Extensió d'un servidor amb servlets

Introducció

Aquesta pràctica mostrarà una altra forma de realitzar aplicacions semblants als CGI, els servlets. L'enfocament del disseny amb servlets és molt diferent al dels CGI. Els CGI són programes on el servidor els ha d'executar, són procediments que el servidor arranca i espera que finalitzin per obtenir un resultat. Per contra, els servlets són extensions del propi servidor.

Objetius

Identificar les parts d'un servlet.

Comparar prestacions dels servlets y CGIs.

Utilizar servlets.

Dissenyar i programar servlets.

CGIs, FastCGI i API del servidor

A la pràctica anterior s'ha vist com extendre la funcionalitat d'un servidor web utilitzant CGI: incorporant un programa extern que respongui a algunes peticions dirigides al servidor web. El mode de funcionament és el següent:

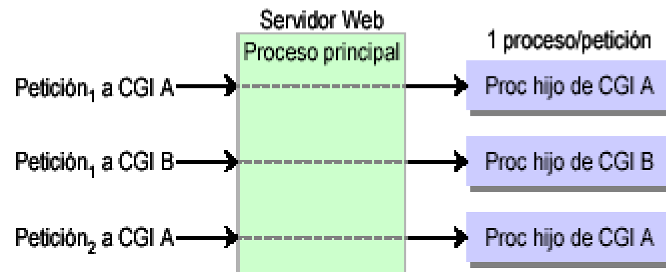

Figura 1.- un servidor web que utilitza processos (comandes) CGI

Cada petició crea un procés que rep les variables d'entorn juntament amb les dades i genera una resposta per la sortida estàndard. Aquest procediment consumeix molts recursos del servidor i és lent (cada cop el procés s'ha de carregar en memòria).

Per solucionar aquest problema es va crear una millora dels CGI, els FastCGI. Que fan que un sol procés carregat vagi servint totes les peticions sense sortir de la memòria (procés persistent o *daemon*). De cops un procés no és suficient i fa falta tenir-ne varios, per així ser capaços d'atendre diverses peticions de forma simultània.

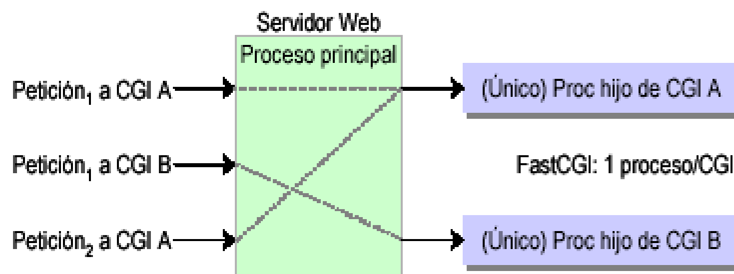

Figura 2.- un servidor que utilitza processos persistents FastCGI

Tant els CGI com els FastCGI no poden interactuar amb l'interior del servidor (per exemple generar una línia als fitxers de LOG del servidor).

Una altra alternativa per extendre el servidor de forma més eficient, consisteix en utilitzar les API d'extensió de cada servidor (NSAPI trobada al servidor de Netscape/Sun, ISAPI al servidor de Netscape, o un mòdul d'Apache). Les extensions formen part del procés servidor i ofereixen una API amb molta més funcionalitat i control sobre el servidor. A més, aporten la velocitat d'estar compilades i formar part del mateix executable.

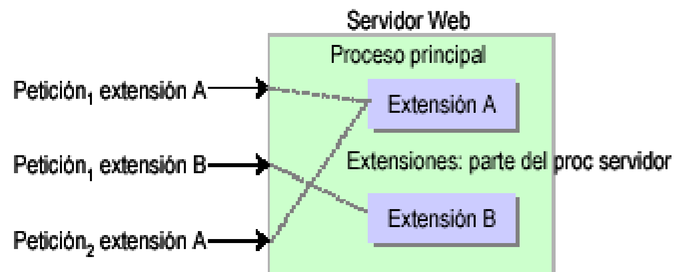

Figura 3.- un servidor extés utilitzant l'API d'extensió

De totes formes, aquesta aproximació té tres problemes importants: les extensions no són portables (entre sistemes i són específiques a cada servidor), són més complexes de desenvolupar i mantenir, introdueixen el risc al procés servidor: perill en quan a seguretat i fiabilitat (una fallada a l'extensió pot acabar amb el procés servidor de web).

