# Introduction à la programmation WEB en JAVA
## - Servlets et pages JSP -

serge.tahe@istia.univ-angers.fr
septembre 2002

# Introduction

Ce document est un support de cours : ce n'est pas un cours complet. Des approfondissements nécessitent l'aide d'un enseignant et par ailleurs un certain nombre de thèmes n'ont pas été abordés.

L'étudiant y trouvera cependant des informations lui permettant la plupart du temps de travailler seul. Il comporte peut-être encore des erreurs : toute suggestion constructive est la bienvenue à l'adresse *serge.tahe@istia.univ-angers.fr*.

1. Programmation j2EE aux éditions Wrox et distribué par Eyrolles
2. Java Server Pages (Fields, Kolb) aux éditions Eyrolles
3. Java Servlet Programming (Hunter) aux éditions O'Reilly

Beaucoup d'informations de ce polycopié sont tirées de la référence 1 dont le contenu (1000 pages) dépasse très largement celui de de polycopié.

S'il est suffisant pour une introduction à la programmation web en Java, ce document pourrait être amélioré en y ajoutant les thèmes importants suivants :

- programmation XML
- balises JSP
- EJB (Enterprise Java Beans)

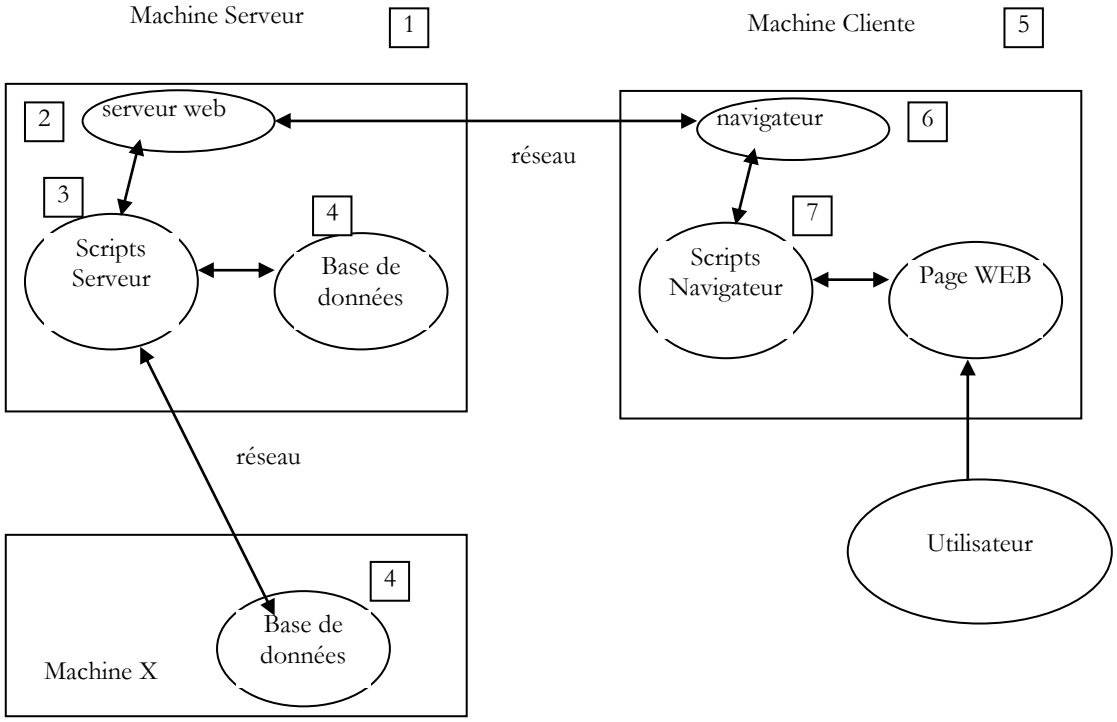
Serge Tahé

Septembre 2002

# 1. Les bases

Dans ce chapitre, nous présentons les bases de la programmation Web. Il a pour but essentiel de faire découvrir les grands principes de la programmation Web avant de mettre ceux-ci en pratique avec un langage et un environnement particuliers. Il présente de nombreux exemples qu'il est conseillé de tester afin de "s'imprégner" peu à peu de la philosophie du développement web.

