Introducción a los servlets

Índice

1. Conceptos básicos de servlets

1.1. Concepto de servlet

Un **servlet** es un programa Java que se ejecuta en un servidor Web y construye o sirve páginas web. De esta forma se pueden construir páginas dinámicas, basadas en diferentes fuentes variables: datos proporcionados por el usuario, fuentes de información variable (páginas de noticias, por ejemplo), o programas que extraigan información de bases de datos.

Comparado con un CGI, un servlet es más sencillo de utilizar, más eficiente (se arranca un hilo por cada petición y no un proceso entero), más potente y portable. Con los servlets podremos, entre otras cosas, procesar, sincronizar y coordinar múltiples peticiones de clientes, reenviar peticiones a otros servlets o a otros servidores, etc.

1.2. Recursos de servlets y JSP

Normalmente al hablar de servlets se habla de JSP y viceversa, puesto que ambos conceptos están muy interrelacionados. Para trabajar con ellos se necesitan tener presentes algunos recursos:

- Un **servidor web** que dé soporte a servlets / JSP (contenedor de servlets y páginas JSP). Ejemplos de estos servidores son Apache Tomcat, Resin, JRun, Java Web Server, BEA WebLogic, etc.
- Las **librerías** (clases) necesarias para trabajar con servlets / JSP. Normalmente vienen en ficheros JAR en un directorio `lib` del servidor (`common/lib` en Tomcat): `servlet-api.jar` (con la API de servlets), y `jsp-api.jar`, para JSP. Al desarrollar nuestra aplicación, deberemos incluir las librerías necesarias en el classpath para que compilen los ficheros (sólo necesitaremos compilar los servlets, no los JSP). También se puede utilizar el fichero JAR `j2ee.jar` que viene con Java Enterprise Edition, pero no es recomendable si se puede disponer de las librerías específicas del servidor.
- La **documentación** sobre la API de servlets / JSP (no necesaria, pero sí recomendable)

Para encontrar información sobre servlets y JSP, son de utilidad las siguientes direcciones:

- http://java.sun.com/j2ee/: referencia de todos los elementos que componen J2EE
- http://java.sun.com/products/jsp: referencia para las últimas actualizaciones en JSP
- http://java.sun.com/products/servlets: referencia para las últimas actualizaciones en servlets

1.3. Arquitectura del paquete servlet

Dentro del paquete **`javax.servlet`** tenemos toda la infraestructura para poder trabajar con servlets. El elemento central es la interfaz **`Servlet`**, que define los métodos para cualquier servlet. La clase **`GenericServlet`** es una clase abstracta que implementa dicha interfaz para un servlet genérico, independiente del protocolo. Para definir un servlet que se utilice vía web, se tiene la clase **`HttpServlet`** dentro del subpaquete **`javax.servlet.http`**. Esta clase hereda de `GenericServlet`, y también es una clase abstracta, de la que heredaremos para construir los servlets para nuestras aplicaciones web.

Cuando un servlet acepta una petición de un cliente, se reciben dos objetos:

- Un objeto de tipo **`ServletRequest`** que contiene los datos de la petición del usuario (toda la información entrante). Con esto se accede a los parámetros pasados por el cliente, el protocolo empleado, etc. Se puede obtener también un objeto **`ServletInputStream`** para obtener datos del cliente que realiza la petición. La subclase **`HttpServletRequest`** procesa peticiones de tipo HTTP.
- Un objeto de tipo **`ServletResponse`** que contiene (o contendrá) la respuesta del servlet ante la petición (toda la información saliente). Se puede obtener un objeto **`ServletOutputStream`**, y un `Writer`, para poder escribir la respuesta. La clase **HttpServletResponse** se emplea para respuestas a peticiones HTTP.

