

Using servlets and JSP

Philippe Laroque

laroque@u-cergy.fr

\$Id: servlets.lyx 1589 2008-11-19 08:32:30Z phil \$

Abstract

This small document describes the main features of the servlet technology, including code examples for each important feature. An introduction to Java Server Pages (JSP) is also presented.

1 Basic principles

Servlets are JAVA programs that run on a web server and build web pages. In that sense, they are an alternative to classical CGI (Common Gateway Interface) scripts.

This section first lists the advantages of using servlets over traditional scripts, then describes the servlet actors (classes and interfaces).

1.1 Advantage of servlets over CGI scripts

1.1.1 Efficiency

A CGI script is handled by a particular process on the server. That means that, if the script task is a fast operation, the cost of creating and launching a new process can be a significant overhead. Servlets are handled by the server's JVM, as threads. Moreover, if N clients request the same servlet, there is only one copy of the servlet class in the JVM memory.

1.1.2 Power

Servlets can talk directly to the web server. That makes things easier when it comes to look up images and other data. Servlets can share data among each other, allowing database connection pools creation for example. They also have built-in session tracking tools (see 8 and 9).

1.1.3 Portability

The servlet API is standardized, so servlet code is instantly portable to any web server that supports them, which is the case of all the major servers nowadays (Apache, IIS, i-Planet,...)

1.2 Main servlet actors

The fig.8 lists the major classes and interfaces involved in writing servlets, together with their most widely used methods. Some of the methods mentioned in the class boxes are actually inherited from base classes/interfaces, but are placed here for easier understanding of the classes responsibilities.

1.3 Servlet life-cycle

The servlets life-cycle is similar to that of the applets:

- The `init()` method is responsible for initializing various specific properties. There is a default `init()` method, so it does not need to be redefined.
- Conversely, the `destroy()` method is in charge of releasing all resources the servlet could have acquired.
- Between the two, one of the various "service" methods (most commonly, `doGet()` or `doPost()`) is called each time a client request is sent to the server.

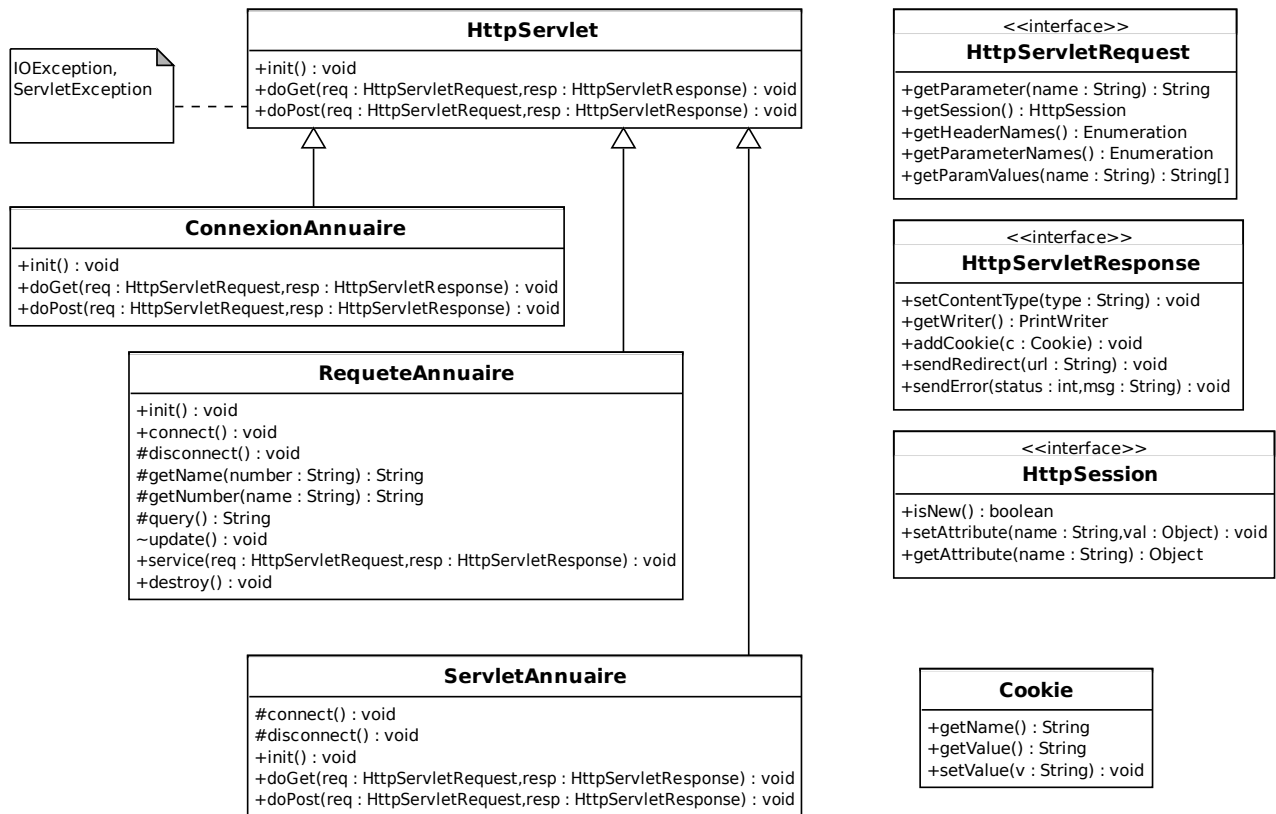


Figure 1: Main servlet actors

1.4 GET and POST methods

Those two methods differ by the way they pass client page parameters to the server:

- the GET method uses the URL to give *name/value* pairs to the server;
- the POST method uses request headers to achieve the same goal.

