

- El cliente todavía no sabe nada acerca de la ***sesión***
- La ***sesión*** todavía no ha comenzado.
- El cliente no quiere unirse a la ***sesión***. Ocurre cuando el browser tiene la aceptación de ***cookies*** desactivada.

7.3 REESCRITURA DE URLS

A pesar de que la mayoría de los browser más extendidos soportan las ***cookies*** en la actualidad, para poder emplear ***sesiones*** con clientes que o bien no soportan ***cookies*** o bien las rechazan, debe utilizarse la reescritura de ***URLs***. No todos los servidores soportan la reescritura de ***URLs*** (por ejemplo el ***servletrunner*** que acompaña el ***JSDK***).

Para emplear está técnica lo que se hace es incluir el código identificativo de la ***sesión*** (***sessionId***) en el ***URL*** de la petición. Los métodos que se encargan de reescribir el ***URL*** si fuera necesario son ***HttpServletResponse.encodeUrl()*** y ***HttpServletResponse.encodeRedirectUrl()*** (sustituidas en el ***API 2.1*** por ***encodeURL()*** **y** ***encodeRedirectURL()*** respectivamente). El primero de ellos lee un ***String*** que representa un ***URL*** y si fuera necesario la reescribe añadiendo el identificativo de la ***sesión***, dejándolo inalterado en caso contrario. El segundo realiza lo mismo sólo que con ***URLs*** de redirección, es decir, permite reenviar la petición del cliente a otro ***URL*** .

Véase el siguiente ejemplo:

```
...
HttpSession miSesion=req.getSesion(true);
CarritoCompras compra = (CarritoCompras)miSesion.getValue(miSesion.getId());

if(compra==null) {
    compra = new CarritoCompras();
    miSesion.putValue(miSesion.getId(), compra);
}
...

PrintWriter out = resp.getWriter();
resp.setContentType("text/html");
...
out.println("Esto es un enlace reescrito");
out.println("<a href\""+
resp.encodeUrl("/servlet/buscador?nombre=Pedro")+"\"</a>");
...
```

En este caso, como hay un ***sesión***, la llamada al método ***encodeUrl()*** tendría como consecuencia la reescritura del enlace incluyendo el identificativo de la ***sesión*** en él.

8 FORMAS DE EJECUTAR UN SERVLET

Existe tres formas de llamar un ***servlet***:

- Escribiendo el ***URL*** del ***servlet*** en el campo de ***dirección*** (*location*) del browser. Debe seguir el siguiente esquema:

  ```
  http://servidor:puerto/servlet/nombre_de_servlet
  ```

  No siempre debe encontrarse el ***servlet*** en el directorio con el nombre ***servlet*** del servidor, aunque suele ser lo habitual. Depende de la configuración del servidor.

- También desde una página ***HTML*** puede llamarse a un ***servlet***. Para ello habrá de emplearse el ***tag*** adecuado. En el caso de que se trate simplemente de un enlace:

  ```
  <a href="http://localhost:8080/servlet/miServlet">Clique Aquí</a>
  ```

  Si se trata de un formulario, habrá que indicar el ***URL*** del ***servlet*** en la propiedad ***ACTION*** de la *tag* **<FORM>** y especificar el método ***HTTP*** (***GET***, ***POST***, etc.) en la propiedad ***METHOD*** también en la misma *tag*.

- Al tratarse de clases ***Java*** como las demás, pueden crearse objetos de dicha clase (ser *instanciadas*), aunque siempre con el debido cuidado de llamar a aquellos métodos de la clase instanciada que sean necesarios. En ocasiones es muy útil escribir ***servlets*** que realicen una determinada función y que sólo puedan ser llamados por otros ***servlets***. En dicho caso, será preciso redefinir su método ***service()*** de la siguiente forma:

  ```
  public void service(ServletRequest req, ServletResponse resp)
              throws ServletException, IOException {

       throw new UnavailableException(this,
          "Este servlet no acepta llamadas de clientes, solamente de otros servlets");
  } // fin del método service()
  ```

  En este caso el programador debe llamar explícitamente desde otro ***servlet*** los métodos del ***servlet*** que quiere ejecutar. Recuérdese que de ordinario los métodos ***service()***, ***doPost()***, ***doGet()***, etc. son llamados automáticamente cuando el ***servidor HTTP*** recibe una solicitud de servicio de un formulario introducido en una página ***HTML***.

9 ACCESO A BASES DE DATOS MEDIANTE *SERVLETS* Y JDBC

Una de las tareas más importantes y más frecuentemente realizadas por los ***servlets*** es la conexión a ***bases de datos*** mediante ***JDBC***. Esto es debido a que los ***servlets*** son un componente ideal para hacer las funciones de capa media en un sistema con una arquitectura de tres capas como la mostrada en la Figura 8.

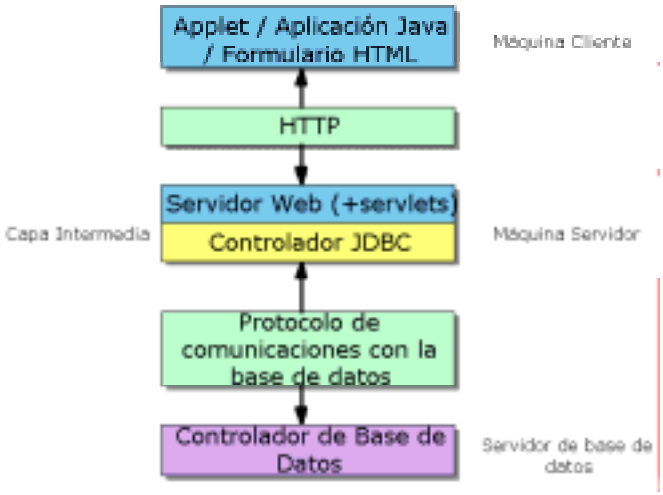

Figura 8. Arquitectura cliente-servidor de 3 capas.

Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los *drivers* ***JDBC*** no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de *driver*. En cualquier caso, tanto el ***Servidor HTTP*** como el ***Servidor de Base de Datos*** pueden estar en la misma máquina, aunque en sistemas empresariales de cierta importancia esto no suele ocurrir con frecuencia.

JDBC (***Java DataBase Connectivity***) es una parte del API de ***Java*** que proporciona clases para conectarse con bases de datos. Dichas clases forman parte del package ***java.sql***, disponible en el ***jdk 1.1.7*** y en ***jdk 1.2***. El nombre ***JDBC*** es fonéticamente similar a ***ODBC*** (***Open DataBase Connectivity***), que es el estándar más extendido para conectar PCs con bases de datos.

SQL (***Structured Query Language***) es un lenguaje estándar de alto nivel que permite leer, escribir y en general gestionar bases de datos.

Los ejemplos que se presentan en este apartado hacen uso de estos conceptos de una forma muy simple, pero perfectamente válida para lo que se pretende de los ***servlets*** en este documento.

