Capítulo 3

Otros conceptos previos

Contenido

3.1 Servlets y filtros Java en Tomcat

En este apartado se pretende proporcionar al lector algunas nociones básicas sobre el despliegue de servlets y filtros de servlets en Tomcat. Su conocimiento simplificará mucho las explicaciones que, sobre el sistema Java implementado, ofreceremos en las secciones posteriores.

Un **servlet** es una aplicación Java que corre en un servidor, usando como soporte de ejecución un contenedor de aplicaciones como, por ejemplo, Tomcat (uno de los más populares). En el lado del servidor, un servlet Java es a Tomcat lo que, en el lado del

cliente, un applet Java es a un navegador web. El servlet se ejecuta "encima" de Tomcat; el applet, "encima" del navegador.

En un servidor donde está activo Tomcat, un servlet puede asociarse con una determinada ruta dentro del árbol de directorios. Cuando solicitemos dicha ruta mediante el protocolo HTTP el servlet entrará en acción. En concreto, se creará un objeto servlet de Java.

Para poder utilizar un servlet es necesario desplegarlo, es decir, informar a Tomcat de su existencia y sus características, para que éste sea capaz de poner el servlet en marcha, en caso de recibir una petición de ejecución proveniente del navegador.

En realidad, lo que se despliega es un elemento que representa un concepto más amplio, una **aplicación web** o ***web application***. Para informar a Tomcat sobre una aplicación web y poder así desplegarla, se utiliza el archivo web.xml (cada aplicación web tiene el suyo propio) y que recibe el nombre de **descriptor de despliegue** o ***deployment descriptor***. Dentro de dicho archivo podrían aparecer descripciones de servlets, que estarían, por tanto, incluidos dentro de dicha aplicación web. En dichas descripciones le indicaremos a Tomcat, entre otras cosas, qué clase Java debe instanciar cuando quiera arrancar un determinado servlet.

Otro paso necesario para que Tomcat despliegue una aplicación web es "empaquetarla" en un archivo WAR. Las clases Java que forman el código perteneciente a todos los servlets de la aplicación web, se compilan y se almacenan comprimidas dentro de un archivo de este tipo. También se añade a este archivo el descriptor de despliegue del que se habló en el párrafo anterior. El nombre del archivo WAR determina el nombre de la aplicación web y la ruta que se le asociará dentro del servidor. Por debajo de esa ruta general asociada a la aplicación web, podremos asociar "subrutas" a alguno de los servlets incluidos dentro de dicha aplicación.

Veamos un ejemplo: supongamos que disponemos de un servidor web donde corre Tomcat, cuya dirección es www.miservidorcontomcat.com. Hemos programado y compilado un servlet en Java y ahora queremos desplegarlo en dicho servidor. Para ello escribimos el descriptor de despliegue de una aplicación web, en el que haremos

referencia al servlet que hemos programado, asignándole la ruta /miservlet e indicándole a Tomcat qué clase Java de dicho servlet debe empezar a ejecutar si recibe una petición del navegador. Después empaquetamos tanto el archivo `web.xml`, como las clases compiladas pertenecientes al servlet, en un fichero al que daremos el nombre de miaplicweb.war (la forma de hacer esto queda fuera del alcance de esta memoria) y lo copiamos a la carpeta `webapps` de Tomcat. Una vez que Tomcat despliegue la aplicación, cuando solicitemos la siguiente dirección del protocolo HTTP:

```
http://www.miservidorcontomcat.com/miaplicweb/miservlet
```

empezará a ejecutarse el servlet que hemos programado. Podríamos añadir otro servlet a la aplicación web, incluyendo también su descripción en el descriptor de despliegue. Si le asignáramos, por ejemplo, la subruta /miotroservlet, podríamos ponerlo en marcha enviando una petición a

```
http://www.miservidorcontomcat.com/miaplicweb/miotroservlet
```

En el descriptor de despliegue de una aplicación web podemos incluir también líneas para establecer el valor de parámetros de inicialización.