1.4. Ciclo de vida de un servlet

Todos los servlets tienen el mismo ciclo de vida:

- Un servidor carga e inicializa el servlet
- El servlet procesa cero o más peticiones de clientes (por cada petición se lanza un hilo)
- El servidor destruye el servlet (en un momento dado o cuando se apaga)

1. Inicialización

En cuanto a la inicialización de un servlet, se tiene una por defecto en el método **`init()`**.

```
public void init() throws ServletException
{
        ...
}

public void init(ServletConfig conf) throws ServletException
{
        super.init(conf);
        ...
}
```

El primer método se utiliza si el servlet no necesita parámetros de configuración externos. El segundo se emplea para tomar dichos parámetros del objeto `ServletConfig` que se le pasa. La llamada a `super.init(...)` al principio del método es MUY importante, porque el servlet utiliza esta configuración en otras zonas.

Si queremos definir nuestra propia inicialización, deberemos sobreescribir alguno de estos métodos. Si ocurre algún error al inicializar y el servlet no es capaz de atender peticiones, debemos lanzar una excepción de tipo `UnavailableException`.

Podemos utilizar la inicialización para establecer una conexión con una base de datos (si trabajamos con base de datos), abrir ficheros, o cualquier tarea que se necesite hacer una sola vez antes de que el servlet comience a funcionar.

2. Procesamiento de peticiones

Una vez inicializado, cada petición de usuario lanza un hilo que llama al método **`service()`** del servlet.

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
throws ServletException, IOException
```

Este método obtiene el tipo de petición que se ha realizado (GET, POST, PUT, DELETE). Dependiendo del tipo de petición que se tenga, se llama luego a uno de los métodos:

- **doGet()**:

  ```
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws ServletException, IOException
  ```

  Para peticiones de tipo GET (aquellas realizadas al escribir una dirección en un navegador, pinchar un enlace o rellenar un formulario que no tenga METHOD=POST)

- **doPost()**:

  ```
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
  throws ServletException, IOException
  ```

  Para peticiones POST (aquellas realizadas al rellenar un formulario que tenga METHOD=POST)

- **doXXX()**: normalmente sólo se emplean los dos métodos anteriores, pero se tienen otros métodos para peticiones de tipo DELETE (**doDelete()**), PUT (**doPut()**), OPTIONS (**doOptions()**) y TRACE (**doTrace()**).

3. Destrucción

El método **`destroy()`** de los servlets se emplea para eliminar un servlet y sus recursos asociados.

```
public void destroy() throws ServletException
```

Aquí debe deshacerse cualquier elemento que se construyó en la inicialización (cerrar conexiones con bases de datos, cerrar ficheros, etc).

El servidor llama a `destroy()` cuando todas las llamadas de servicios del servlet han concluido, o cuando haya pasado un determinado número de segundos (lo que ocurra primero). Si esperamos que el servlet haga tareas que requieran mucho tiempo, tenemos que asegurarnos de que dichas tareas se completarán. Podemos hacer lo siguiente:

- Definir un contador de tareas activas, que se incremente cada vez que una tarea comienza (entendemos por *tarea* cada petición que se realice al servlet), y se decremente cada vez que una termina. Podemos utilizar bloques de código `synchronized` para evitar problemas de concurrencia.
- Hacer que el método `destroy()` no termine hasta que lo hagan todas las tareas pendientes (comprobando el contador de tareas pendientes)
- Hacer que las tareas pendientes terminen su trabajo si se quiere cerrar el servlet (comprobando algún flag que indique si el servlet se va a cerrar o no).

1.5. Estructura básica de un servlet

La plantilla común para implementar un servlet es:

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

public class ClaseServlet extends HttpServlet
{
        public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException
        {
                // ... codigo para una peticion GET
        }

        public void doPost(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, IOException

        {
                // ... codigo para una peticion POST
        }
}
```

El servlet hereda de la clase `HttpServlet`. Normalmente se deben sobreescribir los métodos `doGet()`, `doPost()` o ambos, colocando el código que queremos que se ejecute cuando se reciba una petición GET o POST, respectivamente. Conviene definir los dos para distinguir ambas peticiones. En caso de que queramos hacer lo mismo para GET o POST, definimos el código en uno de ellos, y hacemos que el otro lo llame.