The servlet API encapsulates those behaviors, so most of the time servlet programmers only have to write one of the `doGet()` and `doPost()` methods, and make the other one call it.

1.5 Basic servlet structure

Most of the time, the only method that will have to be refined is `doGet()` (and/or `doPost()`). The classical scheme is as follows:

```

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

public class SomeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        // read incoming HTTP headers (ex. cookies)
        // and HTML form data

        // use "response" to specify the response headers

        PrintWriter out = response.getWriter();
    }
}

```

```

        // use ‘‘out’’ to build the actual page
        // and send it back to the client
    }
}

```

2 Defining and installing a servlet on the J2EE platform

The description below pertains to a J2EE server. As to fall’2008, such a server runs at `http://10.4.106.12:8080` `http://10.4.106.12:8080`. The owner is `etudiant`, it can be started using `~j2ee/bin/asadmin start-domain domain1`. The admin port is 4848.

The servlet installation process uses the `deploytool` command, a program distributed with the J2EE server itself. It follows those steps:

1. Compile all needed source files
2. Create an empty directory with the following structure: `myApp/{images,WEB-INF/{classes,lib}}`
3. html and JSP files are at the root. There should be a `index.{html,jsp}` there. Classes belonging to packages are in (subdirs of) `WEB-INF/classes`. Supplementary JAR files are placed in `lib`. For instance, `mysql.jar` for database applications.
4. Create `web.xml` in `WEB-INF` (see below)
5. From the root of the hierarchy, run `jar cvf ~/myApp.jar *`
6. To deploy, context root must not have a leading “/” (ex. `user/appli`). *Beware*: if the descriptor mentions `/myServlet`, then the servlet will be launched at URL `http://theServer:8080/user/appli/myServlet`

When you want to update your application (for instance, to add a new servlet), you must first delete the `war_deployed` file from `/persist/j2ee/autodeploy`, then update your local war file with the `deploytool` command, then copy it again in `/persist/j2ee/autodeploy`.

If the servlet must be directly called, then the URL will be of the form `http://theServer:1024/yourRootDir/theServletAlias` `http://theServer:1024/yourRootDir/theServletAlias`. If the servlet is (indirectly) launched by an HTML page, then the URL will be something like `http://theServer:1024/yourRootDir/thePage.html` `http://theServer:1024/yourRootDir/thePage.html`

2.1 The descriptor file, web.xml

Links the user-access URL to running servelt(s). For each servlet must be defined:

- a `servlet` tag which associates a symbol to a bytecode (a class, FQDN);
- a `servlet-mapping` tag which associates a symbol to a URL *under the context root* (chosen at deploy-time);
- all `servlet` tags must be defined prior to `servlet-mapping` tags.

Here is a sample descriptor file:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!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>

    <display-name>servlet annuaire</display-name>
    <description>
        exemple de servlet utilisant des JSP et JDBC
    </description>

    <servlet>
        <servlet-name>sname</servlet-name>
        <servlet-class>pl.j2ee.RequeteAnnuaire</servlet-class>

```

```

</servlet>

<servlet-mapping>
  <servlet-name>sname</servlet-name>
  <url-pattern>/requeteAnnuaire</url-pattern>
</servlet-mapping>

</web-app>

```

2.2 Simple example

The following example shows a servlet that displays the traditional “Hello, world!” message. The source code is in the file `HelloWWW.java`:

```

// $ Id: HelloWWW.java,v 1.1 2001/04/12 13:29:04 laroque Exp $
package fr.ucp.pl.servlets;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloWWW extends HttpServlet
{
    /**
     * Handle the HTTP GET method by building a simple web page.
     */
    public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        PrintWriter out;
        String      title = "Simple Servlet Output";

        // set content type and other response header fields first
        response.setContentType("text/html");

        // then write the data of the response
        out = response.getWriter();

        out.println("<HTML><HEAD><TITLE>");
        out.println(title);
        out.println("</TITLE></HEAD><BODY>");
        out.println("<H1>" + title + "</H1>");
        out.println("<P>Hello, world!");
        out.println("</BODY></HTML>");
        out.close();
    }
}

```

3 Reading parameters

When parameters are transmitted using GET method, they are placed after a “?” directly on the URL sent by the client to the server. The string that follows the “?” is a sequence of couples *var=val*, separated from each other by a “&”. When a POST method is used, they are transmitted on a separate line, but formatted the same way.

For example, the string “`homeDir=%7Elaroque&uid=52003`” gives two parameters, `homeDir` (value “`~laroque`”) and `uid` (value “`52003`”). Note that every non-alphanumeric character is translated into a ‘%’ followed by the hex value of the character. The spaces are replaced with “+” signs.

Of course, since extracting information from such a string is a tedious and systematic task, the servlet API provides methods (`getParameter()`, `getParameterValues()`) in class `HttpServletRequest` that make it easier.

3.1 Simple example

As an example, the following program (`ThreeParams.java`) reads three parameters sent by the client, then displays their name and value on a generated web page:

```
// $Id: ThreeParams.java 320 2003-11-05 13:38:09Z root $

package pl.servlets;