## 1.1 Les composantes d'une application Web

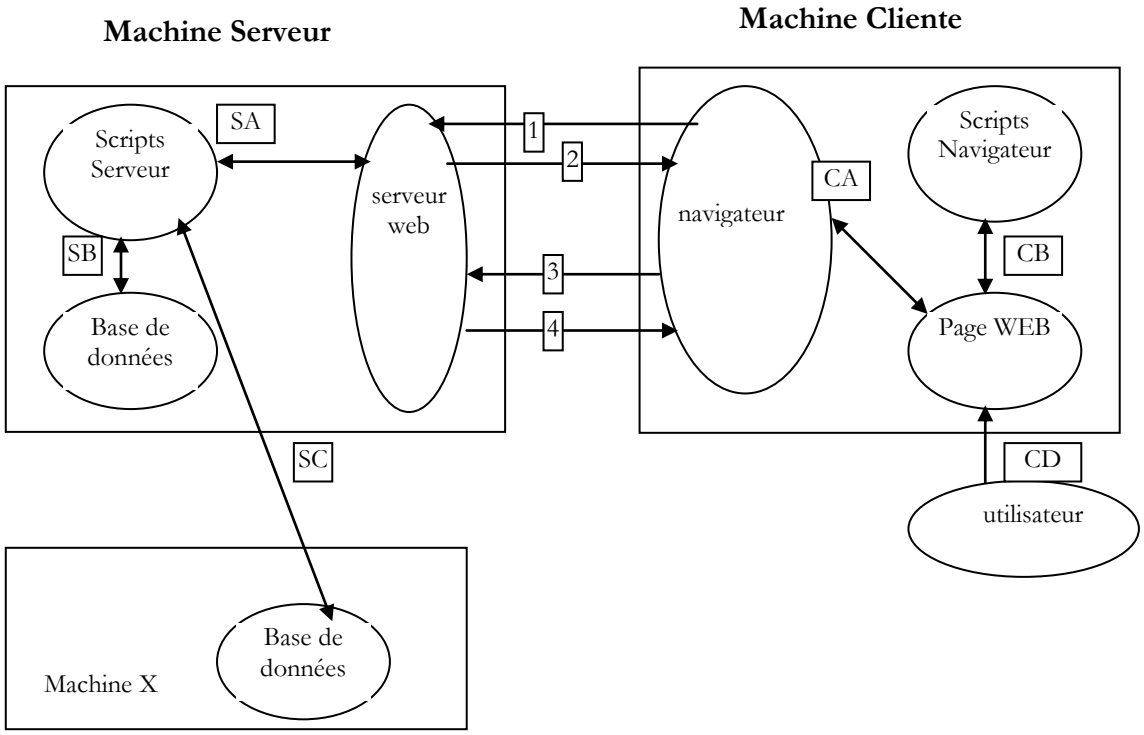



| Numéro | Rôle | Exemples courants |
|---|---|---|
| 1 | OS Serveur | Linux, Windows |
| 2 | Serveur Web | Apache (Linux, Windows)<br>IIS (NT), PWS(Win9x) |
| 3 | Scripts exécutés côté serveur. Ils peuvent l'être par des modules du serveur ou par des programmes externes au serveur (CGI). | PERL (Apache, IIS, PWS)<br>VBSCRIPT (IIS,PWS)<br>JAVASCRIPT (IIS,PWS)<br>PHP (Apache, IIS, PWS)<br>JAVA (Apache, IIS, PWS)<br>C#, VB.NET (IIS) |
| 4 | Base de données - Celle-ci peut être sur la même machine que le programme qui l'exploite ou sur une autre via Internet. | Oracle (Linux, Windows)<br>MySQL (Linux, Windows)<br>Access (Windows)<br>SQL Server (Windows) |
| 5 | OS Client | Linux, Windows |
| 6 | Navigateur Web | Netscape, Internet Explorer |
| 7 | Scripts exécutés côté client au sein du navigateur. **Ces scripts n'ont aucun accès aux** | VBscript (IE) |