Aparte de estos métodos, podemos utilizar otros de los que hemos visto: `init()` (para inicializaciones), `doXXX()` (para tratar otros tipos de peticiones (PUT, DELETE, etc)), **destroy()** (para finalizar el servlet), etc, así como nuestros propios métodos internos de la clase.

2. Configuración de servlets en aplicaciones web

Para instalar un servlet en una aplicación web, se coloca la clase compilada del servlet dentro del directorio `WEB-INF/classes` de la aplicación (respetando también la estructura de paquetes, creando tantos subdirectorios como sea necesario).

De forma alternativa, también podríamos empaquetar nuestros servlets en un JAR y poner esta librería de clases dentro del directorio `WEB-INF/lib`. De cualquiera de las dos formas la clase del servlet estará localizable para la aplicación. Veremos ahora las formas que tenemos de invocar a ese servlet.

2.1. Mapeo de servlets en el fichero descriptor

Los servlets se invocarán cuando desde un cliente hagamos una petición a una URL determinada. Para especificar la URL a la que está asociada cada servlet, deberemos configurar dicho servlet en el fichero descriptor de despliegue (`web.xml`).

En primer lugar deberemos introducir una marca **`<servlet>`** para declarar cada servlet de la siguiente forma:

```
<servlet>
        <servlet-name>nombre</servlet-name>
        <servlet-class>unpaquete.ClaseServlet</servlet-class>
</servlet>
```

Donde **`<servlet-name>`** es un nombre identificativo y arbitrario del servlet, y **`<servlet-class>`** es la clase del servlet.

> **Nota**
>
> Al declarar un servlet, debe indicarse el nombre completo de la clase en la que está implementado, incluyendo paquetes y subpaquetes. Esto es así porque en el `web.xml` no tenemos ningún "import" que nos permita desambiguar entre posibles diferentes clases con el mismo nombre.

Una vez declarado nuestro servlet, deberemos mapearlo a una URL. Esto se consigue mediante las etiquetas <**servlet-mapping**>:

```
<servlet-mapping>
        <servlet-name>nombre</servlet-name>
        <url-pattern>/ejemploservlet</url-pattern>
</servlet-mapping>
```

En la subetiqueta **`<servlet-name>`** se pone el nombre del servlet al que se quiere asignar la URL (será uno de los nombres dados en alguna etiqueta `<servlet>` previa), y en **`<url-pattern>`** colocamos la URL que le asignamos al servlet, que debe comenzar con '/'.

Nota

Destacamos que primero se colocan todas las etiquetas `<servlet>`, y luego las `<servlet-mapping>` que se requieran. En las actuales versiones de los servidores web el orden es indiferente, pero si queremos garantizar la compatibilidad con versiones anteriores, deberemos respetarlo.

Así, con lo anterior, podremos llamar al servlet identificado con `nombre` accediendo a la URL a la que se encuentra mapeado:

```
http://localhost:8080/<dir>/ejemploservlet
```

siendo `<dir>` el directorio en el que se encuentra el contexto de nuestra aplicación Web.

También podemos asignar en `<url-pattern>` expresiones como:

```
<servlet-mapping>
        <servlet-name>nombre</servlet-name>
        <url-pattern>/ejemploservlet/*</url-pattern>
</servlet-mapping>
```

o como:

```
<servlet-mapping>
        <servlet-name>nombre</servlet-name>
        <url-pattern>/ejemploservlet/*.do</url-pattern>
</servlet-mapping>
```

Con el primero, cualquier URL del directorio de nuestra aplicación Web que comience con `/ejemploservlet/` se redirigirá y llamará al servlet identificado con `nombre`. Por ejemplo, las direcciones:

```
http://localhost:8080/<dir>/ejemploservlet/unapagina.html
http://localhost:8080/<dir>/ejemploservlet/login.do
```

acabarían llamando al servlet `nombre`.