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

/** Simple servlet that reads three parameters from the form data.
 * Adapted from
 * (c) 1999 Marty Hall; may be freely used or adapted.
 */

public class ThreeParams extends HttpServlet {
    ////////////////////////////////////////////////////
    /**
     * doGet is used when parsing parameter/value pairs at the end
     * of the URL (after the "?").
     * @param request The client request
     * @param response The object returned to the client
     */
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters";
        out.println(ServletUtilities.headWithTitle(title) +
                   "<BODY>\n" +
                   "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                   "<UL>\n" +
                   " (MODIFIED)" +
                   " <LI>param1: "
                   + request.getParameter("param1") + "\n" +
                   " <LI>param2: "
                   + request.getParameter("param2") + "\n" +
                   " <LI>param3: "
                   + request.getParameter("param3") + "\n" +
                   "</UL>\n" +
                   "</BODY></HTML>");
    }

    ////////////////////////////////////////////////////
    /**
     * doPost is used when the parameters are passed in the
     * request header, not in the URL itself
     * For simple cases, the code may be the same as the servlet API
     * handles both cases the same way.
     * @param request The client request
     * @param response The object returned to the client
     */
}
```

```

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

```

3.2 Reading parameters via an HTML Form

Usually, parameters which are transmitted to the server are given by the client using an HTML form. The following example illustrates most of the classical html widgets (text fields, text areas, radio buttons, etc.) and their use. It is composed of the HTML file that collects the parameter values, and the servlet code that handles them (that is, format and re-display them):

3.2.1 the HTML page

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- $Id: PostForm.html 317 2003-11-05 13:17:48Z root $
-->
<HTML>
<HEAD>
  <TITLE>A Sample FORM using POST</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>

<FORM ACTION="/pl/showParameters"
        METHOD="POST">
  Item Number:
  <INPUT TYPE="TEXT" NAME="itemNum"><BR>
  Quantity:
  <INPUT TYPE="TEXT" NAME="quantity"><BR>
  Price Each:
  <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
  <HR>
  First Name:
  <INPUT TYPE="TEXT" NAME="firstName"><BR>
  Last Name:
  <INPUT TYPE="TEXT" NAME="lastName"><BR>
  Middle Initial:
  <INPUT TYPE="TEXT" NAME="initial"><BR>
  Shipping Address:
  <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
  Credit Card:<BR>
  &nbsp;&nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
            VALUE="Visa">Visa<BR>
  &nbsp;&nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
            VALUE="Master Card">Master Card<BR>
  &nbsp;&nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
            VALUE="Amex">American Express<BR>
  &nbsp;&nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
            VALUE="Discover">Discover<BR>
  &nbsp;&nbsp;&nbsp;<INPUT TYPE="RADIO" NAME="cardType"
            VALUE="Java SmartCard">Java SmartCard<BR>
  Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
  Repeat Credit Card Number:
  <INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
  <CENTER>
  <INPUT TYPE="SUBMIT" VALUE="Submit Order">