**disques du poste client**.

Javascript (IE, Netscape)
Perlscript (IE)
Applets JAVA

## 1.2 Les échanges de données dans une application web avec formulaire




| Numéro | Rôle |
|---|---|
| 1 | Le navigateur demande une URL pour la 1ère fois *(http://machine/url)*. Auncun paramètre n'est passé. |
| 2 | Le serveur Web lui envoie la page Web de cette URL. Elle peut être statique ou bien dynamiquement générée par un script serveur (SA) qui a pu utiliser le contenu de bases de données (SB, SC). Ici, le script détectera que l'URL a été demandée sans passage de paramètres et génèrera la page WEB initiale.<br>Le navigateur reçoit la page et l'affiche (CA). Des scripts côté navigateur (CB) ont pu modifier la page initiale envoyée par le serveur. Ensuite par des interactions entre l'utilisateur (CD) et les scripts (CB) la page Web va être modifiée. Les formulaires vont notamment être remplis. |
| 3 | L'utilisateur valide les données du formulaire qui doivent alors être envoyées au serveur web. Le navigateur redemande l'URL initiale ou une autre selon les cas et transmet en même temps au serveur les valeurs du formulaire. Il peut utiliser pour ce faire deux méthodes appelées GET et POST. A réception de la demande du client, le serveur déclenche le script (SA) associé à l'URL demandée, script qui va détecter les paramètres et les traiter. |
| 4 | Le serveur délivre la page WEB construite par programme (SA, SB, SC). Cette étape est identique à l'étape 2 précédente. Les échanges se font désormais selon les étapes 2 et 3. |

## 1.3 Quelques ressources

Ci-dessous on trouvera une liste de ressources permettant d'installer et d'utiliser certains outils permettant de faire du développement web. On trouvera en annexe, une aide à l'installation de ces outils.

`Serveur Apache` *`http://www.apache.org`*

- Apache, Installation et Mise en œuvre, O'Reilly

`Serveur IIS, PWS` *http://www.microsoft.com*

`PERL` *http://www.activestate.com*
- Programmation en Perl, Larry Wall, O'Reilly
- Applications CGI en Perl, Neuss et Vromans, O'Reilly
- la documentation HTML livrée avec Active Perl

`PHP` *http://www.php.net*
- Prog. Web avec PHP, Lacroix, Eyrolles
- Manuel d'utilisation de PHP récupérable sur le site de PHP

`VBSCRIPT, ASP` *http://msdn.microsoft.com/scripting/vbscript/download/vbsdoc.exe*
*http://msdn.microsoft.com/scripting/default.htm?/scripting/vbscript*
- Interface entre WEB et Base de données sous WinNT, Alex Homer, Eyrolles

`JAVASCRIPT` *http://msdn.microsoft.com/scripting/jscript/download/jsdoc.exe*
*http://developer.netscape.com/docs/manuals/index.html*

`HTML` *http://developer.netscape.com/docs/manuals/index.html*

`JAVA` *http://www.sun.com*
- JAVA Servlets, Jason Hunter, O'Reilly
- Programmation réseau avec Java, Elliotte Rusty Harold, O'Reilly
- JDBC et Java, George Reese, O'reilly

`Base de données` *http://www.mysql.com*
*http://www.oracle.com*
- Le manuel de MySQL est disponible sur le site de MySQL
- Oracle 8i sous Linux, Gilles Briard, Eyrolles
- Oracle 8i sous NT, Gilles Briard, Eyrolles

