Procesamiento de peticiones

Índice

Un servlet maneja peticiones de los clientes a través de su método **`service`**. Con él se pueden manejar peticiones HTTP (entre otras), reenviando las peticiones a los métodos apropiados que las manejan. Por ejemplo, una petición GET puede redirigirse a un método `doGet`. Veremos ahora los elementos principales que intervienen en una interacción vía HTTP.

1. Peticiones: HttpServletRequest

Como hemos visto anteriormente, los objetos **`ServletRequest`** se emplean para obtener información sobre la petición de los clientes. Más en concreto, el subtipo **`HttpServletRequest`** se utiliza en las peticiones HTTP. Proporciona acceso a los datos de las cabeceras HTTP, cookies, parámetros pasados por el usuario, etc, sin tener que parsear nosotros a mano los datos de formulario de la petición.

La clase dispone de muchos métodos, pero destacamos los siguientes:

- Para **obtener los valores de los parámetros** pasados por el cliente, se tienen los métodos:

  ```
  Enumeration getParameterNames()
  String      getParameter (String nombre)
  String[]    getParameterValues (String nombre)
  ```

  Con **`getParameterNames()`** se obtiene una lista con los nombres de los parámetros enviados por el cliente. Con **`getParameter()`** se obtiene el valor del parámetro de nombre `nombre`. Si un parámetro tiene varios valores (por ejemplo, si tenemos un array de cuadros de texto con el mismo nombre en un formulario), se pueden obtener todos separados con **`getParameterValues()`**. Los nombres de los parámetros normalmente sí distinguen mayúsculas de minúsculas, deberemos tener cuidado al indicarlos.

- Para **obtener la cadena de una petición GET**, se tiene el método:

  ```
  String getQueryString()
  ```

  que devuelve todos los parámetros de la petición en una cadena, que deberemos parsear nosotros como nos convenga.

- Para **obtener datos de peticiones POST, PUT o DELETE**, se tienen los métodos:

  ```
  BufferedReader     getReader()
  ServletInputStream getInputStream()
  ```

  Con **`getReader()`** se obtiene un `BufferedReader` para peticiones donde esperemos recibir texto. Si esperamos recibir datos binarios, se debe emplear **`getInputStream()`**. Si lo que esperamos recibir son parámetros por POST igual que se haría por GET, es mejor utilizar los métodos `getParameterXXXX(...)` vistos

antes.

- Para **obtener información sobre la línea de petición**, se tienen los métodos:

```
String getMethod()
String getRequestURI()
String getProtocol()
```

Con **`getMethod()`** obtenemos el comando HTTP solicitado (GET, POST, PUT, etc), con **`getRequestURI()`** obtenemos la parte de la URL de petición que está detrás del `host` y el puerto, pero antes de los datos del formulario. Con **`getProtocol()`** obtenemos el protocolo empleado (`HTTP/1.1`, `HTTP/1.0`, etc).

2. Respuestas: HttpServletResponse

Los objetos **`ServletResponse`** se emplean para enviar el resultado de procesar una petición a un cliente. El subtipo **`HttpServletResponse`** se utiliza en las peticiones HTTP. Proporciona acceso al canal de salida por donde enviar la respuesta al cliente.

La clase dispone de muchos métodos, pero destacamos:

```
Writer              getWriter()
ServletOutputStream getOutputStream()
```

Con **`getWriter()`** se obtiene un `Writer` para enviar texto al cliente. Si queremos enviar datos binarios, se debe emplear **`getOutputStream()`**.

Si queremos especificar información de cabecera, debemos establecerla ANTES de obtener el `Writer` o el `ServletOutputStream`. Hemos visto en algún ejemplo el método **`setContentType()`** para indicar el tipo de contenido. Veremos las cabeceras con más detenimiento más adelante.