Tomcat puede configurarse para que, cada vez que reciba una llamada para ejecutar un servlet, ejecute previamente otra aplicación asociada al servlet. A dicha aplicación que se ejecuta como paso previo a la entrada en acción del servlet se la denomina **filtro de servlet**.

Un filtro puede tener diversos usos como puede ser almacenar un "log" de todas las llamadas al servlet al que filtra, comprimir / descomprimir y/o cifrar / descifrar los datos que pasan por él, controlar el acceso al servlet, etc. En general, podríamos decir que se comporta como un "preprocesador" del servlet.

Para desplegar un filtro también hay que incluirlo dentro del descriptor de despliegue de una aplicación web. Ahí especificaremos a qué servlet o servlets va a filtrar. Ejemplo: configuramos un filtro de servlet para que filtre al primer servlet del ejemplo anterior. En ese caso, una llamada como la que vimos antes,

```
http://www.miservidorcontomcat.com/miaplicweb/miservlet
```

pondría en marcha al filtro, en vez de al servlet. El filtro podría, entonces, ejecutar su propio código y después pasar el control al servlet propiamente dicho (o bien podría bloquear su ejecución: nada nos asegura que el servlet finalmente se ejecute, esto depende de lo que aparezca en el código del filtro).

Podríamos también asociar el filtro no sólo con un servlet, sino con la aplicación web al completo, de manera que entraría en acción siempre que escribiéramos simplemente esto:

```
http://www.miservidorcontomcat.com/miaplicweb
```

Incluso podríamos asociar al filtro con determinados tipos de archivos (por ejemplo, archivos con extensión “htm”) dentro de la aplicación web, de forma que se ejecutara cuando solicitáramos el siguiente fichero:

```
http://www.miservidorcontomcat.com/miaplicweb/mipaginaHTM.htm
```

Pero no cuando pidiéramos cualquier otro como éste:

```
http://www.miservidorcontomcat.com/miaplicweb/mipaginaPHP.php
```

3.2 Cookies y sesiones en Java

El protocolo HTTP funciona según un esquema petición-respuesta, de forma que la conexión se establece mediante la petición de un cliente y se termina tras la respuesta del servidor. Cada par petición-respuesta se gestiona de forma independiente, sin que el servidor guarde ninguna información sobre los que se han intercambiado anteriormente. Si un usuario visita una página web en dos ocasiones consecutivas, no hay, en principio, ningún mecanismo HTTP que permita a una aplicación web reconocer que se trata del mismo usuario. Esto resulta problemático en algunas ocasiones, puesto que hay aplicaciones que necesitan conservar ciertos datos del usuario, cuando éste salta de una

página a otra dentro de la misma aplicación (un ejemplo de esto es el "carrito de la compra" de una web de una tienda de libros, discos, etc.).

Para solventar este problema, los programadores de navegadores web introdujeron el concepto de ***cookie***. Una cookie es un pequeño archivo situado en el disco duro del usuario y que identifica al cliente ante una aplicación web. El servidor envía la cookie al cliente en respuesta a la petición inicial. Las siguientes peticiones de este cliente llevan consigo el envío de dicha cookie al servidor. El servidor puede identificar así que todas las peticiones provienen del mismo cliente.

Cada una de las cookies que posee un cliente se corresponde con el servidor que la ha enviado. Por razones de respeto a la privacidad del usuario no se permite a un cliente enviar una cookie de un servidor a otro que no sea su emisor original.

Java permite la creación y manipulación de cookies mediante el uso de la clase `javax.servlet.http.Cookie`. Sin embargo, no es una alternativa cómoda cuando la cantidad de datos a manejar es considerable. En estos casos, la API de Java para servlets nos proporciona una solución más elegante: el objeto `HttpSession`.

Mediante el uso de un servlet y la asociación del mismo con un objeto `HttpSession`, el servidor puede reconocer si ese cliente ha visitado dicho sitio web en algún otro momento. Internamente el servidor web utiliza cookies para determinar cuál de las sesiones activas pertenece a un servlet en particular, pero este proceso es transparente al programador.

