CAPÍTULO 9

Servlets

9.1. Clientes y Servidores

9.1.1. 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***. 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).

9.1.2. 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.

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.

9.2. 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 9.1 representa gráficamente este concepto.

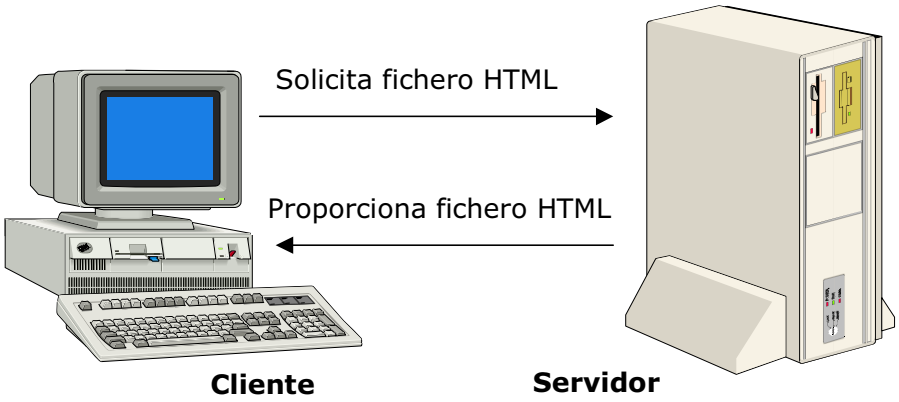

Figura 9.1. Arquitectura cliente-servidor tradicional.

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>. Las ***applets de Java*** tienen 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.

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. 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***[2] 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 9.2 muestra el esquema básico de funcionamiento de los ***programas CGI***.

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.

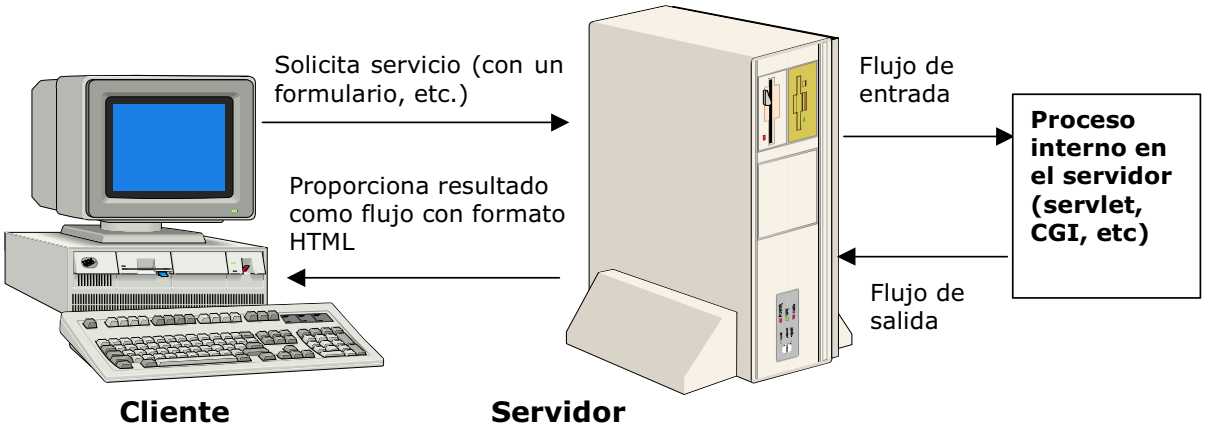

Figura 9.2. Arquitectura cliente-servidor interactiva para la WEB.

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***.

---

[2] 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.

9.3. 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 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.

9.4. 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.

9.5. JSDK 2.0

El ***JSDK*** (***Java Servlet Developer Kit***), distribuido gratuitamente por ***Sun***, proporciona el conjunto de herramientas necesarias para el desarrollo de ***servlets***. Su instalación se realiza a través de un fichero de 950 Kbytes, llamado ***jsdk20-Win32.exe***, que está disponible en la zona de recursos de la web la asignatura. 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.

9.5.1. 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 paquetes: ***javax.servlet*** y ***javax.servlet.http***. Todas las clases e interfaces que hay que utilizar en la programación de ***servlets*** están en estos dos paquetes.