La arquitectura de los ***servlets*** hace que la escritura de aplicaciones que se ejecuten en el servidor sea relativamente sencilla y que sean aplicaciones muy robustas. La principal ventaja de utilizar ***servlets*** es que se puede programar sin dificultad la información que va a proporcionar entre peticiones del cliente. Es decir, se puede tener constancia de lo que el usuario ha hecho en peticiones anteriores. Además, cada objeto del ***servlet*** se ejecuta dentro de un ***thread*** de ***Java***, por lo que se pueden controlar las interacciones entre múltiples objetos; y al utilizar el identificador de sincronización, se puede asegurar que los ***servlets*** del mismo tipo esperan a que se produzca la misma transacción, antes de procesar la petición; esto puede ser especialmente útil cuando mucha gente intenta actualizar al mismo tiempo la base de datos, o si hay mucha gente pendiente de consultas a la base de datos cuando ésta está en pleno proceso de actualización.

En efecto, las características ***multithread*** de los ***servlets*** hacen que se adecuen perfectamente a este tipo de tareas. Si el servidor de base de datos incluye soporte para múltiples conexiones simultáneas, incluso es posible establecer éstas una única vez e ir administrando las conexiones entre las sucesivas peticiones de servicio.

9.1 EJEMPLO 1: ESCRIBIR EN UNA BASE DE DATOS MICROSOFT ACCESS 97

El siguiente ejemplo muestra cómo compartir una única conexión entre todas las peticiones de servicio. Para ello, se retoma el ejemplo introductorio (ver en página 18), pero en lugar de mostrar en pantalla la información recogida en el formulario, se introducirá en una base de datos.

Para poder ejecutar este ejemplo en los PCs de las Salas de la ESIISS (o en el propio domicilio) se deben seguir los siguientes pasos:

1. Se deberá disponer de ***Microsoft Access 97***. Con dicho programa se debe crear una nueva base de datos, llamada por ejemplo ***ServletOpinion2.mdb***. En dicho fichero habrá que crear una tabla vacía llamada ***Opiniones_Recogidas*** cuyos campos se llamen ***nombre***, ***apellidos***, ***opinion*** y ***comentarios***, que sean respectivamente de tipo ***text***, ***text***, ***text*** y ***memo***.
2. A continuación se debe configurar el PC para que pueda conectarse con dicha base de datos. Esto se hace a través del ***ODBC***. Para ello se abre el ***Panel de Control ODBC*** y se elige la carpeta ***System DSN***. A continuación se clica en el botón ***Add*** y se selecciona el driver de ***Microsoft Access***. Después se clica en el botón ***Finish***. En este momento aparece un nuevo cuadro de diálogo con un campo llamado ***Data Source Name*** en el que se debe indicar el nombre con el que se quiere hacer referencia a la base de datos. El campo ***Description*** permite introducir un breve texto descriptivo. Clicando en el botón ***Select*** se debe elegir el fichero ***ServletOpinion2.mdb***. Con esto queda configurado el ***ODBC***.
3. Abrir una consola de ***MS-DOS*** y arrancar ***servletrunner.exe***. Se supone que la variable ***CLASSPATH*** tiene ya un valor correcto.
4. El siguiente fichero contiene la clase ***ServletOpinion2*** que se conectará con la base de datos mencionada escribiendo en ella las opiniones de los usuarios. Dichas opiniones pueden ser recogidas con el fichero ***MiServlet2.htm*** que se incluye a continuación de ***ServletOpinion2.java***. Es importante prestar atención a las ***líneas en negrita***, en las que se concentran los conceptos fundamentales de este ejemplo.

```
// fichero ServletOpinion2.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class ServletOpinion2 extends HttpServlet {

    // Declaración de variables miembro
    private String nombre = null;
    private String apellidos = null;
    private String opinion = null;
    private String comentarios = null;

    // Referencia a un objeto de la interface java.sql.Connection
    Connection conn = null;
```

```
// El siguiente método se ejecuta una única vez (al ser inicializado el servlet
//     por primera vez)
// Se suelen inicializar variables y ejecutar operaciones costosas en tiempo
//     de ejecución (abrir ficheros, conectar con bases de datos, etc)
public void init (ServletConfig config) throws ServletException {

      // Llamada al método init() de la superclase (GenericServlet)
      // Así se asegura una correcta inicialización del servlet
      super.init(config);

      // dsn (Data Source Name) de la base de datos
      String dsn = new String("jdbc:odbc:opinion");

      // Carga del Driver del puente JDBC-ODBC
      try {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch(ClassNotFoundException ex) {
          System.out.println("Error al cargar el driver");
          System.out.println(ex.getMessage());
      }
      // Establecimiento de la conexión con la base de datos
      try {
          conn = DriverManager.getConnection(dsn, "", "");
      } catch (SQLException sqlEx) {
          System.out.println("Se ha producido un error al" +
                             " establecer la conexión con: " + dsn);
          System.out.println(sqlEx.getMessage());
      }

      System.out.println("Iniciando ServletOpinion (version BD)...");
} // fin del método init()


// Este método es llamado por el servidor web al
//     "apagarse" (al hacer shut down).
// Sirve para proporcionar una correcta desconexión de una base de datos,
//     cerrar ficheros abiertos, etc.
public void destroy () {
       super.destroy();
       System.out.println("Cerrando conexion...");
       try {
            conn.close();
       }catch(SQLException ex){
            System.out.println("No se pudo cerrar la conexion");
            System.out.println(ex.getMessage());
       }
 } // fin del método destroy()

// Método de llamada mediante un HTTP POST
public void doPost (HttpServletRequest req, HttpServletResponse resp)
              throws ServletException, IOException {

    boolean hayError = false;

    // adquisición de los valores del formulario
    if(req.getParameter("nombre")!=null)
        nombre = req.getParameter("nombre");
    else
        hayError=true;

    if(req.getParameter("apellidos")!=null)
        apellidos = req.getParameter("apellidos");
    else
        hayError = true;

    if(req.getParameter("opinion")!=null)
        opinion = req.getParameter("opinion");
    else
        hayError = true;

    if(req.getParameter("comentarios")!=null)
        comentarios = req.getParameter("comentarios");
```

```
        else
            hayError = true;

        // Mandar al usuario los valores adquiridos
        //     (Si no se ha producido error)
        if(!hayError) {
            if (actualizarBaseDeDatos() == 0)
                devolverPaginaHTML(resp);
            else
                resp.sendError(500, "Se ha producido un error"+
                                " al actualizar la base de datos");
        } else
            resp.sendError(500, "Se ha producido un error"+
                                " en la adquisición de parámetros");
    } // fin doPost()


    public int actualizarBaseDeDatos() {
        // crear un statement de SQL
        Statement stmt=null;
        int numeroFilasActualizadas=0;

        // Ejecución del query de actualización de la base de datos
        try {
            stmt = conn.createStatement();
            numeroFilasActualizadas = stmt.executeUpdate("INSERT INTO"+
                            " Opiniones_Recogidas VALUES"+
                            "('"+nombre+"','"+apellidos+"','"+opinion+
                            "','"+comentarios+"')");
            if(numeroFilasActualizadas!=1) return -1;
        } catch (SQLException sql) {
             System.out.println("Se produjo un error creando Statement");
             System.out.println(sql.getMessage());
             return -2;
        } finally {
             // Se cierra el Statement
             if(stmt!=null) {
                 try {
                     stmt.close();
                 } catch(SQLException e){
                     System.out.println("Error cerrando Statement");
                     System.out.println(e.getMessage());
                     return -3;
                 }
             }
             return 0;
        } // fin finally
    } // fin método actualizarBaseDeDatos()


    public void devolverPaginaHTML(HttpServletResponse resp) {
        // Se obtiene un PrintWriter donde escribir (sólo para mandar texto)
        PrintWriter out=null;
        try {
            out=resp.getWriter();
        } catch (IOException io) {
            System.out.println("Se ha producido una excepcion");
        }

        // Se establece el tipo de contenido MIME de la respuesta
        resp.setContentType("text/html");

        // Se mandan los valores
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Valores recogidos en el formulario</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<b><font size=+2>Valores recogidos del");
        out.println("formulario: </font></b>");
        out.println("<p><font size=+1><b>Nombre: </b>"+nombre+"</font>");
```

```
            out.println("<br><fontsize=+1><b>Apellido : </b>"
                        +apellidos+"</font><b><font size=+1></font></b>");
            out.println("<p><font size=+1><b>Opini&oacute;n: "+
                        "</b><i>"+opinion+"</i></font>");
            out.println("<br><font size=+1><b>Comentarios: </b>"
                         +comentarios+"</font>");

            out.println("<P><HR><CENTER><H2>Valores actualizados "+
                                "con éxito</CENTER>");

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

            // Se fuerza la descarga del buffer y se cierra el PrintWriter
            out.flush();
            out.close();
        } // fin de devolverPaginaHTML()

        // Función que permite al servidor web obtener una
        //     pequeña descripción del servlet, qué
        //     cometido tiene, nombre del autor, comentarios adicionales, etc.
        public String getServletInfo() {
            return "Este servlet lee los datos de un formulario "+
                "y los introduce en una base da datos";
        } // fin de getServletInfo()

    } // fin de la clase servletOpinion2
```

```
<!-- Fichero MiServlet2.htm -->
<HTML>
<HEAD>
    <TITLE>Envíe su opinión</TITLE>
</HEAD>

<BODY>

<H2>Por favor, envíenos su opinión acerca de este sitio web</H2>

<FORM ACTION="http://C50.ceit.es:8080/servlet/ServletOpinion2" METHOD="POST">

    Nombre: <INPUT TYPE="TEXT" NAME="nombre" SIZE=15><BR>
    Apellidos: <INPUT TYPE="TEXT" NAME="apellidos" SIZE=30><P>

    Opinión que le ha merecido este sitio web<BR>
    <INPUT TYPE="RADIO" CHECKED NAME="opinion" VALUE="Buena">Buena<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Regular">Regular<BR>
    <INPUT TYPE="RADIO" NAME="opinion" VALUE="Mala">Mala<P>

    Comentarios <BR>
    <TEXTAREA NAME="comentarios" ROWS=6 COLS=40>
    </TEXTAREA><P>

    <INPUT TYPE="SUBMIT" NAME="botonEnviar" VALUE="Enviar">
    <INPUT TYPE="RESET" NAME="botonLimpiar" VALUE="Limpiar">

</FORM>

</BODY>
</HTML>
```