Con el segundo, cualquier llamada a un recurso acabado en `.do` del directorio `/ejemploservlet/` de nuestra aplicación se redirigiría al servlet `nombre`. Podemos hacer que distintas URLs llamen a un mismo servlet, sin más que añadir varios grupos `<servlet-mapping>`, uno por cada patrón de URL diferente, y todos con el mismo `<servlet-name>`.

Este mismo procedimiento se puede aplicar también si en lugar de un servlet queremos tratar una página JSP. Para declarar una página **JSP**, sustituiremos la etiqueta `<servlet-class>` por la etiqueta **`<jsp-file>`**:

```
<servlet>
        <servlet-name>nombre2</servlet-name>
        <jsp-file>/mipagina.jsp</jsp-file>
</servlet>
```

Esta página se podrá mapear a diferentes URLs de la misma forma en la que lo hacemos para un servlet.

2.2. Servlet invoker

Hemos visto la forma en la que los servlets se mapean a la URL a la que deberemos acceder para invocarlos. Sin embargo, en algunos casos en los que estemos haciendo pruebas rápidas podría venirnos bien poder invocar un servlet sin tener que declararlo en el descriptor de despliegue. Existe en los servidores web un servlet llamado `invoker` que nos permite hacer esto. Este servlet será quien se encargue de invocar otros servlets sin tener que ser configurados.

Para poder utilizar esta característica deberemos habilitar este servlet `invoker`. Para ello deberemos editar el fichero `web.xml` general de Tomcat (se encuentra en el directorio `$TOMCAT_HOME/conf/web.xml`), y descomentar en él tanto la declaración del servlet `invoker`, como su mapeo a una URL. La declaración será como se muestra a continuación:

```
<servlet>
  <servlet-name>invoker</servlet-name>
  <servlet-class>
    org.apache.catalina.servlets.InvokerServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
```

Por otro lado, el mapeo por defecto se realiza de la siguiente forma:

```
<servlet-mapping>
  <servlet-name>invoker</servlet-name>
  <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

Podemos fijarnos en que este servlet se encuentra mapeado por defecto a la URL `/servlet/*` (aunque podríamos cambiarla). Es decir, este servlet interceptará todas las peticiones a nuestra aplicación web que comiencen por `/servlet`. Lo que se indique a continuación en la URL, será considerado el nombre del servlet que queremos invocar.

```
http://localhost:8080/<dir>/servlet/<nombre-clase-servlet>
```

Nota

A partir de Tomcat 6.0 sólo se puede utilizar el servlet `invoker` en contextos "privilegiados". Para hacer esto deberemos editar también el fichero `$TOMCAT_HOME/conf/context.xml` y sustituir la etiqueta `<Context>` por `<Context privileged="true">`

Por ejemplo, podríamos invocar nuestro servlet con la siguiente URL (recordamos que deberemos siempre proporcionar el nombre completo de la clase, incluyendo el paquete al que pertenezca):

```
http://localhost:8080/<dir>/servlet/unpaquete.ClaseServlet
```

Notar que `servlet` no corresponde a ningún directorio en nuestra aplicación, sino que se trata de la ruta a la que está mapeado el servlet `invoker`.

Si hemos declarado nuestro servlet en el descriptor de despliegue, también podremos utilizar el servlet `invoker` para invocar nuestro servlet mediante su nombre, en lugar de utilizar el nombre de la clase en la que se encuentra implementado:

```
http://localhost:8080/<dir>/servlet/nombre
```

Incluso aunque hayamos mapeado nuestro servlet a una URL determinada, si tenemos habilitado el servlet `invoker` podremos también utilizarlo de forma alternativa para invocar nuestro servlet, bien mediante su nombre, o el nombre de su clase. En este caso podremos acceder al servlet de tres formas:

```
http://localhost:8080/<dir>/servlet/nombre
http://localhost:8080/<dir>/servlet/unpaquete.ClaseServlet
http://localhost:8080/<dir>/ejemploservlet
```

Importante

Utilizar el servlet `invoker` puede ocasionarnos graves problemas de seguridad, al dejar una puerta abierta para ejecutar cualquier servlet de nuestra aplicación. Por lo tanto, se desaprueba totalmente su uso, siendo interesante únicamente para hacer pruebas rápidas.

2.3. Asignar parámetros de inicio a un servlet o página JSP

El hecho de asignar un nombre a un servlet o página JSP mediante la etiqueta `<servlet>` y sus subetiquetas nos permite identificarlo con ese nombre, y también poderle asignar parámetros de inicio. Para asignar parámetros se colocan etiquetas **`<init-param>`** dentro de la etiqueta `<servlet>` del servlet o página JSP al que le queremos asignar parámetros. Dichas etiquetas tienen como subetiquetas un **`<param-name>`** (con el nombre del parámetro) y un **`<param-value>`** (con el valor del parámetro). Por ejemplo:

```
<servlet>
        <servlet-name>nombre</servlet-name>
        <servlet-class>ClaseServlet</servlet-class>
        <init-param>
                <param-name>param1</param-name>
                <param-value>valor1</param-value>
        </init-param>
        <init-param>
                <param-name>param2</param-name>
                <param-value>valor2</param-value>
        </init-param>
```

```
</servlet>
```

Para obtener luego los parámetros desde el servlet se utiliza **`getServletConfig().getInitParameter(nombre)`** donde `nombre` es el valor `<param-name>` del parámetro que se busca, y devuelve el valor (elemento `<param-value>` asociado), que es de tipo `String` siempre. Para obtener estos valores desde páginas JSP se emplean otros métodos.

Los parámetros de inicio sólo se aplican cuando accedemos al servlet o página JSP a través del nombre asignado en `<servlet-name>`, o a través de la URL asociada en un `<servlet-mapping>`.

2.4. Cargar servlets al inicio

A veces nos puede interesar que un servlet se cargue al arrancar el servidor, y no con la primera petición de un cliente. Para hacer eso, incluimos una etiqueta **`<load-on-startup>`** dentro de la etiqueta `<servlet>`. Dicha etiqueta puede estar vacía:

```
<servlet>
        <servlet-name>nombre</servlet-name>
        <servlet-class>ClaseServlet</servlet-class>
        <load-on-startup/>
</servlet>
```

o contener un número:

```
<servlet>
        <servlet-name>nombre</servlet-name>
        <servlet-class>ClaseServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
</servlet>
```

que indica el orden en que el servidor irá cargando los servlets (de menor a mayor valor).

3. Ejemplos básicos de servlets

3.1. Servlet que genera texto plano

El siguiente ejemplo de servlet muestra una página con un mensaje de saludo: "Este es un servlet de prueba". Lo cargamos mediante petición GET.

```
package ejemplos;

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

public class ClaseServlet extends HttpServlet
{
```

```
        public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException
        {
                PrintWriter out = response.getWriter();
                out.println ("Este es un servlet de prueba");
        }
}
```

Se obtiene un `Writer` para poder enviar datos al usuario. Simplemente se le envía la cadena que se mostrará en la página generada.

3.2. Servlet que genera una página HTML

Este otro ejemplo escribe código HTML para mostrar una página web.

```
package ejemplos;

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

public class ClaseServletHTML extends HttpServlet
{
        public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException
        {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println ("<!DOCTYPE HTML PUBLIC \""+
                             "-//W3C//DTD HTML 4.0 " +
                             "Transitional//EN\">");
                out.println ("<HTML>");
                out.println ("<BODY>");
                out.println ("<h1>Titulo</h1>");
                out.println ("<br>Servlet que genera HTML");
                out.println ("</BODY>");
                out.println ("</HTML>");
        }
}
```

Para generar una página HTML con un servlet debemos seguir dos pasos:

- Indicar que el contenido que se va a enviar es HTML (mediante el método `setContentType()` de `HttpServletResponse`):