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 9.3. En dicha figura se representan las ***clases*** con letra normal y las ***interfaces*** con *cursiva*.

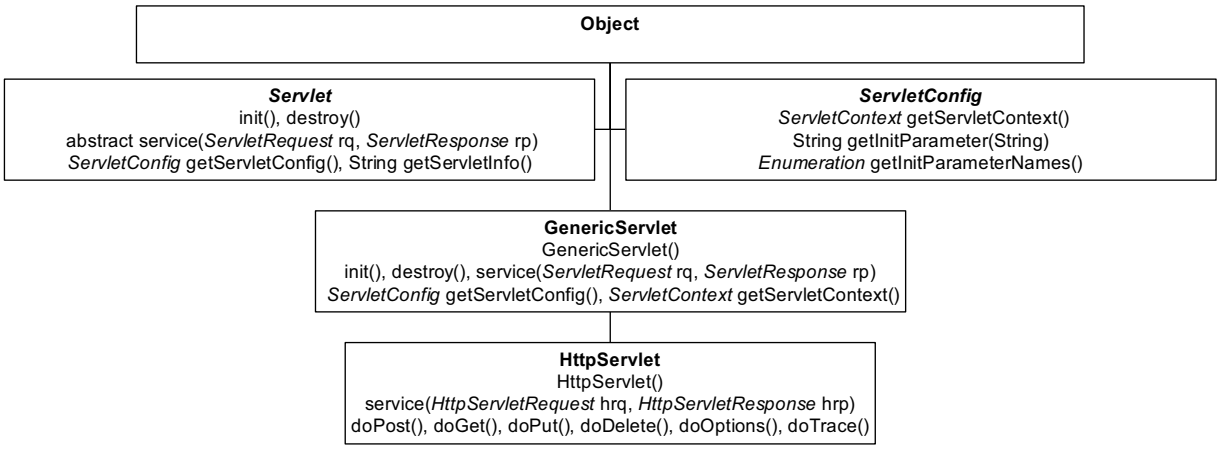

Figura 9.3. Jerarquía y métodos de las principales clases para crear servlets.

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 9.3 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.

9.5.2. 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 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***).

9.5.3. 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***:

```
servlet.nombre.code
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.

9.5.4. 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
```

9.6. 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.).

9.6.1. 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 ESI. Para realizar esta instalación en un ordenador propio se pueden seguir los siguientes pasos:

En primer lugar se debe conseguir el fichero de instalación, llamado ***jsdk20-win32.exe***. Este fichero se puede obtener de la zona de recursos de la página web de la asignatura. Se trata de un fichero de 950 Kbytes, que puede ser transportado en un disquete sin dificultad.

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.

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.

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.

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:

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

9.6.2. 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*** o ***Mala***). Por último, se ofrece al usuario la posibilidad de escribir un ***comentario*** si así lo considera oportuno. En la Figura 9.4 puede observarse el diseño del formulario creado. 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://miServidor: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 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:

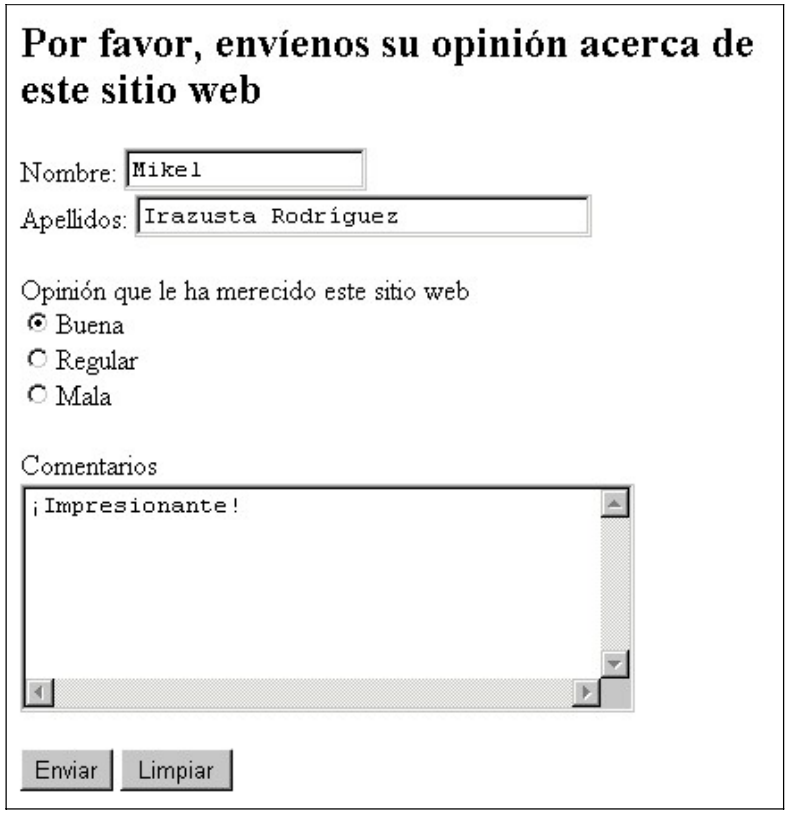

Figura 9.4. Diseño del formulario de adquisición de datos.

- El ***servlet*** se encuentra situado en un servidor cuyo nombre es ***miServidor*** (un ejemplo más real podría ser ***www.tecnun.es***). Este nombre dependerá del ordenador que proporcione los servicios de red. En cualquier caso, para poder hacer pruebas, se puede utilizar como nombre de servidor el *host local* o ***localhost***, o su ***número IP*** si se conoce. Por ejemplo, se podría escribir:

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

  o de otra forma,

```
<FORM ACTION="http://Numero_IP: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 la ***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***.

9.6.3. 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) {

        // 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 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 9.5. 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.

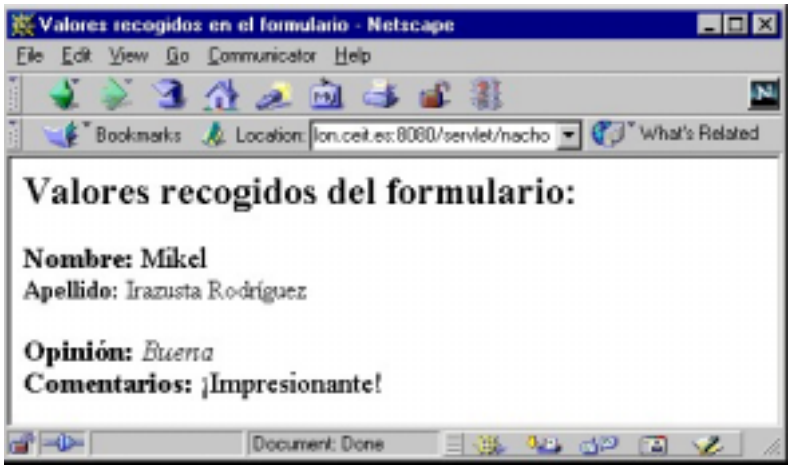

Figura 9.5. Página HTML devuelta por el servlet.

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.
- 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 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 esta tecnología. En posteriores apartados se efectuará un estudio más detallado de las clases y métodos empleados en este pequeño ejemplo.

9.7. 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 paquetes ***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***

9.7.1. El ciclo de vida de un servlet: clase GenericServlet

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. 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 9.6.

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()***

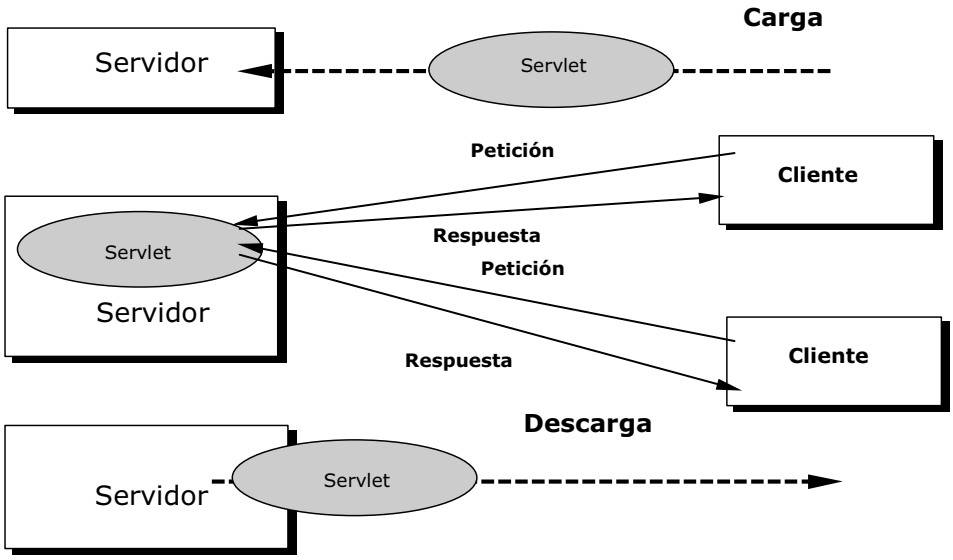

Figura 9.6. Ciclo de vida de un servlet.

9.7.1.1. 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***.

9.7.1.2. 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 9.1.
- 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 9.2 indica los métodos de la interface ***ServletResponse***.

Puede observarse en la Tabla 9.1 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.

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.

9.7.1.3. El método destroy() en la clase GenericServlet:

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.

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 9.1. Métodos de la interface ***ServletRequest***.

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 9.2. Métodos de la interface ***ServletResponse***.

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
```