Nota importante: En este fichero la valor del parámetro ***ACTION*** depende del ordenador en el que se esté trabajando. En el ejemplo se ha supuesto que se está en el ordenador ***50*** del ***Aula C***. Si se tratase de otro ordenador de las ***Salas de PCs*** de la ESIISS habría que cambiar ***C50*** por la letra de la sala y el número de PC correspondiente. Si se está en otro ordenador se puede poner su ***número de IP*** (suponiendo que lo tenga) o un ***número genérico*** que indica que el ***servlet*** se va a ejecutar en

el ordenador local (127.0.0.1). En este caso hay que configurar el browser (Netscape o Internet Explorer) con la opción "***No proxies***" o "***Conexión directa a Internet***".

La ejecución de este ***servlet*** tiene como resultado el siguiente cambio en la base de datos reflejado en la Figura 9:

Figura 9. Resultado obtenido en la base de datos.

y una pantalla similar a la del ejemplo introductorio en el browser. Cada vez que se ejecuta el formulario de añade una nueva línea a la tabla ***Opiniones_Recogidas***.

En este ejemplo cabe resaltar lo siguiente:

- La conexión con la base de datos se hace por medio de un driver que convierte de ***JDBC*** a ***ODBC***. La carga de dicho *driver* ***JDBC-ODBC*** tiene lugar durante la inicialización del ***servlet***, al igual que la ***conexión*** con la base de datos. Basta pues con una única ***conexión*** a la base de datos para satisfacer todas las peticiones de los clientes (a priori, ya que puede ser necesario tener más conexiones abiertas si se espera un tráfico intenso). Además, se ha supuesto que el ***DSN*** del Sistema de ***ODBC*** de 32 bits contenía la entrada ***opinion*.**
- A su vez, en el método ***destroy()*** se ha implementado la ***desconexión*** de la base de datos, de forma que no quede recurso alguno ocupado una vez que el ***servlet*** haya sido descargado.
- En el método ***doPost()*** se ha comprobado que la adquisición de los parámetros del formulario era correcta, enviando un error al cliente en caso contrario. También podría haberse comprobado que ningún campo estuviera vacío.
- El método ***actualizarBaseDeDatos()*** es el que se encarga de la introducción de los valores en la base de datos, mientras que el método ***devolverPaginaHTML()*** se encarga de la presentación en pantalla de los valores leídos. En la ejecución del ***query*** en ***actualizarBaseDeDatos()*** se tienen muy en cuenta posibles excepciones que pudieran surgir y se devuelve como valor de retorno un código de error distinto de cero para esos casos.
- Asimismo, mediante el bloque ***finally*** se consigue cerrar siempre el ***Statement*** tanto si se produce el error como si no, liberando recursos del sistema de esa manera.
- Téngase especial cuidado con la sintaxis del ***query***, y en especial con las comillas simples en que debe ir envuelto cada ***String***.
- Tras mostrar en pantalla los datos recogidos, se comprueba si se ha producido alguna actualización. Si este es el caso, se informa al cliente de esta circunstancia, mostrando un mensaje de error en caso contrario.

