CAPÍTULO 14. DESARROLLO DE SISTEMAS ESPECÍFICOS

1. Introducción

En los últimos años han aparecido multitud de nuevas plataformas para desarrollar aplicaciones y ponerlas en explotación. En este capítulos echaremos un vistazo a algunas técnicas para desarrollar aplicaciones web, servicios web, sistemas de información geográficos y aplicaciones basadas en componentes distribuidos; concluiremos con una sección en la que veremos algunos patrones específicamente diseñados para su aplicación en el desarrollo de sistemas colaborativos.

2. Desarrollo de aplicaciones web

Una aplicación web es una aplicación que ofrece un conjunto de funcionalidades que pueden ejecutarse a través de un navegador que utiliza el cliente. Mediante protocolo *http*, el cliente envía peticiones al servidor, que las ejecuta y que devuelve resultados.

Un gran ventaja de las aplicaciones web es la independencia de la plataforma de ejecución (en el cliente), ya que lo único que necesita es un navegador que interprete adecuadamente texto *html* y, en ocasiones, algunas capacidades adicionales que hoy en día son ofrecidas de manera generalizada (ejecución de *applets* o de lenguajes de *script*, por ejemplo).

Las dos siguientes figuras muestran muy esquemáticamente la diferencia entre descargar un página estática de un servidor web y descargar una página dinámica: en el primer caso, el servidor recibe la petición, recupera la página de su lugar de almacenamiento y la entrega tal cual al cliente, que ya sabrá interpretarla y motrararla; en el segundo caso, el servidor recibe la petición (una *url* quizás con uno o más parámetros) y realiza algún tipo de procesamiento para entregar al cliente los resultados en un formato que éste entienda (normalmente, en forma de código *html*). El código que el servidor interpreta y ejecuta se conoce como "*script* de servidor".