3. Procesamiento de peticiones GET y POST

Como se ha visto anteriormente, el método **`doGet()`** se emplea para procesar peticiones GET. Para realizar nuestro propio procesamiento de petición, simplemente sobreescribimos este método en el servlet:

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

Podemos utilizar los métodos del objeto `HttpServletRequest` vistos antes. Así podremos, entre otras cosas:

- Acceder a elementos de la petición, como valores de parámetros:
  ```
  String nombreUsuario = request.getParameter("nombre");
  ```
- Acceder a los parámetros en la cadena de la petición y procesarlos como queramos:
  ```
  String query = request.getQueryString();
  ...
  ```
- Obtener un canal de entrada (`Reader` o `InputStream`) con que leer los datos de la petición:
  ```
  BufferedReader r = request.getReader();
  ...
  ```
  Esta, sin embargo, no es una buena idea para tomar parámetros de peticiones u otras cosas. Se suele emplear sobre todo para transferencias de ficheros, pero hay que tener en cuenta que si obtenemos un canal de entrada, luego no podremos obtener parámetros u otros valores con métodos `getParameter()` y similares.
- etc.

También podemos utilizar los métodos del objeto **HttpServletResponse** para, entre otras cosas:

- Establecer valores de la cabecera (antes que cualquier otra acción sobre la respuesta):
  ```
  response.setContentType("text/html");
  ```
- Obtener el canal de salida por el que enviar la respuesta:
  ```
  PrintWriter out = response.getWriter();
  out.println ("Enviando al cliente");
  ```
- Redirigir a otra página:
  ```
  response.sendRedirect("http://localhost:8080/pag.html");
  ```
- etc.

De forma similar, el método **`doPost()`**, se emplea para procesar peticiones POST. Igual que antes, debemos sobreescribir este método para definir nuestro propio procesamiento de la petición:

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
throws ServletException, IOException
{
       // ... codigo para una peticion POST
}
```

Las posibilidades de los parámetros `HttpServletRequest` y `HttpServletResponse` son las mismas que para GET. Normalmente muchos servlets definen el mismo código para uno y otro método (hacen que `doPost()` llame a `doGet()` y definen allí el código, o al revés), pero conviene tenerlos separados para poder tratar independientemente uno y otro tipo de peticiones si se quiere.

3.1. Procesamiento secuencial de peticiones

Los servlets normalmente pueden gestionar múltiples peticiones de clientes concurrentemente. Pero puede suceder que, si los métodos que definimos acceden a un recurso compartido, no nos interese que varios clientes accedan a dicho recurso simultáneamente. Para solucionar este problema, podemos definir bloques de código `synchronized`, o bien hacer que el servlet sólo atienda una petición cada vez.

Para esto último, lo único que hay que hacer es que el servlet, además de heredar de `HttpServlet`, implemente la interfaz **SingleThreadModel**. Esto no supone definir más métodos, simplemente añadimos el `implements` necesario al definir la clase Servlet, y ya está:

```
public class MiServlet
extends HttpServlet implements SingleThreadModel
{
        ...
}
```

4. Manejo de formularios

Los datos que se envían como parámetros en una petición (tras el interrogante si es una petición GET, o por otro lado si es POST) se llaman **datos de formulario**. Una vez enviados estos datos como petición, ¿cómo se extraen en el servidor?

Si trabajáramos con CGI, los datos se tomarían de forma distinta si fuese una petición GET o una POST. Para una GET, por ejemplo, tendríamos que tomar la cadena tras la interrogación, y parsearla convenientemente, separando los bloques entre '&', y luego separando el nombre del parámetro de su valor a partir del '='. También hay que descodificar los valores: los alfanuméricos no cambian, pero los espacios se han convertido previamente en '+', y otros caracteres se convierten en '%XX%'.

Con servlets todo este análisis se realiza de forma automática. La clase `HttpServletRequest` dispone de métodos que devuelven la información que nos interesa ya procesada, e independientemente de si es una petición GET o POST. Hemos visto antes los métodos:

```
Enumeration getParameterNames()
String      getParameter (String nombre)
String[]    getParameterValues (String nombre)
```

4.1. Ejemplo

Veamos un ejemplo: supongamos que tenemos este formulario:

```
<html>
<body>
<form action="/appforms/servlet/ejemplos.ServletForm">
        Valor 1: <input type="text" name="texto1">
        <br>
        Valor2:
        <select name="lista">
        <option name="lista" value="Opcion 1">Opcion 1</option>
        <option name="lista" value="Opcion 2">Opcion 2</option>
        <option name="lista" value="Opcion 3">Opcion 3</option>
        </select>
        <br>
        Valores 3:
        <br>
        <input type="text" name="texto2">
        <input type="text" name="texto2">
        <input type="text" name="texto2">

