Práctica 2: Extensión de un servidor con servlets

(2 horas)

Introducción

En esta práctica veremos otra forma de realizar aplicaciones semejantes a los CGIs, los servlets. El enfoque del diseño con servlets es muy diferente al de CGIs. Los CGIs son programas que el servidor debe ejecutar. Son procesos que el servidor arranca y espera a que finalicen para obtener un resultado. Los servlets son extensiones del propio servidor.

Objetivos

Identificar las partes de un servlet.
Comparar prestaciones de los servlets y CGIs.
Utilizar servlets.
Diseñar y programar servlets.

CGIs, FastCGI y API del servidor

En la anterior práctica hemos visto cómo extender la funcionalidad de un servidor web utilizando CGI: incorporando un programa externo que responde a algunas peticiones dirigidas al servidor web. El modo de funcionamiento es el siguiente:

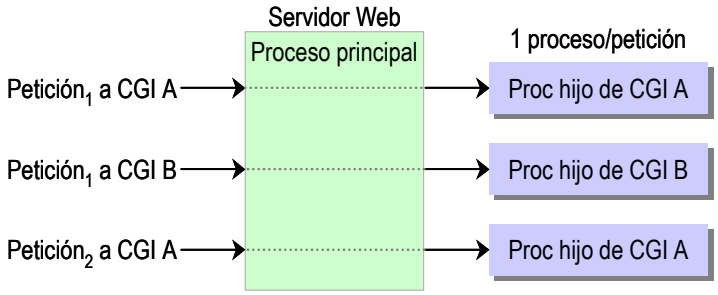

Fig. 1: un servidor web que utiliza procesos (comandos) CGI

Cada petición crea un proceso que recibe las variables de entorno junto con los datos y genera una respuesta por la salida estándar. Este procedimiento consume muchos recursos del servidor y es lento.

Para solucionar lo anterior, se creó una mejora de los CGI, los FastCGI. Hacen que un solo proceso cargado vaya sirviendo todas las peticiones sin descargarse (un proceso persistente o daemon). A veces un proceso no es suficiente y hace falta tener varios procesos a la vez atendiendo a varias peticiones que tienen lugar simultáneamente.

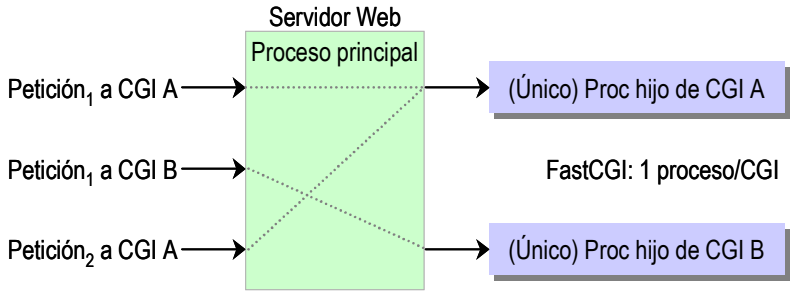

Fig. 2: un servidor que utiliza procesos persistentes FastCGI

Tanto los CGI como FastCGI no pueden interactuar con el interior del servidor (por ejemplo generar una línea en los ficheros de log del servidor).

Otra alternativa para extender el servidor de forma más eficiente consiste en utilizar las API de extensión de cada servidor (NSAPI en el servidor de Netscape/Sun, ISAPI en el servidor de Netscape, o un módulo en Apache). Las extensiones forman parte del proceso servidor y estas API ofrecen mucha más funcionalidad y control sobre el servidor web, además de velocidad al estar compiladas y formar parte del mismo ejecutable.

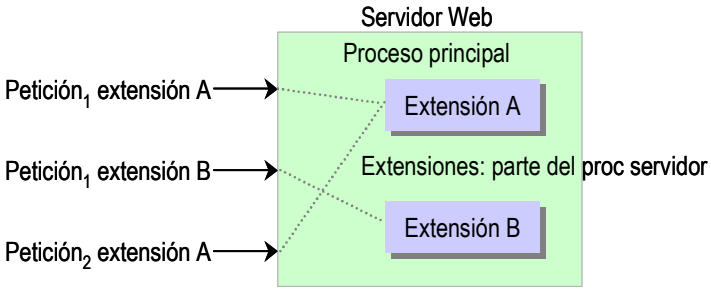

Fig. 3: un servidor extendido usando el API de extensión

Sin embargo tienen tres problemas importantes: son extensiones no portables, específicas para un único servidor; son más complejas de desarrollar y mantener; introducen riesgo en el proceso servidor: peligro en cuanto a seguridad y fiabilidad (un fallo de la extensión puede acabar con el proceso servidor de web).