## 1.4 Notations

Dans la suite, nous supposerons qu'un certain nombre d'outils ont été installés et adopterons les notations suivantes :

| notation | signification |
|---|---|
| `<apache>` | racine de l'arborescence du serveur apache |
| `<apache-DocumentRoot>` | racine des pages Web délivrées par Apache. C'est sous cette racine que doivent se trouver les pages Web. Ainsi l'URL *http://localhost/page1.htm* correspond au fichier *<apache-DocumentRoot>\page1.htm*. |
| `<apache-cgi-bin>` | racine de l'arborescence lié à l'alias *cgi-bin* et où l'on peut placer des scripts CGI pour Apache. Ainsi l'URL *http://localhost/cgi-bin/test1.pl* correspond au fichier *<apache-cgi-bin>\test1.pl*. |
| `<pws-DocumentRoot>` | racine des pages Web délivrées par PWS. C'est sous cette racine que doivent se trouver les pages Web. Ainsi l'URL *http://localhost/page1.htm* correspond au fichier *<pws-DocumentRoot>\page1.htm*. |
| `<perl>` | racine de l'arborescence du langage Perl. L'exécutable *perl.exe* se trouve en général dans *<perl>\bin*. |
| `<php>` | racine de l'arborescence du langage PHP. L'exécutable *php.exe* se trouve en général dans *<php>*. |
| `<java>` | racine de l'arborescence de java. Les exécutables liés à java se trouvent dans *<java>\bin*. |
| `<tomcat>` | racine du serveur Tomcat. On trouve des exemples de servlets dans *<tomcat>\webapps\examples\servlets* et des exemples de pages JSP dans *<tomcat>\webbapps\examples\jsp* |

On se reportera pour chacun de ces outils à l'annexe qui donne une aide pour leur installation.

## 1.5 Pages Web statiques, Pages Web dynamiques

Une page statique est représentée par un fichier HTML. Une page dynamique est, elle, générée "à la volée" par le serveur web. Nous vous proposons dans ce paragraphe divers tests avec différents serveurs web et différents langages de programmation afin de montrer l'universalité du concept web.

### 1.5.1 Page statique HTML (HyperText Markup Language)

Considérons le code HTML suivant :

```
<html>
  <head>
    <title>essai 1 : une page statique</title>
   </head>
   <body>
     <center>
     <h1>Une page statique...</h1>
   </body>
</html>
```

qui produit la page web suivante :


Les tests

- lancer le serveur Apache
- mettre le script *essai1.html* dans *<apache-DocumentRoot>*
- visualiser l'URL *http://localhost/essai1.html* avec un navigateur
- arrêter le serveur Apache

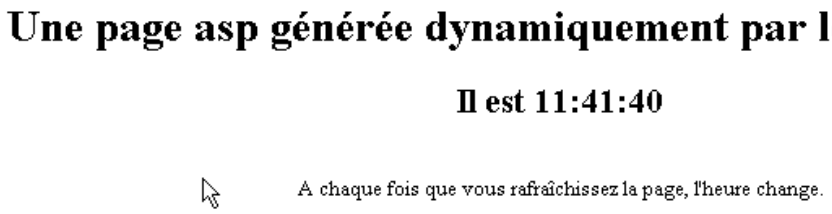
- lancer le serveur PWS
- mettre le script *essai1.html* dans *<pws-DocumentRoot>*
- visualiser l'URL *http://localhost/essai1.html* avec un navigateur

### 1.5.2 Une page ASP (Active Server Pages)

Le script essai2.asp :

```
<html>
  <head>
    <title>essai 1 : une page asp</title>
   </head>
   <body>
     <center>
     <h1>Une page asp générée dynamiquement par le serveur PWS</h1>
     <h2>Il est <% =time %></h2>
     <br>
     A chaque fois que vous rafraîchissez la page, l'heure change.
   </body>
</html>
```

produit la page web suivante :




Le test

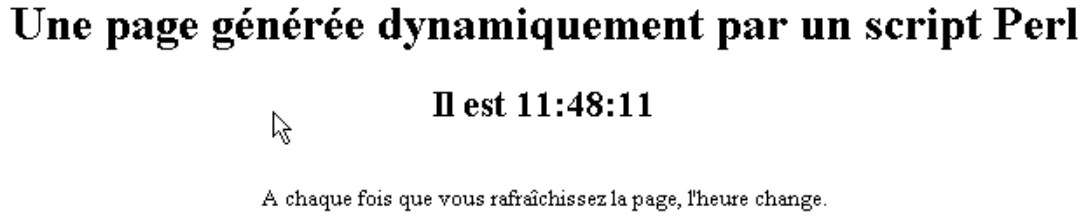
- lancer le serveur PWS
- mettre le script *essai2.asp* dans <*pws-DocumentRoot*>
- demander l'URL *http://localhost/essai2.asp* avec un navigateur