```

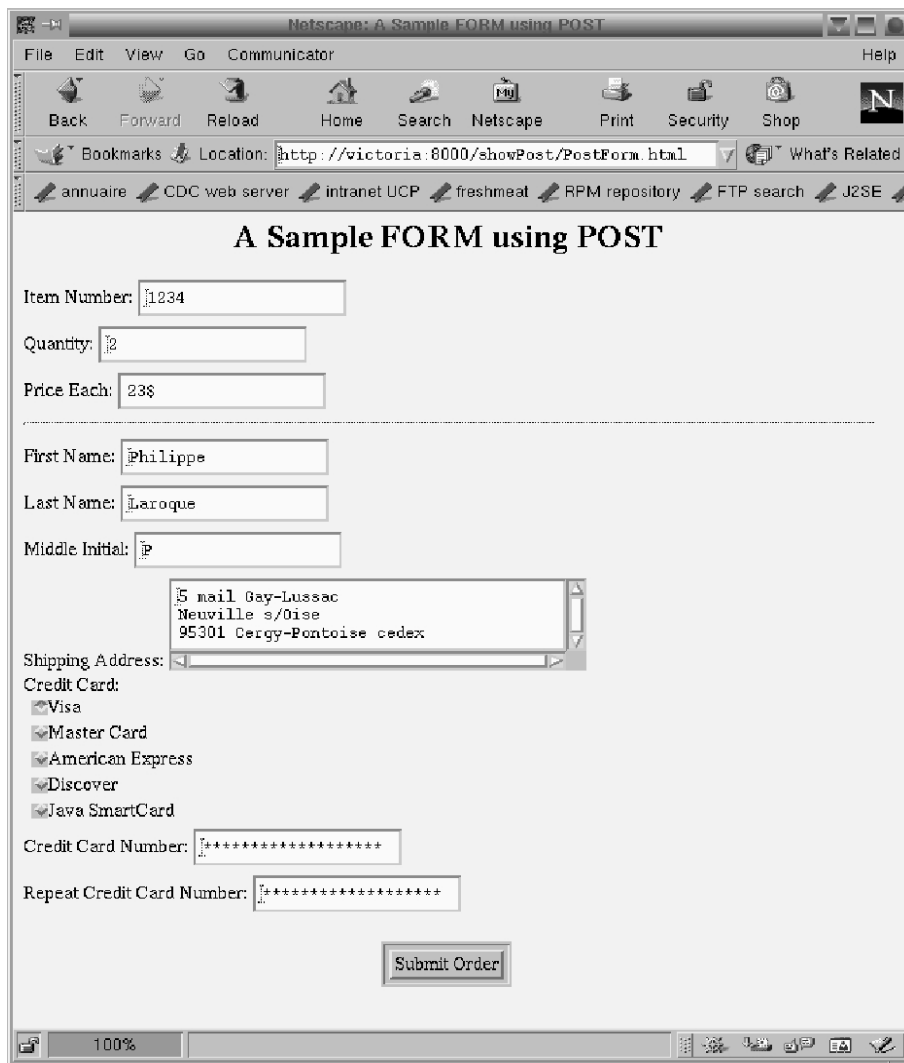


Figure 2: The form example

```

</CENTER>
</FORM>

</BODY>
</HTML>

```

The resulting page is shown on fig. 2.

3.2.2 The servlet source code

The following JAVA code is responsible for reading the parameters sent by the preceding HTML form, and displaying them into an HTML table, as shown in fig. 3.

```

// $Id: ShowParameters.java 44 2002-12-12 15:00:28Z root $

package pl.servlets;

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

/**
 * Shows all the parameters sent to the servlet via either
 * GET or POST. Specially marks parameters that have no values or

```

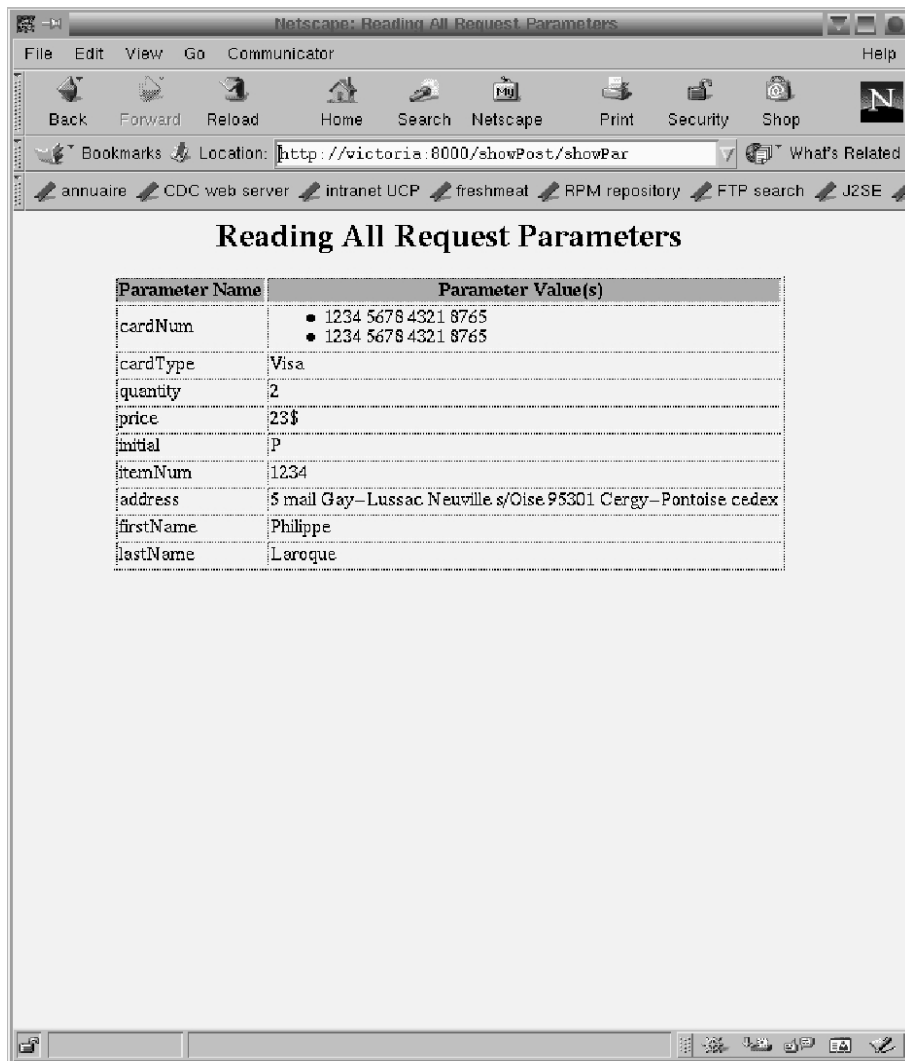


Figure 3: The generated table showing the form parameters


```

* multiple values.
* <P>
* Adapted from
* (c) 1999 Marty Hall; may be freely used or adapted.
*/