Servlet Java

Un servlet és una extensió genèrica del servidor: una classe java que pot carregar-se dinàmicament per "extendre" la funcionalitat del servidor web. En molts casos substitueix amb millores als CGI. És una extensió que corre a una màquina virtual de java (JVM) dins del servidor: això li dóna seguretat i transportabilitat.

Donat que s'executen al servidor, el client web els invocarà com si fossin CGI, i com a resposta només és veurà un HTML, de forma transparent i sense que el client hagi de tractar amb Java (com passa amb aplets, que és codi Java que s'executa a una JVM del client web).

Un servlet és un tros de codi que corre a un servidor: es poden utilitzar per "extendre" servidors web, com ja s'ha comentat, però també es poden utilitzar per extendre altres servidors (p.e. FTP per afegir comandes noves, correu per filtrar i detectar virus...)

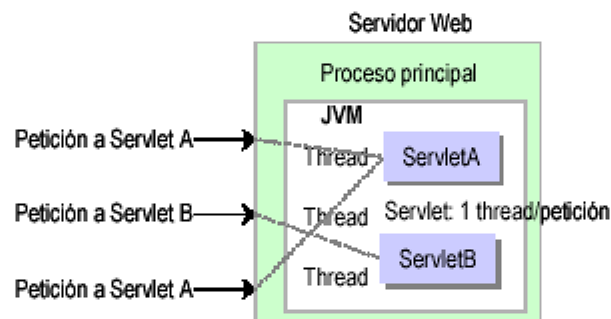

Figura 4.- un servidor extés amb servlets a la seva JVM

Els servlets són una extensió estàndard de Java, les classe solen venir amb les distribucions del Java SDK (Kit de desenvolupament).

Per utilitzar els servlets fa falta un “motor” on provar-los i posar-los en servei. Poden ser servidors que ja suporten servlets (per exemple Domino Go de Lotus, WebSite de O’Reilly, Jigsaw del Consorci Web), o connectors que es poden afegir a servidors que inicialment no els suportaven (per exemple tomcat JK per a Apache o IIS). Aquesta pràctica utilitza el servidor Tomcat, que, escrit en java, ja suporta directament els servlets.

Els principals avantatges dels servlets són els següents:

- *Portabilitat:* utilitzen sempre les mateixes crides (API) i corren sobre Java, pel que són veritablement portables entre diferents entorns (sempre que els servles estiguin suportats)
- *Potència:* poden usar tot tipus d’API de Java (excepte AWT), a més de comunicar-se amb altres components com RMI, CORBA, utilitzar Java Beans, connectar a bases de dades, obrir altres URL...
- *Eficiència:* un cop carregat queda en memòria del servidor com una única instància. Diverses peticions simultànies generen varios thread sobre el servlet, el que és molt ràpid pel fet d’estar en memòria. A més, pel mateix motiu es pot mantenir l’estat, i juntament amb això les connexions amb recursos externs com bases de dades, que normalment requereixen bastant de temps per connectar-se.
- *Seguretat:* a més de la seguretat que introdueix el llenguatge Java (gestió de memòria automàtica, falta de punters, tractament d’excepcions...), el gestor de segureat *“security manager”* pot evitar servlets malintencionats o mal escrits que podrien causar algun problema al servidor web.
- *Integració amb el servidor:* poden cooperar amb el servidor en formes que els CGI no poden, com per exemple canviar el path de la URL, posar línies de LOG al servidor, comprovar l’autorització del client, associar tipus MIME als objectes o inclús afegir usuaris i permisos al servidor.

L’API de Servlets

Els servlets utilitzen classes i interfícies de dos paquets: `javax.servlet` que conté classes per servlets genèrics (independets del protocol que utilitzen) i `javax.servlet.http` (que afegeix funcionalitat particular d’HTTP). El nom `javax` indica que els servlets són una extensió.

Els servlets no tenen mètode `main()` com els programes Java normals, per contra s’invoquen utilitzant uns mètodes quan es reben peticions. Cada cop que el servidor passa una petició a un servlet, s’invoca el mètode `service()` que s’haurà de sobrecarregar. Aquest mètode accepta dos paràmetres: un objecte petició (request) i un objecte resposta.

Els servlets HTTP, que són els que s’utilitzaran a la pràctica, tenen ja definit un mètode `service()` que no fa falta redefinir i que s’anomena `doXxx()`, on Xxx és el nom de l’ordre que ve a la petició del servidor web: `doGet()`, `doPost()`, `doHead()`, etc...