9.2 EJEMPLO 2: CONSULTAR UNA BASE DE DATOS CON ACCESS 97

Este segundo ejemplo pretende mostrar la forma en que se accede a datos contenidos en bases de datos. Para ello, se ha escrito un programa empleando un ***servlet*** que recibe como parámetro el

nombre un grupo de prácticas (parámetro ***GRUPO***), y muestra la lista de los alumnos que están en ese grupo, así como algunos de sus datos personales (número de carnet, nombre, apellidos, curso).

El usuario selecciona el grupo que quiere ver en el formulario mostrado en la Figura 10:

Figura 10. Formulario del Ejemplo 2.

El código **HTML** correspondiente a la página mostrada en la Figura 10 es el siguiente:

```
<!-- fichero Formulario2.htm -->
<html>

<head>
    <title>Grupos de prácticas</title>
</head>

<body>

<h2 align="center">Escoja el grupo de prácticas cuya lista desea ver</h2>

<form method="GET" action="http://wdisney:8080/servlet/ListaAlumnos"
      name="Formulario">
  <div align="center"><center><p>
  <input type="radio" value="Grupo1" checked name="GRUPO">Grupo1 
  <input type="radio" name="GRUPO" value="Grupo2">Grupo2 
  <input type="radio" name="GRUPO" value="Grupo3">Grupo3
  </p></center></div>
  <div align="center"><center><p>
  <input type="submit" value="Enviar" name="BotonEnviar">
  <input type="reset" value="Borrar" name="BotonBorrar">
  </p></center></div>
</form>

</body>
</html>
```

El formulario se compone de tres ***radio buttons*** con los nombres de los grupos disponibles, y de los botones ***Enviar*** y ***Borrar***. Por otra parte, el método ***HTTP*** empleado en este ejemplo ha sido ***HTTP GET***. Por este motivo, si se clica en ***Enviar*** el ***URL*** solicitado al servidor tiene la siguiente forma:

```
http://wdisney:8080/servlet/ListaAlumnos?GRUPO=Grupo1&BotonEnviar=Enviar
```

Si se selecciona el ***Grupo1*** y se clica en ***Enviar***, se obtiene el resultado que se muestra en la Figura 11:

Figura 11. Lista de alumnos del Grupo 1

El ***servlet*** empleado para obtener es bastante simple. He aquí el código del ***servlet***:

```
// fichero ListaAlumnos.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;

public class ListaAlumnos extends HttpServlet {

    Connection conn = null;
    // Vector que contendrá los objetos Alumno
    Vector vectorAlumnos=null;

    // Este método se ejecuta una única vez (al ser inicializado
    //     por primera vez el servlet)
    // Se suelen inicializar variables y ejecutar operaciones costosas en tiempo
    //     de ejecución (abrir ficheros, conectar con bases de datos, etc)
    public void init (ServletConfig config) throws ServletException {

        // Llamada al método init() de la superclase (GenericServlet)
        // Así se asegura una correcta inicialización del servlet
        super.init(config);

        // url de la base de datos
        String url=new String("jdbc:odbc:alumnos");

        // Carga del Driver
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch(ClassNotFoundException ex) {
            System.out.println("Error al cargar el driver");
            System.out.println(ex.getMessage());
        }
```

```
    // Establecimiento de la conexión
    try {
        conn=DriverManager.getConnection(url,"","");
    } catch (SQLException sqlEx) {
        System.out.println("Se ha producido un error al establecer "+
                           "la conexión con: "+url);
        System.out.println(sqlEx.getMessage());
    }

    System.out.println("Iniciando ListaAlumnos");
} // fin del método init()


// Este método es llamado por el servidor web al "apagarse"
//    (al hacer shut down). Sirve para proporcionar una correcta
//    desconexión de una base de datos, cerrar ficheros abiertos, etc.
public void destroy () {
    super.destroy();
    System.out.println("Cerrando conexion...");
    try {
        conn.close();
    } catch(SQLException ex){
        System.out.println("No se pudo cerrar la conexion");
        System.out.println(ex.getMessage());
    }
} // fin de destroy()


// Método llamada mediante un HTTP POST
public void doGet (HttpServletRequest req, HttpServletResponse resp)
                   throws ServletException, IOException {

    // Obtención del grupo de prácticas
    String grupo = null;
    grupo = req.getParameter("GRUPO");

    if(grupo==null) {
        resp.sendError(500, "Se ha producido un error en la lectura " +
                            "de la solicitud");
        return;
    }

    // Consulta a la base de datos para obtener la lista de alumnos de un grupo
    if(obtenerLista(grupo)==0) {
        // Mostrar la lista de alumnos mediante una página HTML
        mostrarListaAlumnos(resp, grupo);
    }
    else if(obtenerLista(grupo)==-3) {
        resp.sendError(500, "No se ha encontrado el grupo: " +grupo);
    }
    else
        resp.sendError(500, "Se ha producido un error en el acceso " +
                            "a la base de datos");

} // fin del método doGet()


public int obtenerLista(String grupo) {

    Statement stmt = null;
    ResultSet rs = null;

    // Ejecución del query
    try {
        stmt=conn.createStatement();
```

```
            rs=stmt.executeQuery("SELECT DISTINCT TablaAlumnos.Nombre, " +
                   "TablaAlumnos.Apellidos, "+
                   "TablaAlumnos.Curso, "+
                   "TablaAlumnos.Carnet, "+
                   "GruposDePracticas.GrupoPractica "+
                   "FROM GruposDePracticas "+
                   "INNER JOIN TablaAlumnos ON "+
                   "GruposDePracticas.Carnet = TablaAlumnos.Carnet "+
                   "WHERE (((GruposDePracticas.GrupoPractica)='"+grupo+"'))");

            vectorAlumnos=new Vector();
            //Alumno temp=new Alumno();

            // Lectura del ResultSet
            // En Java2
            while (rs.next()) {
                Alumno temp=new Alumno();
                temp.setNombre(rs.getString("Nombre"));
                temp.setApellidos(rs.getString("Apellidos"));
                temp.setCarnet(rs.getLong("Carnet"));
                temp.setCurso(rs.getInt("Curso"));
                vectorAlumnos.addElement(temp);
            }

            // En el JDK 1.1.x hay un bug que hace que no se lean bien los valores
            //     que no son Strings
            /* while (rs.next()) {
                 Alumno temp=new Alumno();
                 temp.setNombre(rs.getString("Nombre"));
                 temp.setApellidos(rs.getString("Apellidos"));
                 temp.setCarnet(java.lang.Long.parseLong(rs.getString("Carnet")));
                 temp.setCurso(java.lang.Integer.parseInt(rs.getString("Curso")));
                 vectorAlumnos.addElement(temp);
            }*/
            if(vectorAlumnos.size()==0)
                return -3;

        return 0;

        } catch (SQLException sql) {
             System.out.println("Se produjo un error al crear el Statement");
             System.out.println(sql.getMessage());
             return -1;
        } finally {
            // se cierra el Statment
            if(stmt!=null) {
                try {
                    stmt.close();
                } catch(SQLException e) {
                    System.out.println("Error al cerrar el Statement");
                    System.out.println(e.getMessage());
                    return -2;
                }
            }
       } // fin del finally

    } // fin del método obtenerLista()


    public void mostrarListaAlumnos(HttpServletResponse resp, String grupo) {

        // se establece el tipo de contenido MIME de la respuesta
        resp.setContentType("text/html");

        // se obtiene un PrintWriter donde escribir (sólo para mandar texto)
        PrintWriter out=null;
        try {
            out=resp.getWriter();
        } catch (IOException io) {
            System.out.println("Se ha producido una excepcion");
        }
```

```
            // se manda la lista
            out.println("<html>");
            out.println("");
            out.println("<head>");
            out.println("<title>Lista de alumnos del grupo "+grupo+"</title>");
            out.println("<meta name=\"GENERATOR\" "+
                        "content=\"Microsoft FrontPage 3.0\">");
            out.println("</head>");
            out.println("");
            out.println("<body>");
            out.println("");
            out.println("<H2 align=\"center\">Lista de alumnos del grupo "+
                        grupo+"</H2>");
            out.println("<div align=\"center\"><center>");
            out.println("");
            out.println("<table border=\"1\" width=\"70%\">");
            out.println("<tr>");
            out.println("<td width=\"25%\" bgcolor=\"#808080\">"+
                        "<font color=\"#FFFFFF\">Carnet</font></td>");
            out.println("<td width=\"25%\" bgcolor=\"#808080\">"+
                        "<font color=\"#FFFFFF\">Nombre</font></td>");
            out.println("<td width=\"25%\" bgcolor=\"#808080\">"+
                        "<font color=\"#FFFFFF\">Apellidos</font></td>");
            out.println("<td width=\"25%\" bgcolor=\"#808080\">"+
                        "<font color=\"#FFFFFF\">Curso</font></td>");
            out.println("</tr>");

            // Datos del Alumno por filas
            Alumno alum=null;
            for (int i=0; i<vectorAlumnos.size();i++) {
                alum=(Alumno)vectorAlumnos.elementAt(i);
                out.println("<tr>");
                out.println("<td width=\"25%\">"+alum.getCarnet()+"</td>");
                out.println("<td width=\"25%\">"+alum.getNombre()+"</td>");
                out.println("<td width=\"25%\">"+alum.getApellidos()+"</td>");
                out.println("<td width=\"25%\">"+alum.getCurso()+"</td>");
                out.println("</tr>");
            }

            out.println("</table>");
            out.println("</center></div>");
            out.println("</body>");
            out.println("</html>");

            // se fuerza la descarga del buffer y se cierra el PrintWriter
            out.flush();
            out.close();
        } // fin del método mostrarListaAlumnos()


        // Función que permite al servidor web obtener una pequeña descripción del
        // servlet, qué cometido tiene, nombre del autor, comentarios adicionales, etc.
        public String getServletInfo() {
            return "Este servlet muestra en pantalla la lista de un grupo de prácticas";
        }

    } // fin de la clase ListaAlumnos
```