9.7.2. 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.

9.7.2.1. Información durante la inicialización del servlet

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.
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.
public abstract Enumeration **getServletNames()**	Devuelve un enumeración con los ***servlets*** disponibles en el servidor.
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 9.3. Métodos de la interface ***ServletContext***.

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.

9.7.2.2. 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 9.3.

9.7.3. 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.

9.7.4. 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.

9.7.4.1. 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*.

9.7.4.2. 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***.

9.7.4.3. 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.

9.7.4.4. 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*** incluye un numero de importante de funciones adicionales e implementa la interface ***javax.servlet.Servlet***. La forma más sencilla de escribir un ***servlet HTTP*** es heredando de ***HttpServlet*** como puede observarse en la Figura 9.7.

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.

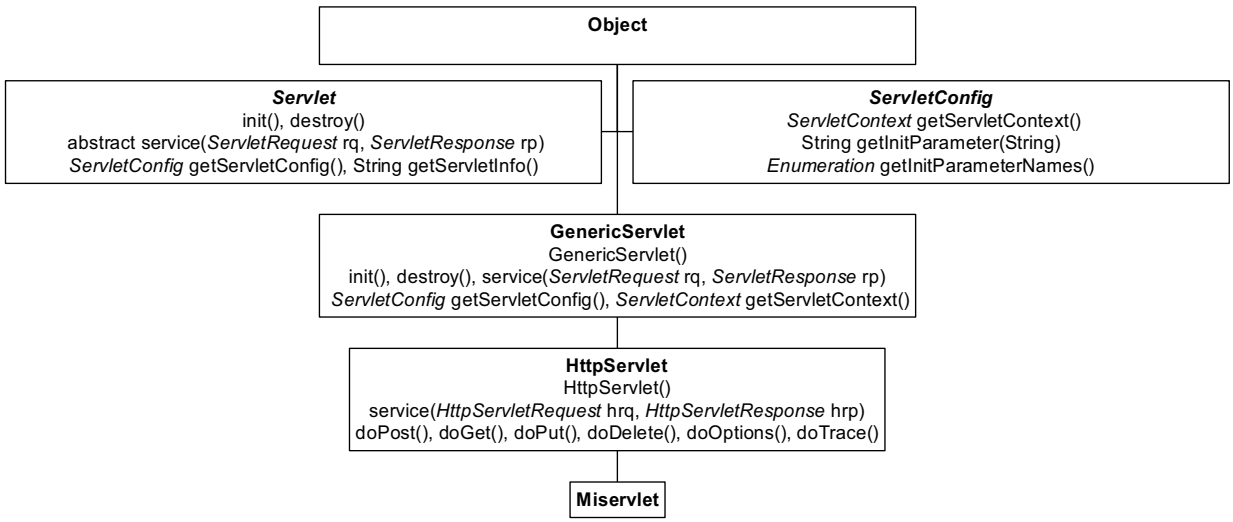

Figura 9.7. Jerarquía de clases en servlets.

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***.

9.7.4.5. 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***.

La interface ***HttpServletRequest*** proporciona numerosos métodos para obtener información acerca de la petición del cliente (así como de la identidad del mismo). Consultar la documentación del ***API*** para mayor información. 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.).

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 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.