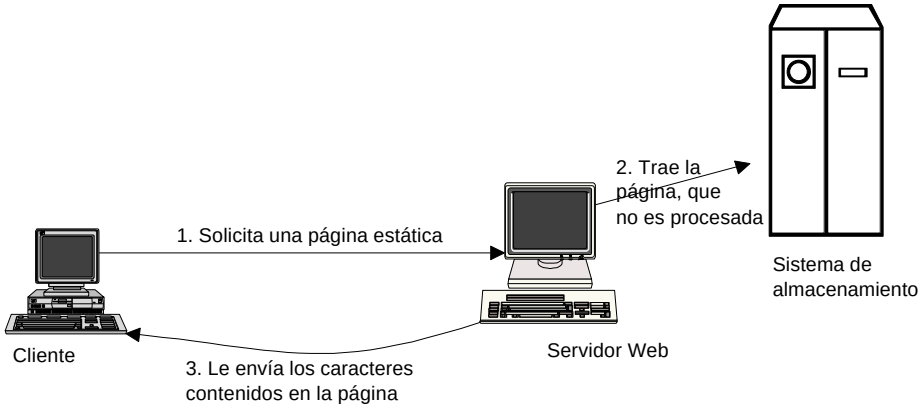


Figura 154. Solicitud de una página sin *scripts* de servidor

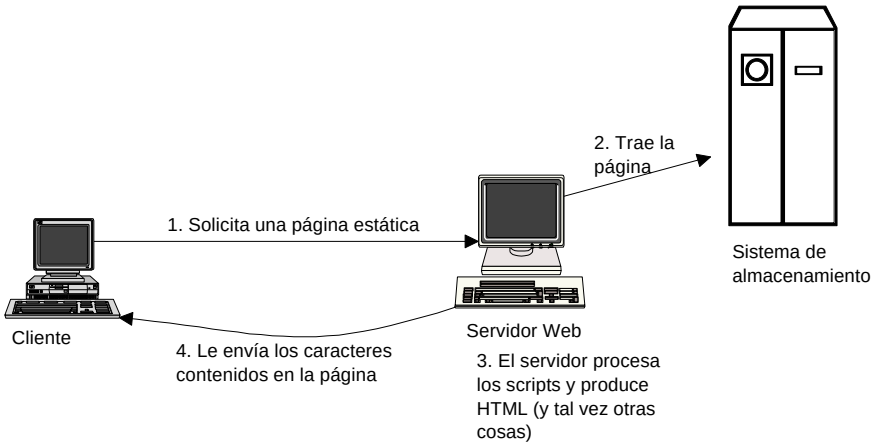


Figura 155. Solicitud de una página con *scripts* de servidor

Existen muchas tecnologías para la programación de "páginas activas" (es decir, con scripts de servidor), entre ellas CGI, PHP, JSP, Servlets, ASP, ASP.NET o CSP, pero básicamente su funcionamiento es siempre el descrito. Existen, por otro lado, lenguajes de script de cliente, como JavaScript o Visual Basic Script, que se ejecutan únicamente en el navegador del cliente y que no deben, en principio, comprometer la seguridad de éste. Evidentemente, y como se ha dicho antes, para que un script de cliente se ejecute en el navegador, éste debe ser capaz de interpretarlo y de ejecutarlo.

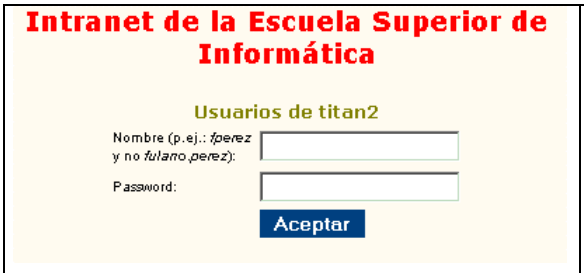
2.01 Envío de parámetros a través de una URL

El protocolo http 1.1 (http://www.w3.org/Protocols/rfc2616/rfc2616.html) define dos tipos importantes de mensajes, *Request* y *Reponse*, que respectivamente representan solicitudes que los clientes realizan al servidor y respuestas que los servidores envían a los clientes.

Existen, además, dos tipos destacados de *requests*, las que se realizan siguiendo el método *get* y las que se realizan siguiendo el método *post*:

- Mediante *get*, se solicita del servidor algún tipo de recurso especificando quizás un conjunto de parámetros que viajan con la url, no pudiendo superar ésta los 1024 bytes. Así, la URL *http://161.67.27.108:8080/curri/vercurri.jsp?formato=uclm&id=5450675* representa una solicitud *get* a la máquina *161.67.27.108* en su puerto *8080*; se le solicita que nos devuelva la página *curri/vercurri.jsp* y le estamos pasando *uclm* y *5450675* como valores respectivos de los parámetros *formato* e *id.*
- Mediante *post* se solicita al servidor algún tipo de recurso especificando quizás un conjunto de parámetros pero, en este caso, sin limitación de tamaño, ya que el conjunto de parámetros y sus valores viajan en una variable junto a la petición.

La siguiente figura muestra el aspecto visual de un sencillo formulario de identificación y su correspondiente código html; en éste, se ha señalado con negrita el fragmento de código que representa la acción que debe ejecutarse en el servidor cuando se pulse el botón etiquetado “Aceptar”, así como el método de envío de los datos: el nombre (en una caja de texto llamada “nombre”) y la password (en una caja especial que muestra asteriscos llamada “pwd”) los recibirá el servidor mediante la url *servlet/identificar*; además, ambos parámetros se enviarán usando el método *post*. Estos dos últimos datos aparecen en la primera línea que describe el formulario.




```
<form action=servlet/identificar method=post>
<table>
<tr>
      <td colspan="2">
        <p align="center"><b>Usuarios de titan2</b>
      </td>
</tr>
<tr>
      <td>
            Nombre (p.ej.: <i>fperez</i><br>y no
                        <i>fulano.perez</i>):
      </td>
      <td>
            <input type=text name=nombre>
      </td>
</tr>
<tr>
      <td>Password:</td>
      <td><input type=password name=pwd></td>
</tr>
<tr>
      <td>
        <input type=submit class=boton value=Aceptar>
      </td>
</tr>
</table>
</form>
```