Puede observarse que este ***servlet*** es bastante parecido al del ejemplo previo, en cuanto a su forma general. Tiene como el anterior un método ***init()*** donde efectúa la conexión con la base de datos cuyo ***DSN*** es ***alumnos***, y comprueba que la conexión se ha realizado con éxito. Asimismo, tiene un método ***destroy()*** donde se cierra dicha conexión y se avisa de posibles eventualidades.

Sin embargo, a diferencia del ***servlet*** del ejemplo anterior, la petición del cliente es de tipo ***HTTP GET,*** por lo que se ha redefinido el método ***doGet()*** y no el ***doPost()*** como en el caso anterior. En este método, lo primero que se hace es leer el parámetro ***GRUPO*** dentro del método

doGet(). En caso de que haya algún problema en la lectura de dicho parámetro, lanza un mensaje de error.

Una vez que se sabe cuál es el grupo cuya lista quiere visualizar el cliente, se llama al método ***obtenerLista(String)***, que tiene como parámetro precisamente el nombre del grupo a mostrar. En este método se realiza la consulta con la base de datos, mediante el método ***executeQuery()*** de la interface ***Statement***. La sentencia ***SQL*** a ejecutar ha sido obtenida mediante los procedimientos descritos en el ***Anexo*** de este manual. Dicha sentencia es bastante compleja de construir a mano, por lo que se recomienda seguir el procedimiento explicado en el ***Anexo*** dedicado a ***SQL***.

La ejecución de dicho método tiene como resultado un objeto ***ResultSet***, que es como una especie de ***tabla*** cuyas ***filas*** son cada uno de los alumnos, y cuyas ***columnas*** son los datos solicitados de ese alumno (***carnet***, ***nombre***, ***apellidos***, ***curso***, ...). Para obtener dichos datos se emplea el método ***next()*** del objeto ***ResultSet*** en conjunción con los métodos ***getXXX()*** de dicho objeto. El método ***next()*** devuelve ***true*** si hay más filas disponibles en el ***ResultSet*** (hay que ir leyendo las filas una por una, empezando por la primera hasta llegar a la última) y actualiza el número de fila, cuyos valores (columnas) se leen mediante los métodos ***getXXX()***, donde ***XXX*** representa el tipo de dato recogido en la base de datos (***String***, ***long***, ***int***, etc.). En el ***JDK 1.1.x*** existía un ***bug*** por el cual ***Java*** fallaba en determinadas circunstancias al intentar leer valores de campos con métodos distintos del ***getString()*** (por ejemplo: ***getLong(), getInt()***, etc.). Esto puede ser solventado empleando el método ***getString()*** y después convirtiendo el ***String*** resultante a los distintos tipos primitivos deseados (***int***, ***long***, etc.). En ***Java 2*** dicho ***bug*** ha sido corregido, y pueden emplearse directamente los métodos adecuados sin ningún problema.

En este ejemplo, además, al leer los valores de la base de datos, estos han sido almacenados en un ***Vector*** de objetos de la clase ***Alumno***, que ha sido creada para este ejemplo, y cuyo código puede observarse a continuación.

```
public class Alumno {

    // Definición de variables miembro
    private String nombre;
    private String apellidos;
    private long carnet;
    private int curso;
    private String grupoPractica;

    // Métodos para establecer los datos
    public void setNombre(String nom) { nombre=nom; }
    public void setApellidos(String apel) { apellidos=apel; }
    public void setCarnet(long carn) { carnet=carn; }
    public void setCurso(int cur) { curso=cur; }
    public void setGrupoPractica(String grupo) { grupoPractica=grupo; }

    // Métodos de recuperación de datos
    public String getNombre() { return nombre; }
    public String getApellidos() { return apellidos; }
    public long getCarnet() { return carnet; }
    public int getCurso() { return curso; }
    public String getGrupoPractica() { return grupoPractica; }

} // fin de la clase Alumno
```

Esta clase lo único que hace es definir las variables que van a contener los datos de cada alumno, y establecer los métodos de introducción y recuperación de dichos datos.