```

```

public class ShowParameters extends HttpServlet {
    ////////////////////////////////////////////////////
    /**
     * doGet is used when parsing parameter/value pairs at the end
     * of the URL (after the "?").
     * @param request The client request
     * @param response The object returned to the client
     */
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading All Request Parameters";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=#FDF5E6>\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +
            "<TR BGCOLOR=#FFA000>\n" +
            "<TH>Parameter Name<TH>Parameter Value(s)");
        Enumeration paramNames = request.getParameterNames();
        while(paramNames.hasMoreElements()) {
            String paramName = (String)paramNames.nextElement();
            out.println("<TR><TD>" + paramName + "\n<TD>");
            String[] paramValues = request.getParameterValues(paramName);
            if (paramValues.length == 1) {
                String paramValue = paramValues[0];
                if (paramValue.length() == 0)
                    out.print("<I>No Value</I>");
                else
                    out.print(paramValue);
            } else {
                out.println("<UL>");
                for(int i=0; i<paramValues.length; i++) {
                    out.println("<LI>" + paramValues[i]);
                }
                out.println("</UL>");
            }
        }
        out.println("</TABLE>\n</BODY></HTML>");
    }

    ////////////////////////////////////////////////////
    /**
     * doPost is used when the parameters are passed in the
     * request header, not in the URL itself
     * For simple cases, the code may be the same as the servlet API
     * handles both cases the same way.
     * @param request The client request
     * @param response The object returned to the client
     */
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

```
}  
}
```

4 Request headers

When a request is sent by the client, it must supply a request line (GET or POST, most of the time), but it can also send any number of headers to add information for the server to handle the request.

The servlet API supplies the `getHeader()` method of the `HttpServletRequest` class to handle this information on the server-side. Although it is sufficient to decode all kinds of headers, some of them are so commonly used that special methods have been designed, such as `getCookie()` (see 8).

The following example shows how to print all headers of a client request:

```
// $Id: ShowRequestHeaders.java 324 2003-11-05 16:19:29Z root $  
  
package pl.servlets;  
  
import java.io.*;  
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.util.*;  
  
/**  
 * Shows all the request headers sent on this  
 * particular request.  
 * <P>  
 * Adapted from  
 * (c) 1999 Marty Hall; may be freely used or adapted.  
 * @version $Id: ShowRequestHeaders.java 324 2003-11-05 16:19:29Z root $  
 */  
  
public class ShowRequestHeaders extends HttpServlet {  
    ///////////////////////////////////////////////////////////////////  
    /**  
     * shows method, URI, protocol, and all headers name/value in a table  
     * @param request contains the client request  
     * @param response builds the server response to the request  
     */  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        String title = "Servlet Example: Showing Request Headers";  
        out.println(ServletUtilities.headWithTitle(title) +  
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +  
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +  
            "<B>Request Method: </B>" +  
            request.getMethod() + "<BR>\n" +  
            "<B>Request URI: </B>" +  
            request.getRequestURI() + "<BR>\n" +  
            "<B>Request Protocol: </B>" +  
            request.getProtocol() + "<BR><BR>\n" +  
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +  
            "<TR BGCOLOR=\"#FFAD00\">\n" +  
            "<TH>Header Name<TH>Header Value");  
        Enumeration headerNames = request.getHeaderNames();  
        while(headerNames.hasMoreElements()) {  
            String headerName = (String)headerNames.nextElement();  
            out.println("<TR><TD>" + headerName);  
            out.println("    <TD>" + request.getHeader(headerName));  
        }  
    }  
}
```

```

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

////////////////////////////////////
/**
 * @see ShowRequestHeaders#doGet
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    doGet(request, response);
}
}
}

```

5 Equivalent to CGI variables

All standard CGI variables (coming from either the request line and headers, or directly from the socket, as for the IP address, or even from the server itself, as the URL mapping to actual directories) can be accessed *via* methods defined in the `HttpServletRequest` class. The following table lists the equivalence for the most widely used CGI variables (the list is not exhaustive):

<i>CGI variable</i>	<i>meaning</i>	<i>method</i>
AUTH_TYPE	authorization scheme	<code>req.getAuthType()</code>
CONTENT_LENGTH	length of data sent (POST)	<code>req.getContentLength()</code>
CONTENT_TYPE	data MIME type	<code>req.getContentType()</code>
DOCUMENT_ROOT	<code>http://host/</code>	<code>getServletContext().getRealPath('/')</code>
HTTP_XXX_YYY	header access	<code>req.getHeader('Xxx_YYY')</code>
PATH_INFO	path information, useless	<code>req.getPathInfo()</code>
PATH_TRANSLATED	path info mapping, useless	<code>req.getPathTranslated()</code>
QUERY_STRING	query data string, coded	<code>req.getQueryString()</code>
REMOTE_ADDR	IP client address	<code>req.getRemoteAddr()</code>
REMOTE_HOST	client FQDN	<code>req.getRemoteHost()</code>
REMOTE_USER	user part of authorization	<code>req.getRemoteUser()</code>
REQUEST_METHOD	method used	<code>req.getMethod()</code>
SCRIPT_NAME	path to servlet	<code>req.getServletPath()</code>
SERVER_NAME	web server name	<code>req.getServerName()</code>
SERVER_PORT	port server is listening on	<code>req.getServerPort()</code>
SERVER_PROTOCOL	protocol name and version	<code>req.getProtocol()</code>
SERVER_SOFTWARE	web server info	<code>getServletContext().getServerInfo()</code>

The following servlet code displays several CGI variables values in a generated web page:

```

// $Id: ShowCGIVariables.java 316 2003-11-03 13:56:46Z root $

package pl.servlets;

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

/**
 * Creates a table showing the values of all the CGI variables.
 * <P>
 * Adapted from
 * (c) 1999 Marty Hall; may be freely used or adapted.
 */

```