Figura 156. Un sencillo formulario y su correspondiente código html

En este ejemplo, por tanto, los parámetros “nombre” y “pwd” se envían a una url que debe encargarse de recogerlos, leer sus valores y procesarlos como corresponda. Puesto que los datos se envían utilizando “post”, la url receptora debe ser capaz de responder a este método de envío. Si se está utilizando un servlet, como es este caso, su correspondiente clase (un servlet es una clase normal) debe contener la implementación del método *doPost*.

doPost toma dos parámetros: un objeto de clase HttpServletRequest, por el que accedemos a los datos que el cliente nos manda, y un objeto de clase HttpServletResponse, que no permite entregarle los resultados. La siguiente figura muestra la posible implementación de este método en el servlet, contenido en la clase *identificar.java* (el nombre de la clase se corresponde con el último fragmento de la URL).

Un servlet, una página JSP o ASP o CSP o PHP o de cualquier otro tipo puede aprovechar por completo todas las funcionalidades de su correspondiente lenguaje de programación. En este caso, el servlet va a comprobar que el nombre y contraseña enviados desde el formulario se encuentren almacenados en una base de datos a la que se va acceder por medio de un agente de base de datos (Broker).

Casi lo primero que se hace es crear una instancia del Broker pasándole como argumento el valor del parámetro nombre que se ha recibido desde el formulario (este valor se lee mediante la instrucción *req.getParameter("nombre")*, donde *req* es el objeto de clase *HttpServletRequest*) así como la dirección IP del cliente (se lee mediante *req.getRemoteAddr()*). Lo que pase ahora dependerá evidentemente de la implementación del constructor *Broker(String, String)*, pero podemos suponer que lo que realmente pasa es que se establece la conexión a la base de datos.

A continuación se crea *usu*, un objeto de clase *Usuario* pasando como argumentos el broker que se acaba de crear, el nombre de usuario y la contraseña. Este constructor de *Usuario* comprueba que el nombre y la contraseña existen en la base de datos asociada al Broker que se ha instanciado en la instrucción anterior.

Podemos suponer que si el establecimiento de la conexión o la identificación del usuario falla, el control del programa saltará al bloque *catch* situado más abajo. En caso de que ambas sentencias hayan funcionado, se procede con la siguiente instrucción.