        <input type="submit" value="Enviar">
</form>
</body>
</html>
```

Al validarlo se llama al servlet `ServletForm`, que muestra una página HTML con los valores introducidos en los parámetros del formulario:

```
package ejemplos;

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

public class ServletForm extends HttpServlet
{
        // Metodo para GET

        public void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException
        {
                response.setContentType("text/html");

                PrintWriter out = response.getWriter();

                // Mostramos los datos del formulario

                out.println ("<HTML>");
                out.println ("<BODY>");
                out.println ("<H1>Datos del formulario</H1>");
                out.println ("<BR>");

                String valor1 =
```

```
                        request.getParameter("texto1");
                String valor2 =
                        request.getParameter("lista");
                String[] valor3 =
                        request.getParameterValues("texto2");

                out.println ("Valor 1:" + valor1);
                out.println ("<BR>");
                out.println ("Valor 2:" + valor2);
                out.println ("<BR>");
                out.println ("Valor 3:");
                out.println ("<BR>");
                if (valor3 != null)
                        for (int i = 0; i < valor3.length; i++)
                        {
                                out.println (valor3[i]);
                                out.println ("<BR>");
                        }

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

        // Metodo para POST

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

Para probar el ejemplo que viene en las plantillas, cargamos la URL:

```
http://localhost:8080/appforms/index_form.html
```

5. Pruebas de servlets con Cactus

Ya vimos en el módulo de servidores web cómo configurar la aplicación para poder realizar pruebas con Cactus en sus diferentes elementos. Una vez la tenemos configurada... ¿cómo escribimos el caso de prueba para un servlet?

Para escribir un caso de prueba, debemos seguir los siguientes pasos

1. Importar los paquetes de JUnit y Cactus
   ```
   import org.apache.cactus.*;
   import junit.framework.*;
   ```
2. Extender un clase de prueba de Cactus (otra posibilidad menos recomendable es reusar un prueba JUnit).
   Necesitamos crear una clase que herede de uno de las casos de prueba Cactus,

dependiendo de lo que estemos probando.

- **Para servlets:**: `ServletTestCase`: debemos extender esta clase para probar codigo que utiliza los objetos del API de los *Servlets* (`HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletConfig`, `ServletContext`, ...), como los *Servlets* o aquellas clases que accedan al contenedor web (como un objeto *DAO* que utiliza un conexión a la base de datos definida en un *DataSource*). Por ejemplo:
```
public class TestSampleServlet extends ServletTestCase
{
```
- **Para páginas JSP:**: `JspTestCase`: si queremos probar objetos del API *JSP* (`PageContext`, `JspWriter`, ...) como *Taglibs*. Por ejemplo:
```
public class TestSampleTag extends JspTestCase
{
```
  Veremos algún ejemplo de estas pruebas más adelante, cuando veamos las páginas JSP.
- **Para filtros:**: `FilterTestCase`: si queremos probar objetos del API de *Filtros* (`FilterChain`, `FilterConfig`, `HttpServletRequest`, `HttpServletResponse`, ...) como *Filtros*. Por ejemplo:
```
public class TestSampleFilter extends FilterTestCase
{
```

La clase de prueba, como en JUnit, deberá estar en el mismo paquete que la clase a probar (aunque su código fuente lo pondremos dentro de la carpeta fuente `test`, en lugar de `src`, donde estará el fuente del servlet a probar).

3. [Opcional] Métodos `setUp()` y `tearDown()`
   Igual que con JUnit, podemos definir estos métodos los cuales se ejecutan antes y después de cada caso de prueba. Sin embargo, mientras que JUnit los ejecuta en el lado del cliente, en Cactus se ejecutan en el servidor. Esto significa que podemos acceder a los objetos del API definidos en el paso anterior. Dicho de otro modo, podemos realizar cosas como explorar parámetros de la petición, o establecer atributos, o explorar cabeceras, antes de llamar a los casos de prueba, etc...
4. [Opcional] Métodos `begin()` y `end()`
   Estos son los métodos equivalentes a `setUp()` y `tearDown()`, pero ejecutados en el lado del cliente. Por lo tanto se ejecutan en el lado del cliente antes de cada prueba.
5. Métodos `testXXX()`
   Como en JUnit, el método `main` de una prueba es el método `testXXX()`. La diferencia estriba en que en Cactus estos métodos se ejecutan en el contenedor. Cada caso de prueba `XXX` debe definir su método `testXXX()`
   En los métodos `testXXX()` podremos:
   - Instanciar la clase a probar (también podemos refactorizar esta instancia fuera del método y definirla como una variable de instancia de clase)