## 1.5.3 Un script PERL (Practical Extracting and Reporting Language)

Le script essai3.pl :

```
#!d:\perl\bin\perl.exe

($secondes,$minutes,$heure)=localtime(time);

print <<HTML
Content-type: text/html

<html>
  <head>
    <title>essai 1 : un script Perl</title>
   </head>
   <body>
     <center>
     <h1>Une page générée dynamiquement par un script Perl</h1>
     <h2>Il est $heure:$minutes:$secondes</h2>
     <br>
     A chaque fois que vous rafraîchissez la page, l'heure change.
   </body>
</html>

HTML
;
```

La première ligne est le chemin de l'exécutable *perl.exe*. Il faut l'adapter si besoin est. Une fois exécuté par un serveur Web, le script produit la page suivante :




Le test

- serveur Web : Apache
- pour information, visualisez le fichier de configuration *srm.conf* ou *httpd.conf* selon la version d'Apache dans <*apache*>*confs* et rechercher la ligne parlant de *cgi-bin* afin de connaître le répertoire <*apache-cgi-bin*> dans lequel placer *essai3.pl.*
- mettre le script *essai3.pl* dans <*apache-cgi-bin*>
- demander l'url *http://localhost/cgi-bin/essai3.pl*

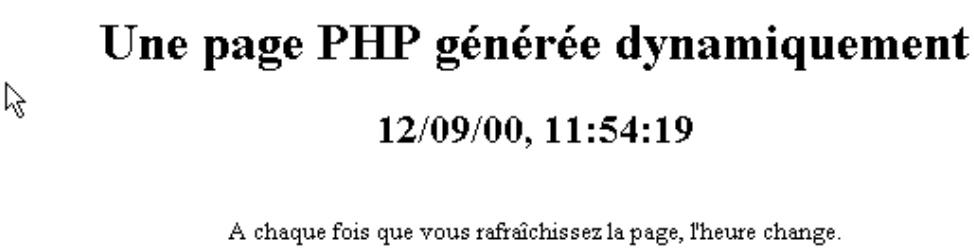
A noter qu'il faut davantage de temps pour avoir la page *perl* que la page *asp*. Ceci parce que le script Perl est exécuté par un interpréteur Perl qu'il faut charger avant qu'il puisse exécuter le script. Il ne reste pas en permanence en mémoire.

## 1.5.4 Un script PHP (Personal Home Page)

Le script essai4.php

```
<html>
  <head>
    <title>essai 4 : une page php</title>
   </head>
   <body>
     <center>
     <h1>Une page PHP générée dynamiquement</h1>
     <h2>
<?
     $maintenant=time();
     echo date("j/m/y, h:i:s",$maintenant);
?>
     </h2>
     <br>
     A chaque fois que vous rafraîchissez la page, l'heure change.
   </body>
</html>
```

Le script précédent produit la page web suivante :




Les tests

- consulter le fichier de configuration *srm.conf* ou *httpd.conf* d'Apache dans <*Apache*>*confs*
- pour information, vérifier les lignes de configuration de *php*
- lancer le serveur Apache
- mettre *essai4.php* dans <*apache-DocumentRoot*>
- demander l'URL *http://localhost/essai4.php*

- lancer le serveur PWS
- pour information, vérifier la configuration de PWS à propos de php
- mettre *essai4.php* dans <*pws-DocumentRoot*>*php*
- demander l'URL *http://localhost/essai4.php*