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)  throws ServletException, IOExce
ption
{
      resp.setContentType("text/html");
      PrintWriter out = new PrintWriter(resp.getOutputStream());
      try {
            Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
            Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
                  ...
      }
      catch (Exception e) {
            req.getSession(true).setAttribute("mensaje", e.getMessage());
            out.println("<html><body onload=\"window.location.href='../error.jsp';\"></body></html>");
      }
      out.close();
   }
```

Figura 157. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde el formulario de la Figura 156

2.02 *http* es un protocolo sin estado

http es un protocolo "sin estado", lo que significa que el servidor no guarda información de la conexión con sus clientes: es decir, que aunque el cliente se conecte e identifique correctamente, si no conservamos de alguna manera la información suya que nos sea relevante, en la segunda página que visitara habríamos perdido sus datos de conexión e identificación y podríamos, por ejemplo, vetarle el acceso, prohibirle la ejecución de operaciones, etc.

Por ello, todas las tecnologías de desarrollo de aplicaciones web incluyen una clase específica que permite almacenar información de la conexión: la sesión (en entornos Java es la HttpSession). La sesión consiste realmente en una *cookie* que se guarda en el disco del cliente y que dura lo que dura la conexión. Por eso, si prohibimos totalmente las cookies en nuestro navegador de Internet, tendremos serias dificultades para interactuar, por ejemplo, con nuestra oficina bancaria virtual. Por eso es importante también, por motivos de seguridad, usar las opciones de desconexión o de cerrar sesión que habitualmente incluyen las aplicaciones web.

En el ejemplo, nos habíamos quedado con que la conexión a la base de datos se había establecido y con que el usuario se había identificado correctamente. Con el fin de que, por toda su visita al portal, el servidor conozca los datos del cliente que está conectado, debemos entonces guardar alguna información en dicha sesión. La Figura 158 incluye el mismo código que la figura anterior más dos nuevas líneas, marcadas en negrita, en las que se recupera un objeto sesión y se colocan en él dos objetos: el Broker, que da así asignado a este cliente para que realice a través de él todas las operaciones de

base de datos, y el usuario, con su nombre, tal vez la contraseña y otros datos que puedan ser de interés.

En Java, la sesión se recupera mediante una llamada al método *getSession(boolean)* del objeto de tipo *HttpServletRequest* pasada como parámetro al *doPost.* Cada conexión puede tener asociada una sola sesión. Si el valor del parámetro es *true*, se le crea una sesión nueva, destruyéndose sin preguntar más todo lo que hubiera en la sesión antigua; si el valor es *false*, el método devuelve (si la hay) la sesión de la conexión. En el ejemplo de la Figura 158, se le pasa el valor *true* porque el cliente acaba de identificarse, por lo que se supone que acaba de llegar al portal y que no tenía sesión. En las siguientes páginas que visite recuperaremos la sesión pasándole valor *false*, porque se estará utilizando la sesión creada al haber identificado al usuario.

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)  throws ServletException, IOException
{
      resp.setContentType("text/html");
      PrintWriter out = new PrintWriter(resp.getOutputStream());
      try {
            Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
            Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
            HttpSession sesion=req.getSession(true);
            sesion.setAttribute("bd", bd);
            sesion.setAttribute("usuario", usu);
                  ...
      }
      catch (Exception e) {
            req.getSession(true).setAttribute("mensaje", e.getMessage());
            out.println("<html><body onload=\"window.location.href='../error.jsp';\"></body></html>");
      }
      out.close();
   }
```

Figura 158. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde la Figura 156, continuación de la Figura 157

Por último, si queremos que el navegador del cliente muestre un mensaje cuando el usuario se ha identificado correctamente, el servlet le puede enviar una cadena (normalmente en formato HTML) a través de un objeto de tipo *PrintWriter*, que se habrá recuperado a partir del objeto *HttpServletRequest:*

```
public void doPost(HttpServletRequest req, HttpServletResponse resp)  throws ServletException, IOExce
ption
{
      resp.setContentType("text/html");
      PrintWriter out = new PrintWriter(resp.getOutputStream());
      try {
            Broker bd=new Broker(req.getParameter("nombre"), req.getRemoteAddr());
            Usuario usu=new Usuario(bd, req.getParameter("nombre"), req.getParameter("pwd"));
            HttpSession sesion=req.getSession(true);
            sesion.setAttribute("bd", bd);
            sesion.setAttribute("usuario", usu);
            String s="<html><head>"+
                  "</head>"+
                  "<body onload=\"window.location.href= '../menu.jsp';\">"+
                  "</body>"+
                  "</html>";
            out.println(s);
      }
      catch (Exception e) {
         req.getSession(true).setAttribute("mensaje", e.getMessage());
                  out.println("<html><body onload=\"window.location.href='../error.jsp';\"></body></html>");
      }
      out.close();
   }
```

Figura 159. Fragmento del método *doPost* del servlet que recibe el nombre y la contraseña enviados desde la Figura 156, continuación de la Figura 158

2.03 Arquitectura de una aplicación web

En principio, una aplicación web puede ofrecer tantas funcionalidades como una "de escritorio", siendo además deseable que el código de una sea reutilizable para el desarrollo de la otra. Si usamos una arquitectura multicapa, se pondrá especial cuidado en diseñar adecuadamente la capa de dominio, con el fin de que sirva para las dos aplicaciones.