Siguiendo con el método ***obtenerLista(String)***, su función es comprobar que se ha obtenido al menos una fila como resultado de la consulta y capturar posibles excepciones, retornando los correspondientes códigos de error si fuera necesario.

Por último, volviendo al método ***doGet()*** que se estaba describiendo, sólo queda llamar al método ***mostrarListaAlumnos(HttpServletResponse resp, String grupo)***, en caso de que no se haya producido ningún error. Este método es análogo al método ***devolverPaginaHTML()*** del ejemplo anterior. En este caso, muestra una tabla ***HTML*** que va construyendo dinámicamente, añadiendo tantas filas a la misma como alumnos estén contenidos en el vector de alumnos.

10 ANEXO: INTRODUCCIÓN A SQL (STRUCTURED QUERY LANGUAGE)

SQL (***Structured Query Language*** o ***Lenguaje Estructurado de Consultas***) es un lenguaje empleado para crear, manipular, examinar y manejar ***bases de datos relacionales***. Proporciona una serie de sentencias estándar que permiten realizar las tareas antes descritas. ***SQL*** fue estandarizado según las normas ***ANSI*** (*American National Standards Institute*) en 1992, paliando de alguna forma la incompatibilidad de los productos de los distintos fabricantes de bases de datos (***Oracle***, ***Sybase***, ***Microsoft***, ***Informix***, etc.). Esto quiere decir que una misma sentencia permite a priori manipular los datos recogidos en cualquier base de datos que soporte el estándar ANSI, con independencia del tipo de base de datos.

La mayoría de los programas de base de datos más populares soportan el estándar ***SQL-92***, y adicionalmente proporcionan ***extensiones*** al mismo, aunque éstas ya no están estandarizadas y son propias de cada fabricante. ***JDBC*** soporta el estándar ***ANSI SQL-92*** y exige que cualquier driver ***JDBC*** sea compatible con dicho estándar.

Para poder enviar sentencias ***SQL*** a una base de datos, es preciso que un programa escrito en ***Java*** esté previamente conectado a dicha base de datos, y que haya un objeto ***Statement*** disponible.

10.1 REGLAS SINTÁCTICAS

SQL tiene su propia sintaxis que hay que tener en cuenta, pues a veces puede ocurrir que sin producirse ningún problema en la compilación, al tratar de ejecutar una sentencia se produzca algún error debido a una incorrecta sintaxis en la sentencia. Por tanto, será necesario seguir las siguientes normas:

- ***SQL*** no es sensible a los espacios en blanco. Los retornos de carro, tabuladores y espacios en blanco no tienen ningún significado especial. Las palabras clave y comandos están delimitados por ***comas*** (,), y cuando sea necesario, debe emplearse el paréntesis para agruparlos.
- Si se van a realizar múltiples consultas a un mismo tiempo, se debe utilizar el ***punto y coma*** (;) para separar cada una de las consultas. Además, todas las ***sentencias SQL*** deben finalizar con el carácter ***punto y coma*** (;).
- Las ***consultas*** son ***insensibles a mayúsculas y minúsculas***. Sin embargo, los valores almacenados en las bases de datos sí que son sensibles a las mismas, por lo que habrá que tener cuidado al introducir valores, efectuar comparaciones, etc.
- Para ***caracteres reservados*** como % o _ pueden y deben emplearse los ***caracteres escape***. Así por ejemplo en lugar de % habrá de emplearse \%.
- A la hora de introducir un ***String***, éste deberá ir encerrado ***entre comillas simples***, ya que de lo contrario se producirán errores en la ejecución.

10.2 EJECUCIÓN DE SENTENCIAS SQL

La interface ***Statement*** proporciona dos métodos distintos de ejecución de ***sentencias SQL*** en función del tipo de sentencia que se vaya a ejecutar:

- ***executeQuery(String)***: Sirve para recuperar información contenida en una base de datos. Ejecuta la ***consulta*** (***query***) pasada como parámetro y devuelve un objeto ***Resultset*** como resultado de la consulta.

- ***executeUpdate(String)***: Análogo al anterior, con la diferencia de que este método no sirve para realizar una consulta, sino para ***modificar o introducir datos*** en la base de datos. Tiene como valor de retorno un ***int***, que indica el número de filas actualizadas.

10.2.1 Tipos de datos SQL y equivalencia

SQL emplea unos tipos (***String***, **int**, etc) distintos a los de ***Java***. La Tabla 7 muestra los tipos más empleados y su equivalencia:

Tipo de dato Java	Tipo de dato SQL
int	INTEGER
long	BIG INT
float	REAL
double	FLOAT
BigDecimal	DECIMAL
boolean	BIT
String	VARCHAR
String	CHAR
Date	DATE
Time	TIME

Tabla 7. Relación entre los tipos de datos de Java y SQL.

10.2.2 Creación de tablas

Para la creación de tablas se emplea la sentencia ***CREATE TABLE***. El formato de la sentencia es:

```
CREATE TABLE <nombre de tabla> (<elemento columna> [<elemento columna>]...)
```

donde el ***elemento columna*** se declara en la forma:

```
<nombre columna> <tipo de dato> [DEFAULT <expresión>]
```

Todo aquello encerrado entre corchetes representa elementos opcionales, no necesarios. Así, para crear un tabla de nombre ***ALUMNOS*** que contuviera el ***nombre*** y ***apellidos*** del alumno como ***Strings*** y el número de ***carnet*** como un ***long***, se tendría que ejecutar la siguiente sentencia:

```
CREATE TABLE ALUMNOS (Nombre CHAR(15), Apellidos VARCHAR (30), Carnet INTEGER) ;
```

Esta sentencia crea una tabla de nombre ***ALUMNOS*** con tres campos:

- ***Nombre***, que es de tipo CHAR y admite hasta 15 caracteres
- ***Apellidos***, que es de tipo VARCHAR y admite hasta 30 caracteres
- ***Carnet***, que es de tipo INTEGER (int)

La diferencia entre ***CHAR*** y ***VARCHAR*** es que si el valor introducido en un campo es inferior al tamaño asignado al mismo (15 y 30 en este caso), en el primer caso rellena el espacio con espacios, y en el segundo lo deja tal y como está.

Obsérvese por otra parte que tal y como se ha mencionado antes, las mayúsculas/minúsculas no influyen en el resultado final. En efecto:

```
CREATE TABLE ALUMNOS (Nombre CHAR(15), Apellidos VARCHAR (30), Carnet INTEGER) ;
```

tiene el mismo resultado que la primera sentencia. En la Figura 12 puede observarse el resultado de la ejecución de la sentencia CREATE TABLE:

Figura 12. Resultado de ejecutar la sentencia CREATE TABLE.