```

public class ShowCGIVariables extends HttpServlet {
    ////////////////////////////////////////////////////
    /**
     * shows most usual CGI variables
     */
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String[] [] variables =
            { { "AUTH_TYPE", request.getAuthType() },
              { "CONTENT_LENGTH", String.valueOf(request.getContentLength()) },
              { "CONTENT_TYPE", request.getContentType() },
              { "DOCUMENT_ROOT", getServletContext().getRealPath("/") },
              { "PATH_INFO", request.getPathInfo() },
              { "PATH_TRANSLATED", request.getPathTranslated() },
              { "QUERY_STRING", request.getQueryString() },
              { "REMOTE_ADDR", request.getRemoteAddr() },
              { "REMOTE_HOST", request.getRemoteHost() },
              { "REMOTE_USER", request.getRemoteUser() },
              { "REQUEST_METHOD", request.getMethod() },
              { "SCRIPT_NAME", request.getServletPath() },
              { "SERVER_NAME", request.getServerName() },
              { "SERVER_PORT", String.valueOf(request.getServerPort()) },
              { "SERVER_PROTOCOL", request.getProtocol() },
              { "SERVER_SOFTWARE", getServletContext().getServerInfo() }
            };
        String title = "Servlet Example: Showing CGI Variables";
        out.println(ServletUtilities.headWithTitle(title) +
                   "<BODY BGCOLOR=#FDF5E6>\n" +
                   "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
                   "<TABLE BORDER=1 ALIGN=CENTER>\n" +
                   "<TR BGCOLOR=#FFAD00>\n" +
                   "<TH>CGI Variable Name<TH>Value");
        for(int i=0; i<variables.length; i++) {
            String varName = variables[i][0];
            String varValue = variables[i][1];
            if (varValue == null)
                varValue = "<I>Not specified</I>";
            out.println("<TR><TD>" + varName + "<TD>" + varValue);
        }
        out.println("</TABLE></BODY></HTML>");
    }

    ////////////////////////////////////////////////////
    /**
     * @see ShowCGIVariables#doGet
     */
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

6 Status line codes

The typical response from a server to a client request consists of:

- a status line;
- some response headers;
- a blank line;
- the actual document returned back to the client.

A simple example is given below:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello, world
```

The status line includes the HTTP version, a status code and a short message that is mapped to the status code.

All headers except `Content-Type` are optional.

Servlets can manipulate the status line (see below) and response headers (see 7).

6.1 Standard status codes

Since the small error message is bound to the status code, and the protocol version is determined by the server, the only thing the servlet needs to do is set the status code.

The following table lists several standard HTTP 1.1 status codes and their meaning. The list is not exhaustive.

status code	message	meaning
100	continue	continue partial request
101	switch protocole	change to different protocol
200	OK	default for servlets
201	created	server created a document
202	accepted	processing not complete
204	no content	no new document
301	moved permanently	document URL is indicated in Location header
302	found	moved temporarily (see <code>sendRedirect()</code>)
400	bad request	syntax error in request
401	unauthorized	access attempt to passwd-protected page
403	forbidden	resource is not available
404	not found	no such page (see <code>sendError()</code>)
405	method not allowed	
406	not acceptable	generated MIME type incompatible with client
407	proxy authentication required	proxy server must return a Proxy-Authenticate header
408	request timeout	
500	internal server error	
501	not implemented	server can't fulfill request
503	service unavailable	server maintenance or overloading

7 Response headers

When the status line indicates an error, it is common to send additional information to help the client understand the cause of the error. This additional information can be sent back to the client *via* one or more additional response headers.

The servlet API supply a standard way to specify response headers, it is the `setHeader()` method of the `HttpServletResponse` class. It takes two strings as arguments, the header name and its value.

The following table shows the most common response headers and their meaning:

header name	meaning
Allow	supported request methods
Content-Encoding	used encoding method for the document
Content-Length	document length in bytes
Content-Type	the MIME type of the document
Date	current time (see <code>setDateHeader()</code>)
Expires	time when document will no more be cached
Last-Modified	useful to update or not the page
Location	new location for requested document (code 302)
Refresh	time before asking for an updated page
Server	the server the page comes from
Set-Cookie	servlets use <code>addCookie()</code> (see 8)
WWW-Authenticate	required for responses with code 401

8 Cookies

Cookies are small bits of textual information that servers send to clients, and that client browsers can send back unchanged to servers when needed.

That is useful in a number of situations:

- identifying a user during a multi-page session;
- avoiding username and password;
- customizing responses.

Cookies can be disabled from client browsers, so they should not be the only technique the server relies on (see 9 for an alternative). Nevertheless, they are a convenient way to have a basically stateless protocol behave as a stateful one when needed, even if they should not be used when security issues are of high importance.

The servlet API supplies the `Cookie` class (see fig. 1) and a bunch of tool methods to easily manipulate cookies. A cookie is created with its name and value given as constructor arguments. It can be added to a response via the `addCookie()` method of the `HttpServletResponse` class. The `getCookies()` method of the `HttpServletRequest` class returns an array of cookies associated with the client issuing the request. Finally, the `getName()` and `getValue()` methods of the `Cookie` class can be used to retrieve the corresponding information. For each cookie attribute listed in fig. 1, there is a couple of `get/set` methods.

9 Session tracking

Aside from the cookie technique described above, there are two classical ways to make the server “remember” who the client is from one request to another:

1. URL rewriting. The session-related information is passed *via* the URL of the request. The advantage is that this technique works with the wider set of browsers, they don’t have to support anything like cookies and so on. However, like cookies, it is not very secure and the server has a lot of information extraction work to do.
2. Hidden form fields. a special HTML entry, which looks like `<INPUT TYPE=’’HIDDEN’’ NAME=’’session’’ VALUE=’’...’’>`. The problem here is that it only works with dynamically-generated web pages.

Servlets provide another solution, *via* a special object called `HttpSession`. This class encapsulates the session tracking technique under a uniform API, making the use of cookies, URL-rewriting or whatever transparent. Each request can have an associated session, which can be retrieved (and, if there is none, created) via the `getSession()` method of the `HttpServletRequest` class.

Each session can be given an arbitrary number of *key/value* pairs, *via* the `getAttribute()` / `setAttribute()` methods. The following piece of code illustrates the use of such objects:

```
// retrieve current session and create one if none
HttpSession session = request.getSession(true);

// retrieve current bought items
```

```

ShoppingCart items = (ShoppingCart) session.getValue('items');

// some items have already been bought
if (items != null) {
    doSomethingWith(items);
} else {
    // no items bought until now:
    // create an empty cart
    items = new ShoppingCart(...);
    doSomethingElseWith(items);
}

```

10 Connecting a servlet to a (distant) Database

Using JDBC, a servlet easily can connect to a database. In a simple case (but this is not required), the database server can be the same machine as the j2ee platform server.

The servlet opens the same kind of connection as for an ordinary application, as the following example shows:

```

// $Id: ServletAnnuaire.java 325 2003-11-05 16:21:12Z root $

package pl.servlets;

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

/**
 * Illustration of servlet connection to a DB.
 * <p>
 * BEWARE: the j2ee server needs to access the JDBC driver classes, so
 * the jar file must be included in the web component.
 * <p>
 * @author Philippe.Laroque@dept-info.u-cergy.fr
 * @version $Id: ServletAnnuaire.java 325 2003-11-05 16:21:12Z root $
 */
public class ServletAnnuaire extends HttpServlet {

    Connection c;
    Statement instr;
    String driver, URL;

    /** instantiate a connection from the driver
     * and creates a statement.
     */
    protected void connect() {
        try {
            c = DriverManager.getConnection(URL);
            instr = c.createStatement();
        } catch (Exception e) {
            System.err.println("connection failed " + e.getMessage());
            if (c != null) try { c.close(); } catch (Exception ee) {}
            //System.exit(2);
        }
    }

    /** close both statement and connection */

```

```

protected void disconnect() {
    try {
        instr.close();
        c.close();
    } catch (Exception e) {System.err.println("disconnection failed " + e.getMessage());}
}

////////////////////////////////////
/**
 * Servlet initialization. <p>
 * Driver loading is made during servlet initialization
 * and not for each connection.
 */
public void init() throws ServletException {
    try {
        driver = "com.mysql.jdbc.Driver";
        URL = "jdbc:mysql://10.4.101.229/test?user=etudiant";
        c = null;
        Statement instr = null;
        Class.forName(driver).newInstance();
    } catch (Exception e) {
        System.err.println("driver load failed : " + e.getMessage());
    }
}

/** generate the answer page, containing either the searched number
 * or an error message.
 * @param request incoming client request
 * @param response to send back the built page
 */
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String nom = request.getParameter("nom");

    out.print("<HTML><TITLE> Reponse annuaire </TITLE>" +
              "<BODY BGCOLOR=#FDF5E6>\n" +
              "<H1 ALIGN=CENTER> reponse annuaire :</H1>\n" +
              "Le numero de " + nom);

    connect();    // connexion a la base pour la requete

    try {
        // recherche du numero correspondant
        ResultSet res = instr.executeQuery("SELECT NUM FROM tableTest WHERE NOM=\""
                                          + nom + "\"");

        if (res.next()) // trouve !
            out.println(" est " + res.getString("NUM"));
        else // pas trouve
            out.println(" n'est pas dans l'annuaire");
    } catch (Exception e) {
        System.err.println("select failed: " + e.getMessage());
    }
    finally { disconnect(); } // deconnexion systematique

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

/** @see ServletAnnuaire#doGet */

```



```

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

```

The only thing to remember is that the JAR file containing the classes that implement the JDBC driver (here, `mysql.jar`) must be included in the servlet deployment war file. The entry HTML page looks like the following:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- $Id: annuaire.html 44 2002-12-12 15:00:28Z root $
-->
<HTML>
<HEAD>
  <TITLE>Annuaire</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">Annuaire</H1>

<FORM ACTION="/pl/annuaire">
  <CENTER>
    nom:
    <INPUT TYPE="TEXT" NAME="nom"><BR>
    <BR>
    <INPUT TYPE="SUBMIT" VALUE="OK">
  </CENTER>
</FORM>

</BODY>
</HTML>

```

11 Introduction to the JSP

Java Server Pages (JSP) allow to separate the dynamic part of the pages from the static HTML. This is achieved *via* some directives that are directly included in classical HTML code. An HTML page with such directives (which is usually named with a `.jsp` extension) can then be compiled into a servlet by the server, the first time the page is requested by a client.

Most of the directives appear in the HTML page as special tags which start with “<%” and end with “%>”, as in the example below, which supposes the request URL is something like `http://host/order.jsp?title=introduction+to+th`

```

...
Thanks for ordering
<I><%= request.getParameter('title') %></I>

```

A typical JSP file is composed of four types of elements:

1. Ordinary HTML code
2. scripting elements
3. directives
4. actions

Each of the last three is now described in some details.