- Configurar cualquier objeto de servidor del dominio (poner variables en la sesión, etc...). De hecho, las clases Cactus que hemos heredado en el paso 2 contiene diversas variables de instancia se inicializan con objetos validos.
- Llamar al método a probar
- Realizar aserciones JUnit (`asserts(..)`, `assertEquals(...)`, `fail(...)`, ...) para verificar que la prueba ha funcionado correctamente.

6. [Opcional] Métodos `beginXXX()`
   Para cada caso de prueba XXX, podemos definir los correspondientes métodos `beginXXX()`. Se utilizan para inicializar parámetros relacionados con HTTP (parámetros HTTP, cookies, cabeceras HTTP, URL a simular, ...). Posteriormente podremos acceder a estos valores en los métodos `testXXX()` mediante las llamadas a las diferentes API de `HttpServletRequest` (como `getQueryString()`, `getCookies()`, `getHeader()`, ...).
   La firma del método begin es:

```
public void beginXXX(WebRequest laRequest)
{
  [...]
}
```

   donde utilizaremos el objeto `laRequest` para poner todos los parámetros relacionados con HTTP.
   Los métodos `beginXXX()` se ejecutan en el lado del cliente, antes de ejecutar los métodos `testXXX()` en el servidor, y por lo tanto no se tiene acceso a las variables de clase que representan a los objetos del API (sus valores son nulos)
7. [Opcional] Métodos `endXXX()`
   Para cada caso de prueba XXX, podemos definir los correspondientes métodos `endXXX()`. Se utilizan para verificar los parámetros HTTP devueltos por los casos de prueba (como el contenido devuelto por la respuesta HTTP, cualquier cookie devuelta, cabeceras HTTP, etc...).
   Para versiones de Cactus hasta la 1.1, la firma del método es la siguiente:

```
public void endXXX(HttpURLConnection theConnection)
{
  [...]
}
```

   y algunos métodos de ayuda (clase `AssertUtils`) para extraer el contenido de la respuesta y las cookies.
   Sin embargo, a partir de Cactus 1.2, esta firma ha quedado "deprecated". Ahora existen 2 posibles firmas para el método `end`, dependiendo de si necesitamos realizar comprobaciones complejas del contenido de vuelta. Para comprobaciones complejas, utilizaremos la integración con el framework *HttpUnit*. Para comprobaciones sencillas, el

objeto `WebResponse`.
Los métodos `endXXX()` se ejecutan en el lado del cliente, tras ejecutar los métodos `testXXX()` en el servidor, y por lo tanto no se tiene acceso a las variables de clase que representan a los objetos del API (sus valores son nulos)

A continuación se muestra un pequeño ejemplo que prueba un *Servlet*:

```
import java.io.*;
import org.apache.cactus.*;

public class PruebaServletTest extends ServletTestCase
{
        public void beginXXX(WebRequest theRequest)
        {
            // Inicializa los parámetros HTTP relacionados
            theRequest.setURL("jakarta.apache.org", "/mywebapp",
"/test/test.jsp",
                null, null);
            theRequest.addCookie("cookiename", "cookievalue");
        }

        public void testXXX()
        {
                //Servlet a probar
            PruebaServlet servlet = new PruebaServlet();
            servlet.init(config);

            // Llama al metodo a probar, por ejemplo, doGet
            try
            {
                    servlet.doGet(request, response);
            } catch (Exception ex) {}

            // Realiza algunas asercion en el lado del servidor
            assertEquals("someValue",
session.getAttribute("someAttribute"));
            assertEquals("jakarta.apache.org", request.getServerName());
        }

        public void endXXX(WebResponse theResponse)
        {
            // Aserta la respuesta HTTP
            Cookie cookie = theResponse.getCookie("someCookie");
            assertEquals("someValue2", cookie.getValue());

            assertEquals("some content here", theResponse.getText());
        }
```