10.2.3 Recuperación de información

La sentencia ***SELECT*** es la que se utiliza cuando se quieren recuperar datos de la información almacenada en un ***conjunto de columnas***. Las columnas pueden pertenecer a ***una o varias tablas*** y se puede indicar el criterio que deben seguir las filas de información que se extraigan. Muchas de las cláusulas que permite esta sentencia son simples, aunque se pueden conseguir capacidades muy complejas a base de una gramática más complicada.

La sintaxis de la sentencia es:

```
SELECT [ALL | DISTINCT] <seleccion>
FROM <tablas>
WHERE <condiciones de seleccion>
[ORDER BY <columna> [ASC | DESC]
[,<columna> [ASC | DESC]]...]
```

La selección contiene normalmente una ***lista de columnas*** separadas por comas (,), o un asterisco (*) para seleccionarlas todas. Un ejemplo ejecutado contra una de las tablas creadas anteriormente podría ser:

```
SELECT * FROM ALUMNOS;
```

que devolvería el contenido completo de la tabla ***ALUMNOS***. Si solamente se quiere conocer los datos del alumno cuyo número de carnet es 12345, la consulta sería:

```
SELECT * FROM ALUMNOS WHERE Carnet = 12345;
```

Se quiere ahora saber ***cuántos alumnos hay*** de nombre “Mikel”. Para ordenar la lista resultante por apellidos, por ejemplo, se usaría la directiva ***ORDER BY***:

```
SELECT * FROM ALUMNOS
WHERE Nombre = ‘Mikel’
ORDER BY Apellidos;
```

Puede especificarse que esta ordenación es realizada de forma ***ascendente*** (cláusula ***ASC***) o ***descendente*** (cláusula ***DESC).*** Así, si se quiere ordenar de forma ascendente:

```
SELECT * FROM ALUMNOS
WHERE Nombre = ‘Mikel’
ORDER BY Apellidos ASC;
```

Si lo que se quiere, además de que la lista esté ordenada por ***apellidos***, es ver solamente el número de carnet, se consultaría de la forma:

```
SELECT Carnet FROM ALUMNOS
WHERE Nombre = ‘Mikel’
ORDER BY Apellidos;
```

Si se quieren resultados de ***dos tablas*** a la vez, tampoco hay problema en ello, tal como se muestra en la siguiente sentencia:

```
SELECT ALUMNOS.*, PROFESORES.* FROM ALUMNOS, PROFESORES;
```

Obsérvese que se especifica el nombre de la tabla (***ALUMNOS*** o ***PROFESORES***) a la hora de acceder a los campos de una tabla. Así:

```
SELECT ALUMNOS.CampoDeAlumnos, PROFESORES.CampoDeProfesores FROM ALUMNOS,
         PROFESORES;
```

Por otra parte, puede utilizarse el carácter % como comodín para especificar términos de consulta que ***empiecen*** por un determinado valor. Así, para recuperar todos los datos de los alumnos cuyo nombre comienza por "M":

```
SELECT * FROM ALUMNOS WHERE Nombre LIKE 'M%' ;
```

Obsérvese que ***M%*** está encerrado entre comillas simples ('). Puede asimismo emplearse la cláusula ***NOT LIKE***, para indicar por ejemplo que se quieren recuperar todos los datos de los alumnos cuyo nombre no empiece por "B":

```
SELECT * FROM ALUMNOS WHERE Nombre NOT LIKE 'B%' ;
```

Adicionalmente pueden emplearse los operadores relacionales recogidos en la Tabla 8:

Operador relacional	**Significado**
=	Igual
<> o !=	Distinto
<	Menor
>	Mayor
<=	Menor o igual
>=	Mayor o igual

Tabla 8. Operadores relacionales.

Así, para obtener los nombres de aquellos alumnos cuyo número de carnet sea mayor que 70000, se emplearía la siguiente sentencia:

```
SELECT Nombres FROM ALUMNOS WHERE Carnet>70000 ;
```

Por otra parte, pueden emplearse las cláusulas lógicas ***AND***, ***OR*** y/o ***NOT***. Así, si se quisiera obtener todos los datos de los alumnos cuyo número de carnet es superior a 70000 y cuyo nombre es "Mikel", se debería emplear:

```
SELECT * FROM ALUMNOS WHERE Carnet>70000 AND Nombre='Mikel';
```

El orden de los operadores lógicos es el siguiente:

1. ***NOT***
2. ***AND***
3. ***OR***

Por último, indicar que el empleo de la cláusula ***DISTINCT*** elimina cualquier ***fila duplicada*** que pueda obtenerse como resultado de una consulta a varias tablas relacionadas entre sí.

10.2.4 Almacenar información

La sentencia ***INSERT*** se utiliza cuando se quieren ***insertar filas*** de información en una tabla. Aquí también se pueden presentar diferentes capacidades, dependiendo del nivel de complejidad soportado. La sintaxis de la sentencia es:

```
INSERT INTO <nombre tabla>
[(<nombre columna> [,<nombre columna]...)]
VALUES (<expresion> [,<expresion>]...)
```

Por ejemplo, en la tabla de los ***ALUMNOS*** se podría ingresar uno nuevo con la siguiente información:

```
INSERT INTO ALUMNOS VALUES ( 'Juan', 'Pérez Etxeberria', 23456);
```

10.2.5 Eliminación de datos

La sentencia ***DELETE*** es la que se emplea cuando se quieren ***eliminar filas*** de las columnas, y su gramática también es muy simple:

```
DELETE FROM <nombre tabla> WHERE <condicion busqueda>
```

Si no se especifica la cláusula ***WHERE***, se eliminará el contenido de la tabla completamente, sin eliminar la tabla, por ejemplo:

```
DELETE FROM ALUMNOS;
```

vaciará completamente la tabla, dejándola sin ningún dato en las columnas, es decir, esencialmente lo que hace es borrar todas las columnas de la tabla. Especificando la cláusula ***WHERE***, se puede introducir un criterio de selección para el borrado, por ejemplo:

```
DELETE FROM ALUMNOS WHERE Carnet=12345;
```

10.2.6 Actualización de datos

Para actualizar filas ya existentes en las columnas, se utiliza la sentencia ***UPDATE***, cuya gramática es la siguiente:

```
UPDATE <nombre tabla> SET <nombre columna = ( <expresion> | NULL )
[, <nombre columna = ( <expresion> | NULL )]... WHERE <condicion busqueda>
```

Este comando permite ***cambiar uno o más campos*** existentes en una fila. Por ejemplo, para cambiar el nombre de un alumno en la tabla de ***ALUMNOS***, se haría:

```
UPDATE ALUMNOS SET Nombre = 'Amaia' WHERE Carnet=12345;
```

10.3 SENTENCIAS SQL CON MICROSOFT ACCESS

Todo lo anteriormente expuesto sobre la sintaxis de las sentencias ***SQL*** se simplifica mucho mediante el empleo de opciones recogidas en los distintos productos de bases de datos. En el caso concreto de ***Microsoft Access***, es posible diseñar una ***consulta*** para recuperar datos de una forma sencilla y directa.