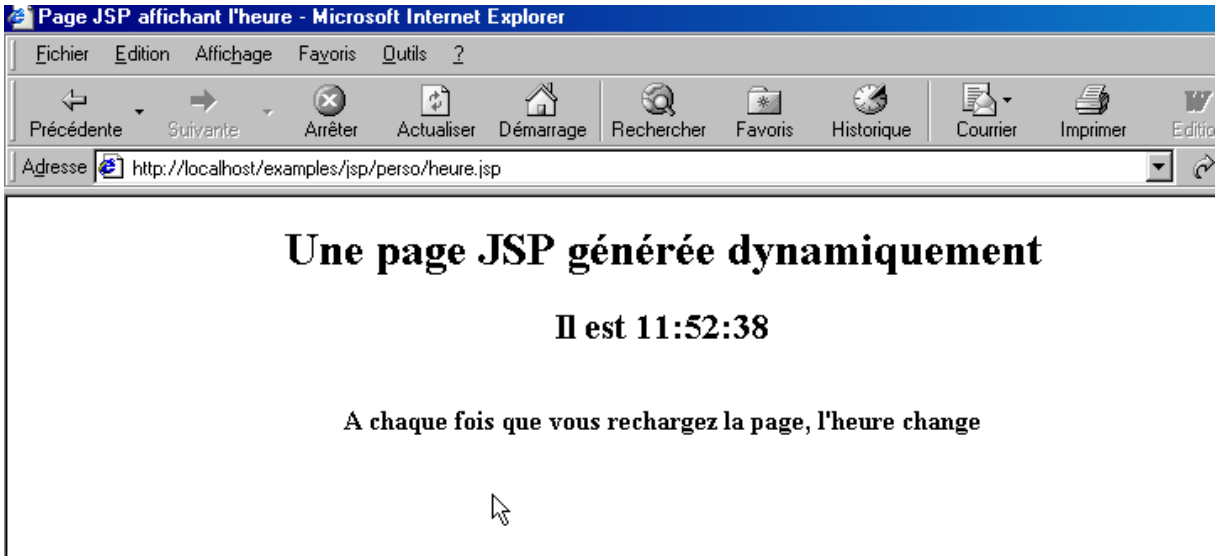
## 1.5.5 Un script JSP (Java Server Pages)

Le script heure.jsp

```
<%  //programme Java affichant l'heure %>

<%@ page import="java.util.*" %>

<%
  // code JAVA pour calculer l'heure
  Calendar calendrier=Calendar.getInstance();
  int heures=calendrier.get(Calendar.HOUR_OF_DAY);
  int minutes=calendrier.get(Calendar.MINUTE);
  int secondes=calendrier.get(Calendar.SECOND);
  // heures, minutes, secondes sont des variables globales
  // qui pourront être utilisées dans le code HTML
%>

<% // code HTML %>
<html>
  <head>
```

```
    <title>Page JSP affichant l'heure</title>
  </head>
  <body>
    <center>
    <h1>Une page JSP générée dynamiquement</h1>
    <h2>Il est <%=heures%>:<%=minutes%>:<%=secondes%></h2>
    <br>
    <h3>A chaque fois que vous rechargez la page, l'heure change</h3>
  </body>
</html>
```

Une fois exécuté par le serveur web, ce script produit la page suivante :




Les tests

- mettre le script *heure.jsp* dans *<tomcat>\jakarta-tomcat\webapps\examples\jsp* (Tomcat 3.x) ou dans *<tomcat>\webapps\examples\jsp* (Tomcat 4.x)
- lancer le serveur Tomcat
- demander l'URL *http://localhost:8080/examples/jsp/heure.jsp*

### 1.5.6 Conclusion

Les exemples précédents ont montré que :

- une page HTML pouvait être générée dynamiquement par un programme. C'est tout le sens de la programmation Web.
- que les langages et les serveurs web utilisés pouvaient être divers. Actuellement on observe les grandes tendances suivantes :
  - les tandems Apache/PHP (Windows, Linux) et IIS/PHP (Windows)
  - la technologie ASP.NET sur les plate-formes Windows qui associent le serveur IIS à un langage .NET (C#, VB.NET, ...)
  - la technologie des servlets Java et pages JSP fonctionnant avec différents serveurs (Tomcat, Apache, IIS) et sur différentes plate-formes (Windows, Linux). C'est cette denière technologie qui sera plus particulièrement développée dans ce document.