Servlet Java

Un servlet es una extensión genérica del servidor: una clase java que puede cargarse dinámicamente para "extender" la funcionalidad del servidor web. En muchos casos sustituye con mejoras a los CGI. Es una extensión que corre en una Máquina Virtual Java (JVM) dentro del servidor: es seguro y transportable. Como se ejecutan en el servidor, el cliente web los invocará como si fueran CGI, y en respuesta sólo verá html, sin que el cliente tenga que tratar con Java (como en los applets, que es código Java que se ejecuta en una máquina virtual Java del cliente web).

Un servlet es un trozo de código que corre en un servidor: se pueden usar para "extender" servidores de web pero también otros servidores (ej: ftp para añadir comandos nuevos, correo para filtrar o detectar virus, etc.)

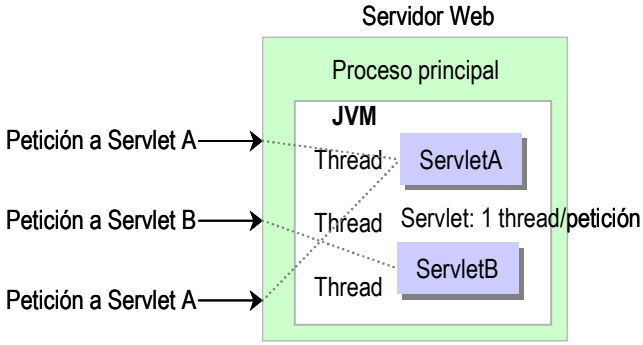

Fig. 4: un servidor extendido con servlets en su JVM

Los servlets son una extensión estándar de Java, y las clases suelen venir con las distribuciones de Java SDK (Kit de Desarrollo Java).

Para usar los servlets hace falta un "motor" donde probarlos y ponerlos en servicio. Pueden ser servidores que ya soportan servlets (por ejemplo Domino Go de Lotus, WebSite de O'Reilly, Jigsaw del Consorcio Web), o módulos que se pueden añadir a servidores que inicialmente no los soportaban (por ejemplo Jserv para Apache). En esta práctica vamos a usar el servidor tomcat, escrito en Java que directamente soporta servlets.

Las ventajas principales de los servlets son las siguientes:

- Portabilidad: usan siempre las mismas llamadas (API) y corren sobre Java, por lo que son verdaderamente portátiles entre entornos (que soporten servlets).
- Potencia: pueden usar todo el API de Java (excepto AWT), además de comunicarse con otros componentes con RMI, CORBA, usar Java Beans, conectar con bases de datos, abrir otros URL, etc …
- Eficiencia: una vez cargado queda en la memoria del servidor como una única instancia. Varias peticiones simultáneas generan varios threads sobre el servlet, que es muy eficiente. Además, por estar cargado en memoria puede mantener su estado y mantener conexiones con recursos externos como bases de datos que pueden requerir un cierto tiempo para conectar.
- Seguridad: además de la seguridad que introduce el lenguaje Java (gestión de memoria automática, ausencia de punteros, tratamiento de excepciones), el gestor de seguridad "security manager" puede evitar servlets malintencionados o mal escritos que podrían dañar al servidor web.
- Integración con el servidor: pueden cooperar con el servidor en formas que los CGI no pueden, como cambiar el path del url, poner líneas de log en el servidor, comprobar la autorización, asociar tipos MIME a los objetos o incluso añadir usuarios y permisos al servidor.

El API de Servlets

Los servlets usan clases e interfaces de dos paquetes: javax.servlet que contiene clases para servlets genéricos (independientes del protocolo que usen) y javax.servlet.http (que añade funcionalidad particular de http). El nombre javax indica que los servlets son una extensión.

Los servlets no tienen el método `main()` como los programas Java, sino que se invocan unos métodos cuando se reciben peticiones. Cada vez que el servidor pasa una petición a un servlet se invoca el método `service()` que habrá que rescribir (override). Este método acepta dos parámetros: un objeto petición (request) y un objeto respuesta.