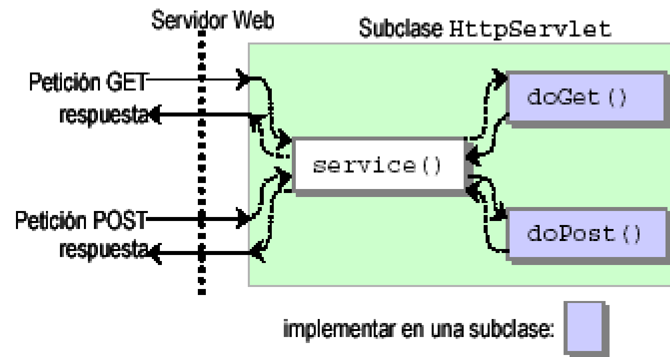

Figura 5.- Un servlet http que tracta peticions GET i POST.

Tasques

Instal·lació del servidor i prova del primer servlet

A continuació veurem el codi d'un servlet HTTP mínim que genera una pàgina HTML. La pàgina conté "Hola Amic!" cada cop que s'invoca al client 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 Amic !</big></html>");

  }

}
```

El servidor estén la classe `HttpServlet` i reescriu el mètode `doGet()`. Cada cop que el servidor web rep una petició GET per al servlet, el servidor invoca el seu mètode `doGet()` passant-li un objecte amb dades de la petició `HttpServletRequest` i amb un altre objecte `HttpServletResponse` per retornar dades amb la resposta.

El mètode `setContentType()` de l'objecte resposta (res) estableix com text/html el contingut MIME de la resposta. El mètode `getWriter()` obté un canal d'escritura que converteix els caràcters Unicode que utilitza Java amb el joc de caràcters de l'operació HTTP (normalment iso-8859-1). Aquest canal s'utilitza per escriure el text HTML que veurà el client web.

Per poder-lo executar, s'ha de disposar d'una JVM, les classes de servlets i un servidor web amb suport per servlets. Per al guió pràctica s'utilitzarà la màquina virtual de Linux i el servidor Tomcat: servidor web escrit en java que suporta servlets i inclou les classes necessàries. Vosaltres podeu realitzar la pràctica tant amb Linux com amb Windows, adaptant les comandes al vostre entorn.

Un cop comprovat que la JVM funciona (es pot executar `javac` i comprovar que la variable CLASSPATH apunta a les classes estàndard de Java):

- **S’ha de baixar el Tomcat i instal·lar-lo**
- **Arrencar el servidor web i provar-ne el funcionament, tal i com explica la documentació que l’acompanya.**

Per provar els servlets, hem d’anar al directori de servlets de prova del Tomcat. Si està instal·lat com un subdirectori del vostre `$HOME` s’hi va amb la ordre:

```
cd ~/jakarta-tomcat-5.0.28/webapps/servlets-examples/WEB-INF/classes
```

Tomcat implementa l’especificació 2.4 dels servlets. Treballa amb unitats anomenades webapps que poden ser empaquetades i instal·lades a altres servidors. Una “webapp” és una aplicació comperta que pot tenir pàgines web estàtiques (arxius html, gif, jpg...) i servlets. Els servlets estan al directori `WEB-INF/classes`. Allà per facilitar les coses podem deixar tant el codi font (`.java`) com el codi compilat (`.class`). En un entorn de producció el codi font no s’hauria de deixar en aquest directori, es farà així per comoditat.

Si arranquem un client web (per exemple Mozilla), i es visita la URL http://localhost:8080/ ens sortirà una pàgina que ha servit el nostre propi Tomcat a la nostra màquina (localhost) al port TCP 8080. Si editem el fitxer de `hola.java` amb el contingut anterior, el compilem amb:

```
javac hola.java
```

Per poder compilar-lo el CLASSPATH ha d’estar apuntant a les classes java i a les classes del servlet, per fer-ho cal executar:

```
export CLASSPATH=$CLASSPATH:~/jakarta-tomcat-5.0.28/common/lib/servlet-api.jar
```

Si no hi ha errors de compilació, tindrem al mateix directori que el fitxer un `hola.class`. Això serà el nostre primer servlet. Però abans de poder-lo executar hem de tenir en compte que a partir de la versió 4 de Tomcat és necessari indicar-li al servidor que el nostre servlet existeix (per raons de seguretat), per això cal afegir al fitxer `~/jakarta-tomcat-5.0.28/webapps/servlets-examples/WEB-INF/web.xml` els següents tags on correspongui:

```
<servlet>
        <servlet-name>hola</servlet-name>
        <servlet-class>hola</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>hola</servlet-name>
        <url-pattern>/servlet/hola</url-pattern>
</servlet-mapping>
```

Sempre que es modifiqui el fitxer `$TOMCAT_HOME/*/web.xml` s’ha de reiniciar el Tomcat per a que els canvis tinguin efecte.

Es pot provar el funcionament visitant amb el client web la URL: http://localhost:8080/servlets-examples/servlet/hola

Formulari i servlet

Si volem enviar al servlet dades per a que les processi, podem provar el següent formulari HTML (`hola2.html`), que col·locarem al directori: `~/jakarta-tomcat-5.0.28/webapps/servlets-examples/`. Si visitem la URL `http://localhost:8080/servlets-examples/hola2.html` podrem veure el formulari que correspon al següent codi HTML:

```
hola2.html:
<html>
<p>Posa el teu nom:</p>
<form action="http://localhost:8080/servlets-examples/hola2"
method=post>
<input type=text name="nom">
i la teva edat:
<input type=text name="edat">
<input type=submit value="Enviar amb Post">
</form>
<hr>
<form action="http://localhost:8080/servlets-examples/servlet/hola2"
method=get>
<input type=text name="nom">
I la teva edat:
<input type=text name="edat">
<input type=submit value="Enviar amb Get">
</form>
</html>
```

Abans de polsar algun botó hem de deixar al lloc adient dels servlets l'arxiu hola2.java i compilar-lo (pensa també d'activar l'execució al `web.xml`). **Un cop obtingut el hola2.class provar la interacció entre un formulari HTML i un servlet que processa les dades que li arriben tant amb una petició GET com 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 nom = req.getParameter("nom");
    String edat = req.getParameter("edat");
    out.println("<html><big>Hola "+ nom +
              " de " + edat + ".</big></html>");
  }
}
```

El mètode `doPost()` es pot implementar molt fàcilment cridant al mètode `doGet()`, ja que en aquest nivell no importa com s'hagin passat les dades al servidor. En general no fa falta el codi següent, ja que un servlet acostuma a implementar només un mètode.

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

L'avantatge principal d'implementar `doPost()` és que pot acceptar gran quantitat de dades d'entrada (és una de les limitacions dels clients web). Com a informació interessant per a altres servlets, totes les dades que es passen en una petició són strings o dades binàries, però no tipus com enters o reals. Si es vol convertir la edat a un enter s'hauria d'introduir el codi següent. No és necessari fer-ho ara, només és per aclarir un dubte que es pot presentar més endavant.

```
int edatI = 0;
try {
  edatI = Integer.parseInt(edat);
} catch (NumberFormatException ignored) { }
if (edat != null)
  out.println("<html><big>Hola " + nom +
              " de " + edat + " " + edatI + ".</big></html>");
else
  out.println("<html>Amic " + nom +
              ", falta l'edat.</html>");
```

Com s'ha comentat abans un servlet "*sobreviu*" a una petició. El cicle de vida és:
1. Crear i inicialitzar el servlet
2. Processar zero o més peticions de clients web
3. Destruir el servlet i "recollir la brossa" (garbage collection)

D'aquesta manera un cop es carrega el servlet, és molt eficient, ja que només hi ha una còpia carregada (s'executen un o varios thread), no s'han de crear nous objectes (només un objecte servlet), i té persistència: pot guardar infomació entre peticions, com comptadors o connexions a la base de dades (això pot ser molt més eficient que obrir i tancar la connexió amb la base de dades a cada petició)

A l'exemple següent s'afegeix l'atribut `cont`, que es va incrementant a cada petició. **Modificar l'exemple anterior per provar l'efecte d'invocar varies vegades el servlet des del client 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 nom = req.getParameter("nom");
    cont ++;
    out.println("<html><big>Hola "+ nom +
                "</big><br>"+ cont +
                " Accessos des de la càrrega.</html>");
    }
  public void doPost(HttpServletRequest req, HttpServletResponse res)
                    throws ServletException, IOException {
    doGet(req, res);
  }
}
```

Així ja tenim una solució al problema de guardar l'estat, els CGI no ho permetien, i haviem de guardar nosaltres mateixo l'arxiu o la base de dades. De totes maneres, no tot és perfecte, amb els servlets hi ha el problema dels accessos concurrents a variables o parts del codi crítiques. S'han d'utilitzar monitors o semàfors per a gestionar aquestos accessos concurrents. Java proporciona una forma molt fàcil de treballar amb monitors: `synchronized()`