Para ello, habrá que ***crear una consulta*** dentro de la base de datos que se esté empleando.

Se estudiará a continuación cómo generar con ***Microsoft Access*** el código ***SQL*** del ***Ejemplo 2*** (ver Apartado9.2, página 49). En dicho ejemplo, se listan los datos de los alumnos que forman un determinado ***grupo***. La base de datos se encuentra diseñada de forma que hay ***dos tablas***:

- ***TablaAlumnos***, que contiene los datos (***Carnet***, ***Nombre***, ***Apellidos***, ***Curso***) de los alumnos.
- ***GruposDePracticas***, que contiene los campos ***Carnet*** y ***GrupoPractica***.

Ambas tablas están ***relacionadas*** mediante el campo ***Carnet***, que es la ***clave principal*** en ambas tablas.

Para ***generar el código de la consulta***, será preciso seguir los siguientes pasos:

1. En primer lugar será preciso crear un ***query*** o consulta. Para ello, estando abierta la base de datos, dentro de la pantalla principal de ***Microsoft Access*** se selecciona la lengüeta ***Queries*** (Consultas) y se clica en ***New***.
2. En la pantalla que aparece, se selecciona la opción ***Design View*** (Modo de diseño) y se clica en ***OK***.
3. Aparecerá otra nueva pantalla (***Show Table***) que listará las tablas y consultas presentes en la base de datos. Será preciso elegir ***aquellas tablas que contengan campos*** que se vayan a emplear en la consulta. Como en este caso, se quieren obtener los datos personales de aquellos alumnos de un grupo determinado, habrá que seleccionar las dos tablas, clicando en al botón ***Add*** para que dichas tablas seleccionadas se incluyan en la ventana que se encuentra al fondo. Una vez finalizado se clica en ***Close*** (Cerrar).
4. A continuación se ***seleccionarán aquellos campos que se quieran incluir*** en la consulta. En este caso, se van a emplear todos los campos (***Carnet***, ***GrupoPractica***, ***Nombre***, ***Apellidos***, ***Curso***). Según se vaya haciendo doble click en los campos, estos irán apareciendo en la parte inferior de la pantalla. También es posible seleccionar todos los campos de una vez, clicando en el símbolo asterisco (*).
5. Tal y como se ha dicho antes, en la parte inferior de la pantalla aparecen los campos que van a aparecer en la consulta. Además, aparecen los siguientes datos (para cada campo):
   - ***Field*** (Campo): Tal y como su nombre indica es el nombre del campo a incluir.
   - ***Table*** (Tabla): Representa el nombre de la tabla a que pertenece dicho campo.
   - ***Sort*** (Ordenar): Si se clica en él, aparece una lista desplegable que permite establecer si los resultados tienen que estar ordenados ascendentemente (***Ascending***) o descendentemente (***Descending***) según los valores de un determinado campo, o no estar ordenados. Así, si se quisiera que el resultado de la consulta de grupos de práctica estuviera ordenado ascendentemente según el número de carnet, habría que clicar en la celda ***Sort*** que hay a la altura de ***Carnet***, y elegir la opción ***Ascending***. En caso de querer establecer varios criterios de ordenación en una misma consulta, será preciso tener en cuenta que los criterios de ordenación tienen una precedencia de izquierda a derecha.
   - ***Show*** (Mostrar): Permite seleccionar si el campo correspondiente se va a mostrar o no.
   - ***Criteria*** (Criterio): Establece el criterio de selección. Así, si se quiere mostrar los datos de los alumnos cuyo grupo de prácticas es el ***Grupo1***, deberá escribirse en la celda ***Criteria*** del campo ***GrupoPractica*** "***Grupo1***".

   En la Figura 13 se muestra el estado de la ventana para el query que emplea el Ejemplo 2:

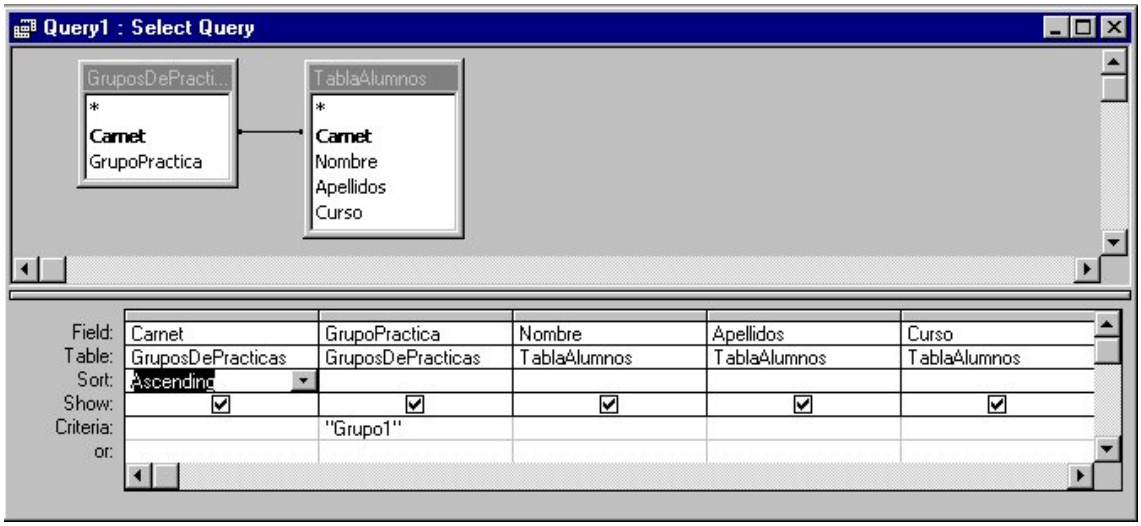

Figura 13. Relación entre campos de dos tablas.

6. Ya sólo queda ver el código ***SQL*** generado. Para ello, basta con seleccionar la opción ***SQL View*** en el menú ***View*** de la ventana principal. El código que aparece puede ser copiado dentro de los métodos de ***JDBC*** que ejecutan ***queries SQL***. Aquí está el código generado en este caso:

```
SELECT GruposDePracticas.Carnet, GruposDePracticas.GrupoPractica,
       TablaAlumnos.Nombre, TablaAlumnos.Apellidos, TablaAlumnos.Curso

FROM GruposDePracticas INNER JOIN TablaAlumnos ON GruposDePracticas.Carnet =
       TablaAlumnos.Carnet

WHERE (((GruposDePracticas.GrupoPractica)="Grupo1"))

ORDER BY GruposDePracticas.Carnet;
```

   Sin embargo hay que tener cuidado con las ***comillas dobles*** de "***Grupo1***", que deben ser sustituidas por comillas simples (‘***Grupo1***’).

7. Adicionalmente, puede ejecutarse la ***query*** mediante la opción ***Run*** del menú ***Query***, a fin de observar si los resultados son los deseados. Además es posible salvar la ***query*** para posteriores modificaciones.