Los servlets http, que son los que vamos a usar, tienen ya definido un método `service()` que no hace falta redefinir y que llama a doXxx(), con Xxx el nombre de la orden que viene en la petición al servidor web: `doGet()`, `doPost()`, `doHead()`, etc…

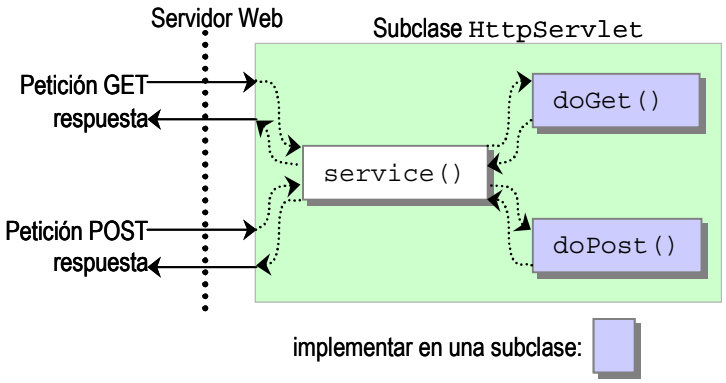

Fig. 5: Un servlet http que trata peticiones GET y POST.

Tareas

Instalación del servidor y prueba del primer servlet

A continuación veremos el código de un servlet http mínimo que genera una página HTML que dice "¡Hola Amigos!" cada vez que se invoca en el cliente web.

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

public class hola extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                   throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<html><big>¡Hola Amigos !</big></html>");
  }
```

```
}
```

El servidor extiende la clase `HttpServlet` y rescribe el método `doGet()`. Cada vez que el servidor web recibe una petición GET para este servlet, el servidor invoca su método `doGet()` pasándole un objeto con datos de la petición `HttpServletRequest` y con otro objeto `HttpServletResponse` para devolver datos en la respuesta.

El método `setContentType()` del objeto respuesta (res) establece como text/html el contenido MIME de la respuesta. El método `getWriter()` obtiene un canal de escritura que convierte los caracteres Unicode que usa Java en el juego de caracteres de la operación HTTP (normalmente iso-8859-1). Ese canal se usa para escribir el texto HTML que verá el cliente web.

Para poderlo ejecutar, hay que disponer de una máquina virtual Java (JVM), las clases de servlets, y un servidor web que soporte servlets. Vamos a usar la máquina virtual de Linux, y el servidor Tomcat. Un servidor web escrito en java que soporta servlets e incluye las clases necesarias.

Una vez comprobado que la JVM funciona (se puede ejecutar javac y comporbar que la variable CLASSPATH apunta a las clases estándar de Java, hay que copiar tomcat e instalarlo, arrancar el servidor web y probar que funciona, tal como explica la documentación que lo acompaña.

Para probar los servlets, tenemos que colocarnos en el directorio de trabajo. Si tomcat está instalado como un subdirectorio de vuestro directorio personal, vamos al directorio de trabajo con la orden:

```
cd ~/tomcat/webapps/test/Web-inf/classes
```

Tomcat implementa la especificación 2.2 de servlets. Trabaja con unidades llamadas webapps que pueden ser empaquetadas e instaladas en otros servidores. Una "webapp" es una aplicación completa que puede tener páginas web estáticas (ficheros html, gif, jpeg, etc.) y servlets. Los servlets están en el directorio Web-inf/classes. Allí para facilitar las cosas podemos dejar tanto el código fuente .java como el código compilado .class (el código fuente no se debería dejar en este lugar en un servidor real, se hace así por comodidad).

Si arrancamos un cliente web (por ejemplo netscape), y se visita el url http://localhost:8080/ nos saldrá una página que ha servido nuestro propio servidor en nuestra máquina (localhost), en el puerto tcp 8080. Si editamos el fichero hola.java con el contenido anterior. Lo compilamos con javac hola.java. Si no hay errores de compilación, tendremos en el mismo directorio el fichero hola.class. Nuestro primer servlet. **Probar que funciona visitando en el cliente web el url:** http://localhost:8080/test/servlet/hola

Formulario y servlet

Si queremos enviar al servlet datos para que los procese, podemos probar el siguiente formulario HTML (hola2.html) que colocaremos en el directorio: ~/jakarta-tomcat/webapps/test Si visitamos el url http://localhost:8080/test/hola2.html se podrá ver el formulario que corresponde al siguiente código HTML:

hola2.html:

```
<html>
<p>Dame tu nombre:</p>
<form action="http://localhost:8080/test/servlet/hola2" method=post>
<input type=text name="nombre">
y tu edad: <input type=text name="edad">
<input type=submit value="Enviar con Post">
</form>
<hr>
<form action="http://localhost:8080/test/servlet/hola2" method=get>
<input type=text name="nombre">
y tu edad: <input type=text name="edad">
<input type=submit value="Enviar con Get">
</form>
</html>
```

Antes de pulsar algún botón, tendremos que dejar en el lugar de los servlets el fichero hola2.java y compilarlo. **Una vez obtenido hola2.class probar la interacción entre un formulario html y un servlet que procesa los datos que le llegan tanto en una petición GET como POST.**

hola2.java:

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

public class hola2 extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    String nombre = req.getParameter("nombre");
    String edad = req.getParameter("edad");

    out.println("<html><big>Hola Amigo "+ nombre +
                " de " + edad + ".</big></html>");
  }
}
```

El método `doPost()` se puede implementar muy fácil llamando al método `doGet()`, ya que en este nivel no importa cómo los datos se han pasado al servidor. En general no hace falta el código siguiente y un servlet suele implementar sólo un método.

```
  public void doPost(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    doGet(req, res);
  }
```

La ventaja principal de implementar `doPost()` es que tiene la ventaja adicional de aceptar gran cantidad de datos de entrada (es una limitación de los clientes y servidores web). Como información interesante para otros servlets, todos los datos que se pasan en una petición al servidor son strings. Si se quisiera convertir la edad en un entero habría que introducir el código siguiente. No es necesario hacerlo ahora, es sólo para aclarar una duda que se os podría presentar.

```
int edadI = 0;
try {
edadI = Integer.parseInt(edad);
    } catch (NumberFormatException ignored) { }
if (edad != null) out.println("<html><big>Hola Amigo "+ nombre +
           " de " + edad + " " + edadI + ".</big></html>");
else out.println("<html>Amigo " + nombre +
                 ", falta la edad.</html>");
```

Como hemos apuntado antes, un servlet "sobrevive" a una petición. El "ciclo de vida" es:

1. Crear e inicializar el servlet
2. Procesar cero o más peticiones de clientes web
3. Destruir el servlet y "recoger sus restos" (garbage collection)

De esta manera, una vez se carga el servlet, es muy eficiente pues sólo hay una copia cargada (se ejecutan uno o varios thread), no hay que crear nuevos objetos (un solo objeto servlet), y tiene persistencia: puede guardar información entre peticiones, como contadores o conexiones a una base de datos (esto último puede ser muchísimo más eficiente que abrir y cerrar la conexión con la base de datos en cada petición).

En el ejemplo siguiente, se añade el atributo `cont`, que se va incrementando en cada petición. **Modificar el ejemplo anterior para probar el efecto de invocar varias veces el servlet desde el cliente web.**

hola3.java:

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

public class hola3 extends HttpServlet {

  int cont = 0;

  public void doGet(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    String nombre = req.getParameter("nombre");
    cont ++;
```

```
    out.println("<html><big>Hola Amigo "+ nombre + "</big><br>"+
                cont + " Accesos desde su carga.</html>");
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    doGet(req, res);
  }
}
```

Tenemos una solulción al problema de guardar el estado, los CGIs no lo permiten tenemos que guardarlo nosotros mismos en ficheros o bases de datos. Pero no todo es perfecto. Tenemos el importante problema de accesos concurrentes a variables o partes del código críticas. Tendremos que usar monitores o semáforos pare gestionar estos accesos concurrentes. En Java tenemos una forma muy sencilla de trabajar con monitores `synchronized()`