El servlet creará el objeto sesión y almacenará datos en él. Las llamadas posteriores al servidor desde ese mismo navegador tendrán acceso a esos datos almacenados. El servlet puede, en cualquier momento, anular dicha sesión. Los objetos sesión utilizan las cookies solamente para almacenar un identificador del navegador. El resto de los datos de la sesión se guarda en el servidor.

Java es capaz de posibilitar este funcionamiento incluso si el usuario deshabilita las cookies en su navegador. Para ello utiliza la técnica denominada **reescritura de URL**: una vez creada la sesión, en todas las llamadas a páginas del servidor se añadirán los

datos de la sesión como parámetros. Veamos un ejemplo simplificado: supongamos que queremos mantener dos datos pertenecientes a la sesión, ID1 e ID2, los cuales tienen como valores "valor1" y "valor2", respectivamente. En este caso, las llamadas tendrían un aspecto parecido a éste:

```
http://www.miservidorweb.com?ID1=valor1&ID2=valor2
```

El hecho de que Java mantenga la sesión utilizando cookies o mediante reescritura URL es transparente al usuario.

Los objetos sesión de Java pueden utilizarse para aplicaciones web single sign-on, en las que al usuario se le exige identificación sólo en su primera visita a un servidor.

3.3 LDAP

LDAP (*Lightweight Directory Access Protocol*) es un servicio de directorio ordenado y distribuido para buscar información diversa en un entorno de red. Un servidor LDAP puede considerarse una base de datos a la que pueden dirigirse consultas. Habitualmente almacena la información de *login* (usuario y contraseña) y es utilizado para autenticarse, aunque es posible almacenar otra información, como datos de contacto del usuario, ubicación de diversos recursos de la red, certificados, etc.). A esta información se accede utilizando un protocolo normalizado.

Existen diversas implementaciones del protocolo LDAP:

- **Active Directory:** es propiedad de Microsoft. Define un esquema (definición de los campos que pueden ser consultados) para los datos almacenados.

- **Novell Directory Services:** conocido también como eDirectory, es la implementación de Novell para manejar el acceso a recursos en diferentes servidores y computadoras de una red. La ventaja de esta implementación es que corre en diversas plataformas, por lo que puede adaptarse fácilmente a entornos que utilicen más de un sistema operativo.

- **iPlanet:** desarrollado por AOL y Sun Microsystems y basado en la antigua implementación de Netscape.

- **OpenLDAP:** se trata de una implementación del protocolo de código abierto y que soporta múltiples esquemas.

3.4 SOAP

SOAP (siglas de *Simple Object Access Protocol*) es un protocolo estándar creado por Microsoft, IBM y otros, y actualmente bajo el auspicio de la W3C, que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML. SOAP es uno de los protocolos utilizados en los servicios Web.

A diferencia de DCOM y CORBA, que son binarios, los mensajes SOAP se componen en formato XML, que facilita la eliminación de errores, pero es menos efectivo. SOAP es un marco extensible y descentralizado que permite trabajar sobre múltiples pilas de protocolos. Las llamadas a métodos remotos pueden ser modeladas en forma de varios mensajes SOAP interactuando entre sí.

SOAP corre generalmente sobre HTTP, que es el único protocolo utilizado en Internet homologado por el W3C. SOAP tiene como base XML y la estructura de sus mensajes sigue el patrón *Header-Body*. La cabecera *Header* es opcional y contiene metadatos sobre enrutamiento (*routing*), seguridad o transacciones. El desarrollo *Body* contiene la información principal, que se conoce como carga útil (*payload*). La carga útil se acoge a un XML Schema propio.

HTTP fue elegido como protocolo de transporte primario por sus ventajas, para lidiar con *firewalls*, por ejemplo. Otros protocolos como GIOP/IIOP o DCOM suelen ser repelidos por estos *firewalls*.