  ```
  response.setContentType("text/html");
  ```

  Esta línea es una cabecera de respuesta, que veremos más adelante cómo utilizar. Hay que ponerla antes de obtener el `Writer`.
- Escribir en el flujo de salida el texto necesario para generar la página HTML. La línea que genera el DOCTYPE no es necesaria, aunque sí muy recomendada para que se sepa qué versión de HTML se está empleando.

3.3. Servlet que utiliza parámetros de inicialización

Este otro ejemplo utiliza dos parámetros de inicialización externos:

```
package ejemplos;

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

public class ClaseServletInit extends HttpServlet
{
        // Mensaje que se va a mostrar en la pagina
        String mensaje = "";
        // Numero de veces que se va a repetir el mensaje
        int contador = 1;

        // Metodo de inicializacion

        public void init(ServletConfig conf)
        throws ServletException
        {
                super.init(conf); // MUY IMPORTANTE

                mensaje = conf.getInitParameter("mensaje");
                if (mensaje == null)
                        mensaje = "Hola";

                try
                {
                        contador = Integer.parseInt(
                           conf.getInitParameter("contador"));
                } catch (NumberFormatException e) {
                        contador = 1;
                }
        }

        // Metodo para procesar una peticion GET

        public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException
        {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println ("<!DOCTYPE HTML PUBLIC \""+
                             "-//W3C//DTD HTML 4.0 " +
                             "Transitional//EN\">");
                out.println ("<HTML>");
                out.println ("<BODY>");

                for (int i = 0; i < contador; i++)
                {
                        out.println (mensaje);
                        out.println ("<BR>");
                }
```

```
                out.println ("</BODY>");
                out.println ("</HTML>");
        }
}
```

- Se utiliza el método **init()** con un parámetro `ServletConfig` para poder tomar los parámetros externos. Es importante la llamada a `super` al principio del método.
- Mediante el método **getInitParameter()** de `ServletConfig` obtenemos dos parámetros: `mensaje` y `contador`, que asignamos a las variables del mismo nombre. El primero indica el mensaje que se va a mostrar en la página, y el segundo el número de veces que se va a mostrar.
- En **doGet()** hacemos uso de esos parámetros obtenidos, para mostrar el mensaje las veces indicadas.

3.4. Prueba de los ejemplos

Para probar el ejemplo, tendríamos dos posibilidades

- Crear un directorio en `webapps` para el ejemplo (por ejemplo, `webapps/ejemplobasico`), y dentro de él:
  - Definir un directorio `WEB-INF`
  - Dentro de `WEB-INF` colocar un fichero `web.xml` descriptor de la aplicación. Podemos mapear los servlets en este fichero descriptor, como veremos a continuación.
  - Colocar los servlets en el directorio `WEB-INF/classes`.
- Crear un fichero WAR con la estructura de ficheros y directorios vista en el punto anterior, y copiar dicho fichero WAR en el directorio `webapps` de Tomcat

Un ejemplo de fichero `web.xml` podría ser el siguiente:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//
 DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
        <servlet>
                <servlet-name>ejemplo1_1</servlet-name>
<servlet-class>ejemplos.ClaseServlet</servlet-class>
        </servlet>
        <servlet>
                <servlet-name>ejemplo1_2</servlet-name>
<servlet-class>ejemplos.ClaseServletHTML</servlet-class>
        </servlet>
        <servlet>
                <servlet-name>ejemplo1_3</servlet-name>
<servlet-class>ejemplos.ClaseServletInit</servlet-class>
                <init-param>
                        <param-name>
                        mensaje
                        </param-name>
                        <param-value>
```

```
                        Mensaje de prueba
                        </param-value>
                </init-param>
                <init-param>
                        <param-name>contador</param-name>
                        <param-value>10</param-value>
                </init-param>
        </servlet>

        <servlet-mapping>
                <servlet-name>ejemplo1_1</servlet-name>
                <url-pattern>/ejemploservlet</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
                <servlet-name>ejemplo1_2</servlet-name>
                <url-pattern>/ejemploservletHTML</url-pattern>
        </servlet-mapping>
        <servlet-mapping>
                <servlet-name>ejemplo1_3</servlet-name>
                <url-pattern>/ejemploservletInit</url-pattern>
        </servlet-mapping>
</web-app>
```