```
synchronized(this) {
       cont ++;
       out.println("<html><big>Hola Amigo "+ nombre + "</big><br>"+
                    cont + " Accesos desde su carga.</html>");
}
```

Además, si el servlet crea threads, también éstas son persistentes. A continuación se usa el método `init()` y el método `destroy()` para iniciar y parar un thread de ejecución, lo que permite que el servlet vaya trabajando durante su vida a la vez que va atendiendo en otros thread a las peticiones que le llegan. En el ejemplo siguiente, el thread que inicial el método `init()` va contando segundos transcurridos desde la carga del servlet: duerme durante 1000 milisegundos e incrementa en uno el contador de segundos, dentro de un bucle infinito.

hola4.java:

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

public class hola4 extends HttpServlet implements Runnable {

  int cont = 0;
  int segs = 0;
  Date fechaInicio = new Date(); // Fecha inicio ejecución
  Thread contadorSegs;

  public void init(ServletConfig config) throws ServletException {
    super.init(config);  // siempre
    contadorSegs = new Thread(this);
    contadorSegs.setPriority(Thread.MIN_PRIORITY);
    contadorSegs.start();
  }

  public void run() {
    while (true) {
      try {
        contadorSegs.sleep(1000);
      } catch (InterruptedException ignored) { }
    segs ++;
```

```
  }
}

public void destroy() {
  contadorSegs.stop();
}
```

Modelo de ejecución alternativo.

El modelo de ejecución estándar es una sola instancia de servlet por cada servlet registrado y tantos threads como peticiones simultáneas. Hay otro modelo de ejecución: tener varias instancias de servlet procesando cada una peticiones distintas.

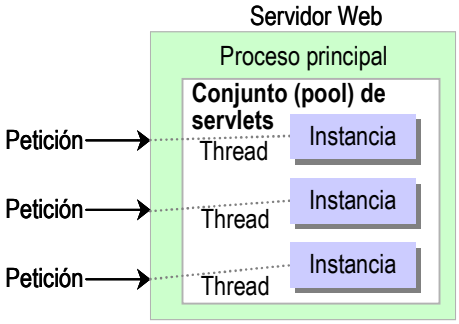

Fig. 6: El modelo de un thread por servlet.

Este modelo se activa indicando que se implementa el interface javax.servlet.SingleThreadModel. No tienen ningún requisito adicional, indicando que implementa ese interface ya actúa de esta manera. Dicho de otra manera: nunca dos threads ejecutarán el método `service()` de un servlet.

```
public class hola5 extends HttpServlet implements SingleThreadModel
```

Este modelo no tiene utilidad para un contador o aplicaciones que precisan mantener un estado común. Puede servir para evitar sincronización pero tratando peticiones de forma eficiente. Por ejemplo en una aplicación con base de datos en que cada instancia ha de ejecutar un conjunto de operaciones que forma parte de una transacción o en que cada conexión es diferente. De esta forma, cada instancia tendría una conexión distinta a la base de datos.

Diseño

Vista esta pequeña introducción a la programación de servlets, pasamos a nuestro primer diseño. Queremos de diseñéis un servlet que tengan la misma aplicación que la práctica con CGIs.

El servlet tiene que aceptar los pedidos de muebles extendiendo el método GET `doGet()`. El mismo servlet ha de listas los pedidos hechos extendiendo el método POST `doPost()`. En este caso los pedidos podemos guardarlos en variables globales dentro del propio servlet (la forma real de hacerlo es guardarlos en una base de datos).

Recordemos que los parametros son cadenas de caranteres `String`. Para poder procesar y convertir estas cadenas a otros tipos podemos usar java.lang.String

Referencias:

Jason Hunter, William Crawford. *Libro "Java Servlet Programming"*. 1ª Ed. Nov. 1998. O'Reilly. ISBN: 1-56592-391-X

The Apache Software Foundation. *Tomcat Documentation*. [en línea] versión 3.2 <http://jakarta.apache.org/tomcat/jakarta-tomcat/src/doc/index.html> [Consulta: 5 marzo 2001]

Sun Microsystems, Inc. *Java (TM) Servlet Technology*. [en línea] 23 febrero 2001 <http://java.sun.com/products/servlet/index.html> [Consulta: 5 marzo 2001]

Sun Microsystems, Inc. *JavaTM 2 Platform, Standard Edition, API Specification.* [en línea] v1.2.2 < http://java.sun.com/products/jdk/1.2/docs/api/overview-summary.html> [Consulta: 5 marzo 2001]

Dan Connolly. *CGI: Common Gateway Interface* [en línia] 13 octubre 1999 <http://www.w3.org/CGI/> [Consulta: 19 febrer 2001]

Rob Saccoccio. *FastCGI* [en línea] 25 noviembre 2001http://www.fastcgi.com/ [Consulta: 5 marzo 2001]