```
synchronized(this) {
  cont ++;
  out.println("<html><big>Hola "+ nom +
              "</big><br>"+ cont + " Accessos desde la càrrega.
</html>");
}
```

A més si el servlet crea threads, també són persistents. A continuació s'utilitza el mètode `init()` i el mètode `destroy()` per iniciar i parar un thread d'execució, el que permet que el servlet vagi treballant durant la seva vida mentre va atenent a altres threads les peticions que arriben. El següent exemple, el thread que inicia el mètode `init()` va comptant els segons transcorreguts des de la càrrega del servlet: dorm durant 1000 milisegons i incrementa en un el comptador de segons dins d'un bucle infinit.

```
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;
  Thread contadorSegs;
  public void init(ServletConfig config)
```

```
                   throws ServletException {
    super.init(config); // sempre
    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();
}
```

Model d'execució alternativa.

El model d'execució estàndard és una sola instància del servlet per cada servlet registrat, amb tants thread com peticions simultànies. Hi ha un altre model d'execució: tenir tantes instàncies del servlet processant cada una de les peticions diferents.

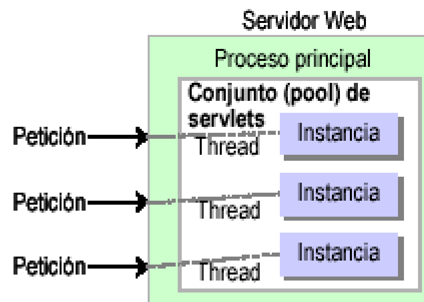

Figura 6.- El model d'un thread per servlet.

Aquest model s'activa indicant que s'implementa la interfície `javax.servlet.SingleThreadModel`. No té cap requisit adicional, indicant que implementa aquesta interfície ja actua com és esperat. Dit d'una altra forma mai dos thread executaran el mètode `service()` d'una instància d'un servlet.

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

Aquest model no té utilitat per a un comptador o aplicacions que volen mantenir un estat comú. Pot servir per a evitar sincronització però tractant peticions de forma eficient. Per exemple una aplicació de base de dades on cada instància ha d'executar un conjunt d'operacions que forma part d'una transacció o, on cada petició és diferent. D'aquesta manera, cada instància tindrà una connexió diferent a la base de dades.

NOTA: actualment aquesta interfície es troba *Deprecated* des de la versió 2.4 de l'API de servlets, la suportada per tomcat 5.X, si bé segueix funcionant.

Disseny

Després d'aquesta petita introducció a la programació de servlets, passem al nostre primer disseny. Volem que dissenyeu un servlet que tingui la mateixa aplicació que heu fet a la pràctica de CGI.

El servlet ha d'acceptar lloguers de cotxes extenent el mètode GET `doGet()`. El mateix servlet ha de llistar els lloguers extenent el mètode POST `doPost()`. En aquest cas els lloguers es poden guardar en variables globals dins del propi servlet (la forma real de fer-ho seria amb una base de dades).

Recordeu que els paràmetres són String. Per poder processar i convertir aquestes cadenes a altres tipus podeu utilitzar java.lang.String.

Referències

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 5.0 <http://tomcat.apache.org/tomcat-5.0-doc/index.html> [Consulta: 26 febrer 2007]

Sun Microsystems, Inc. Java (TM) Servlet Technology. [en línea] <http://java.sun.com/products/servlet/index.html> [Consulta: 26 febrer 2007]

Sun Microsystems, Inc. Java™ 2 Platform, Standard Edition, 1.4.2 API Specification. [en línea] < http://java.sun.com/j2se/1.4.2/docs/api/> [Consulta: 26 febrer 2007]

Sun Microsystems, Inc. Java™ 2 Platform, Standard Edition, 5.0 API Specification. [en línea] <http://java.sun.com/j2se/1.5.0/docs/api/> [Consulta: 26 febrer 2007]

Dan Connolly. CGI: Common Gateway Interface [en línia] 13 octubre 1999 <http://www.w3.org/CGI/> [Consulta: 26 febrer 2007]

Rob Saccoccio. FastCGI [en línea] <http://www.fastcgi.com/> [Consulta: 26 febrer 2007]