## 1.6 Scripts côté navigateur

Une page HTML peut contenir des scripts qui seront exécutés par le navigateur. Les langages de script côté navigateur sont nombreux. En voici quelques-uns :

| Langage | Navigateurs utilisables |
|---|---|
| Vbscript | IE |
| Javascript | IE, Netscape |

| | |
|---|---|
| PerlScript | IE |
| Java | IE, Netscape |

Prenons quelques exemples.

## 1.6.1 Une page Web avec un script Vbscript, côté navigateur

La page vbs1.html

```
<html>
  <head>
    <title>essai : une page web avec un script vb</title>
    <script language="vbscript">
      function reagir
        alert "Vous avez cliqué sur le bouton OK"
      end function
    </script>
   </head>

   <body>
<center>
     <h1>Une page Web avec un script VB</h1>
     <table>
       <tr>
         <td>Cliquez sur le bouton</td>
         <td><input type="button" value="OK" name="cmdOK" onclick="reagir"></td>
       </tr>
    </table>
   </body>
</html>
```

La page HTML ci-dessus ne contient pas simplement du code HTML mais également un programme destiné à être exécuté par le navigateur qui aura chargé cette page. Le code est le suivant :

```
    <script language="vbscript">
      function reagir
        alert "Vous avez cliqué sur le bouton OK"
      end function
    </script>
```

Les balises <*script*></*script*> servent à délimiter les scripts dans la page HTML. Ces scripts peuvent être écrits dans différents langages et c'est l'option *language* de la balise <*script*> qui indique le langage utilisé. Ici c'est VBScript. Nous ne chercherons pas à détailler ce langage. Le script ci-dessus définit une fonction appelée *réagir* qui affiche un message. Quand cette fonction est-elle appelée ? C'est la ligne de code HTML suivante qui nous l'indique :

```
         <input type="button" value="OK" name="cmdOK" onclick="reagir">
```

L'attribut *onclick* indique le nom de la fonction à appeler lorsque l'utilisateur cliquera sur le bouton OK. Lorsque le navigateur aura chargé cette page et que l'utilisateur cliquera sur le bouton OK, on aura la page suivante :


Les tests

Seul le navigateur IE est capable d'exécuter des scripts VBScript. Netscape nécessite des compléments logiciels pour le faire. On pourra faire les tests suivants :

- serveur Apache
- script *vbs1.html* dans <*apache-DocumentRoot*>
- demander l'url *http://localhost/vbs1.html* avec le navigateur IE

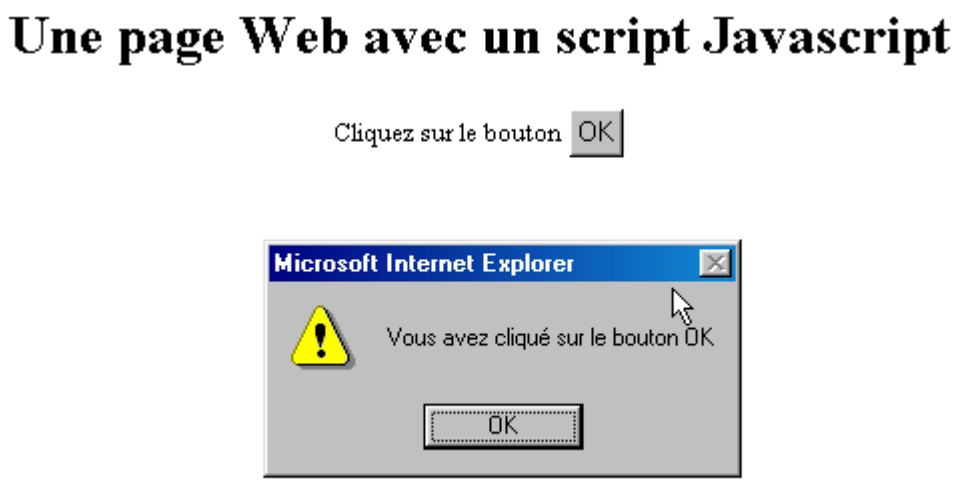
- serveur PWS
- script *vbs1.html* dans <*pws-DocumentRoot*>
- demander l'url *http://localhost/vbs1.html* avec le navigateur IE