11.1 Scripting elements

There are three scripting elements:

1. *expressions*, which are introduced by “<%=”, are evaluated and placed in the output, as in the example above;
2. *scriptlets*, which are introduced by “<%” and contain java code which is inserted in the `service()` method of the generated servlet;
3. *declarations*, which begin with “<!” and contain java code which is inserted in the servlet class, outside of `service()` method.

To simplify the programmer’s task, a number of predefined variables are available, among which `request` (as in the above example), `response`, `session`, `out...`

Some more (simple) examples:

```
...
<% String queryData = request.getQueryString();
out.println(‘‘Attached GET data : ‘‘ + queryData);
%>
...
<%! private int accessCount = 0; %>
accesses to this page since last server reboot:
<%= ++accessCount %>
```

11.2 Directives

Directives let the programmer control the servlet structure. A directive usually associates one attribute to a value, but can handle several attribute / value pairs.

There are two main types of directives:

1. the `page` directive, which allows to import classes, customize servlet superclass etc.
2. the `include` directive, which inserts a file just before the jsp file is transformed into a servlet.

11.2.1 The page directive

This directive lets the programmer define several case-sensitive attributes. Its syntax is

```
<% page attr=’’value’’ %>
```

Some of the most widely used attributes are listed below:

- `import=’’package.class’’`, to import one or more classes / packages (if multiple imports, separate them with a “,”)
- `contentType=’’MIME-type’’`, to specify another type (default is text/html)
- `isThreadSafe=’’false’’` (default true), to allow a separate servlet instance creation for each request (or serial request handling)
- `session=’’false’’` (default true), not to create any Session object.
- `buffer=’’sizekb|none’’`, the out buffer size (must be at least 8kb)
- `autoflush=’’false’’` (default true), not to flush the buffer when full.
- `extends=’’package.class’’`, specifies the superclass of the generated servlet.
- `info=’’message’’`, specifies the string that is to be returned by the `getServletInfo()` method
- `errorPage=’’url’’`, specifies a jsp file that should process any `Throwable` not caught in current page.
- `isErrorPage=’’true’’` (default false), specifies that the current page can serve as error page for other JSP.

11.2.2 The include directive

This directive lets the programmer insert whatever file is needed just before the translation of the JSP into a servlet.

The syntax is

```
<%@ include file='relativeURL' %>
```

This is a convenient way to insert homogeneous elements into several pages (for instance, navigation bars). Notice however that this insertion is *not dynamic*, as when the included file changes, the JSP have to be recompiled into servlets to reflect the changes.

11.3 Actions

JSP actions use XML-like constructs to control the behavior of the servlet engine. For instance, the following actions are recognized:

- `jsp:include` – include a file at each request of the page (dynamically)
- `jsp:useBean` – find or instantiate a javaBean
- `jsp:setProperty` – set a javaBean property
- `jsp:getProperty` – insert the javaBean property in the output
- `jsp:forward` – forward the requester to a new page
- `jsp:plugin` – generate browser-specific code that makes an OBJECT or EMBED tag for the java plugin

All of these actions, and the values associated, are case-sensitive (as is XML). Some simple examples:

```
<jsp:include page='mostRecentNews.html' flush='true' />

<jsp:useBean id='test' class='fr.ucp.pl.beans.Example' />
<jsp:setProperty name='test' property='message' value='Hello, world' />
<H2>Message: <I>
<jsp:getProperty name='test' property='message' />
</I></H2>

<jsp:forward page='/utils/errorReporter.jsp' />
<jsp:forward page='<%= someJavaExpression %>' />
```

11.4 Comments in JSP

The table below shows some constructs that can be used in various situations to insert comments:

Syntax	Purpose
<code>!%– comment –%i</code>	a JSP comment
<code>!– comment –i</code>	an HTML comment
<code>i\%</code>	used in static HTML when we want “i%”
<code>%\i</code>	used in static HTML when we want “%i”
<code>\'</code>	a single quote in an attribute delimited by single quotes
<code>\”</code>	a double quote in an attribute delimited by double quotes
<code>%\i</code>	%i in an attribute
<code>i\%</code>	i% in an attribute

Contents

1	Basic principles	1
1.1	Advantage of servlets over CGI scripts	1
1.1.1	Efficiency	1
1.1.2	Power	1
1.1.3	Portability	1
1.2	Main servlet actors	1
1.3	Servlet life-cycle	1
1.4	GET and POST methods	2
1.5	Basic servlet structure	2
2	Defining and installing a servlet on the J2EE platform	3
2.1	The descriptor file, web.xml	3
2.2	Simple example	4
3	Reading parameters	4
3.1	Simple example	5
3.2	Reading parameters via an HTML Form	6
3.2.1	the HTML page	6
3.2.2	The servlet source code	7
4	Request headers	10
5	Equivalent to CGI variables	11
6	Status line codes	12
6.1	Standard status codes	13
7	Response headers	13
8	Cookies	14
9	Session tracking	14
10	Connecting a servlet to a (distant) Database	15
11	Introduction to the JSP	17
11.1	Scripting elements	18
11.2	Directives	18
11.2.1	The page directive	18
11.2.2	The include directive	19
11.3	Actions	19
11.4	Comments in JSP	19