Así, por ejemplo, en el caso de la aplicación bancaria, el siguiente método, que inserta un producto en la base de datos (este código aparece en la página 58), no sería utilizable para una aplicación web del estilo de la que hemos descrito en el epígrafe anterior:

```
public void insert() throws Exception {
   String SQL="Insert into Producto (Codigo, NIFCliente) values (" +
      mPKCodigo + ", '" + mCliente.getNIF() + "')";
   Broker.getBroker().insert(SQL);
}
```

¿Por qué? Porque, según se ha comentado, cada cliente tiene asociada una sesión en la que se almacena, entre otras informaciones, el Broker que utiliza para realizar las operaciones de persistencia. En el código de arriba, el agente se recupera mediante una llamada al método *getBroker()* de la clase *Broker*, que es un singleton. Que el agente sea un singleton interesa poco para un aplicación web, que va a tener visitas múltiples y

simultáneas. Lo que haremos será pasar el Broker como parámetro al método *insert*. La aplicación de escritorio deberá tener esto en cuenta si queremos que una sola capa de dominio sirva para el desarrollo de las dos aplicaciones:

```
public void insert(Broker bd) throws Exception {
    String SQL="Insert into Producto (Codigo, NIFCliente) values (" +
       mPKCodigo + ", '" + mCliente.getNIF() + "')";
    bd.insert(SQL);
}
```

Como en la aplicación del escritorio, el usuario usa las funcionalidades de la aplicación mediante la capa de presentación. En el caso de la aplicación web, el usuario interactúa enviando datos a URLs mediante formularios, y estas URLs serán servlets, páginas JSP, ASP o de cualquier otro tipo. La URL receptora de los datos del formulario será la encargada de recuperar el Broker del objeto sesión, actualizar el estado del objeto de dominio que se esté manipulando, y llamar al correspondiente método (un método CRUD, por ejemplo, pero puede ser cualquier otro método de negocio) pasándole si es preciso el Broker como parámetro.

Esta secuencia en los dos siguientes pasos:

1) El usuario rellena los datos de un cliente en una página web y pulsa el botón "Guardar nuevo":

Figura 160

2) Los datos se envían mediante un post a un servlet llamado "gestCliente". El servlet mira qué botón se ha pulsado en la página anterior para saber la operación que debe realizar. Esto se consigue dando a todos los botones el mismo *name* pero distinto *value* (p.ej.: *<input type=submit name=boton value='Guardar nuevo'>* o *<input type=submit name=boton value='Modificar'>*).

```
...
HttpSession sesion=request.getSession(false);
if (sesion!=null) {
      String BOTON=request.getParameter("BOTON");
      Usuario usu=(Usuario) sesion.getAttribute("usuario");
      Broker bd=(Broker) sesion.getAttribute("bd");

      if (BOTON.equals("Guardar nuevo")) {
            try {
                  Cliente o=new Cliente();
                  cargaObjeto(request, o, usu, sesion, IConstantesBotones.SAVE_NEW);
                  o.insert(bd);
                  out.println("<html><body> ….  </body></html>");
                  }
                  catch (Exception e) {
                        sesion.setAttribute("mensaje", e.getMessage());
                        out.println("<html><body> ….  </body></html>");
                  }
            }
      }
      ...
```

Figura 161. Fragmento del método *doPost* en el servlet que recibe los datos del formulario mostrado en la Figura 160

2.04 Ejercicio

El sistema de gestión bancaria permite la manipulación de los datos de su base de datos mediante una aplicación web y una aplicación de escritorio que comparten capa de dominio. Ambas aplicaciones permiten la obtención de listados de, por ejemplo, clientes, como se muestra en la siguiente figura:

Figura 162

Para la obtención del listado de la derecha, el programador añadió un método *getListado* a la clase *dominio.Cliente,* que devuelve en formato HTML la tabla que se muestra. Explique si esta solución es adecuada o inadecuada y por qué.

2.05 Ejercicio

¿Qué desventajas tiene usar un agente de base de datos singleton en una aplicación web?