Vemos que se mapean los tres servlets:

- **ejemplo1_1** es el nombre para el primer servlet `ClaseServlet`, que se puede referenciar también con `/ejemploservlet`.
- **ejemplo1_2** es el nombre para el segundo servlet `ClaseServletHTML`, que se puede referenciar también con `/ejemploservletHTML`
- **ejemplo1_3** es el nombre para el tercer servlet `ClaseServletInit`, que se puede referenciar también con `/ejemploservletInit`. En él se definen los parámetros de inicio `mensaje` y `contador`, utilizados internamente por el servlet (se definen mediante etiquetas `<init-param>`, indicando el nombre (`param-name`) y el valor (`param-value`)).

Para probar los servlets, instalamos la aplicación web en Tomcat. Para llamar a los servlets, podemos hacerlo de tres formas:

- Escribiendo, respectivamente para cada servlet:

```
http://localhost:8080/appserv1/servlet/ejemplos.ClaseServlet
http://localhost:8080/appserv1/servlet/ejemplos.ClaseServletHTML
http://localhost:8080/appserv1/servlet/ejemplos.ClaseServletInit
```

- Sustituir la clase del servlet por su nombre asociado en el mapeo:

```
http://localhost:8080/appserv1/servlet/ejemplo1_1
http://localhost:8080/appserv1/servlet/ejemplo1_2
http://localhost:8080/appserv1/servlet/ejemplo1_3
```

- Utilizar el mapeo en el fichero descriptor y llamar a los servlets con su URL asociada:

```
http://localhost:8080/appserv1/ejemploservlet
http://localhost:8080/appserv1/ejemploservletHTML
http://localhost:8080/appserv1/ejemploservletInit
```

NOTA: el servlet `ClaseServletInit` no tomará los parámetros si lo llamamos del primer modo, debido a que se asignan esos parámetros al mapeo.

4. Logging en aplicaciones Java EE con servlets

Utilizaremos Log4J como librería de logging, pero encapsulada dentro de la librería *commons-logging* de Jakarta. Ya vimos en el módulo de servidores web que, para poder imprimir mensajes de log en una aplicación que contenga servlets, se debían seguir estos pasos:

- Añadir los ficheros JAR de las librerías (`commons-logging-X.X.jar` y `log4j-X.X.X.jar`) en la carpeta **WEB-INF/lib** de nuestra aplicación
- Colocar dos ficheros `.properties` en el CLASSPATH de la aplicación (carpeta `WEB-INF/classes`):
  - Un fichero **commons-logging.properties** indicando que vamos a utilizar Log4J como librería subyacente
  - Un fichero **log4j.properties** con la configuración de logging para Log4J.

  Vimos que estos ficheros los colocaríamos en una carpeta fuente llamada `resources`, para luego desde Ant o con WebTools llevarlo a `build/WEB-INF/classes`.
- Finalmente, sólo queda en cada servlet o clase Java colocar los mensajes de log donde queramos. Veremos cómo hacerlo en servlets y páginas JSP en el siguiente módulo. Aquí vemos un ejemplo de cómo ponerlos en cada servlet:

```
...

import org.apache.commons.logging.*;

public class ServletLog4J1 extends HttpServlet
{
        static Log logger = LogFactory.getLog(ServletLog4J1.class);

        // Metodo para procesar una peticion GET

        public void doGet(HttpServletRequest request,
HttpServletResponse response)
        throws ServletException, IOException
        {
                logger.info("Atendiendo peticion Servlet Log4J");
                PrintWriter out = response.getWriter();
                out.println ("Servlet sencillo de prueba para
logging");
                logger.debug("Fin de procesamiento de petición");
        }
}
```