## 1.6.2 Une page Web avec un script Javascript, côté navigateur

La page : js1.html

```
<html>
  <head>
    <title>essai 4 : une page web avec un script Javascript</title>
    <script language="javascript">
      function reagir(){
        alert ("Vous avez cliqué sur le bouton OK");
      }
    </script>
   </head>

   <body>
     <center>
     <h1>Une page Web avec un script Javascript</h1>
     <table>
       <tr>
         <td>Cliquez sur le bouton</td>
         <td><input type="button" value="OK" name="cmdOK" onclick="reagir()"></td>
       </tr>
  </table>
   </body>
</html>
```

On a là quelque chose d'identique à la page précédente si ce n'est qu'on a remplacé le langage VBScript par le langage Javascript. Celui-ci présente l'avantage d'être accepté par les deux navigateurs IE et Netscape. Son exécution donne les mêmes résultats :



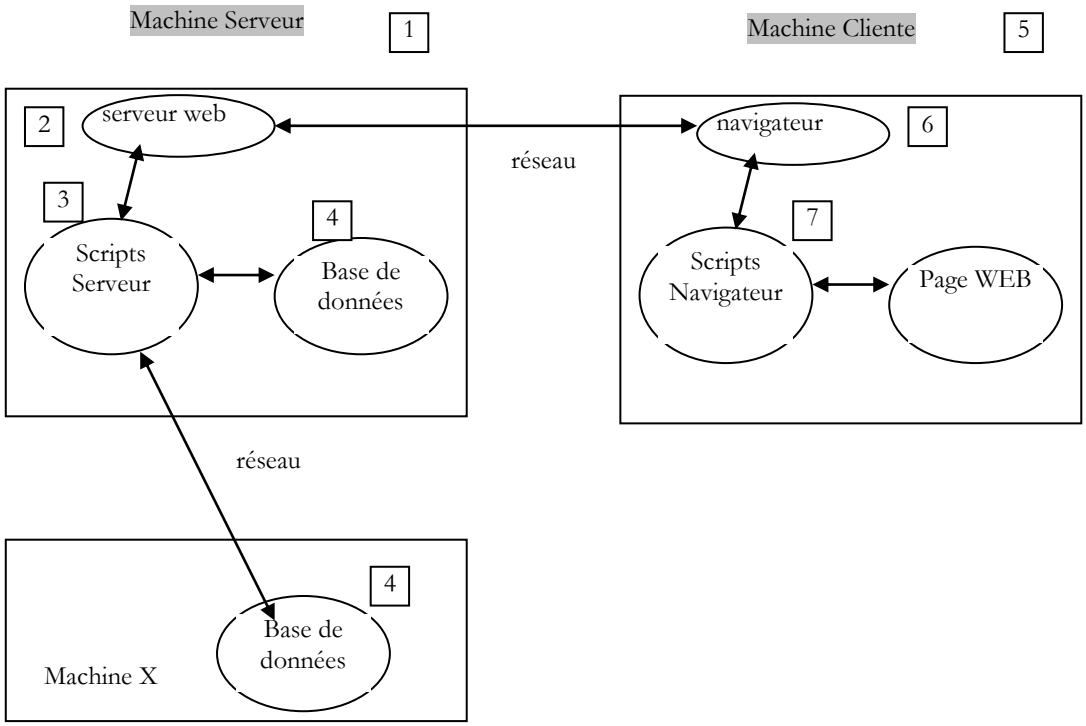

Les tests

- serveur Apache
- script *js1.html* dans <*apache-DocumentRoot*>
- demander l'url *http://localhost/js1.html* avec le navigateur IE ou Netscape

- serveur PWS
- script *js1.html* dans <*pws-DocumentRoot*>
- demander l'url *http://localhost/js1.html* avec le navigateur IE ou Netscape

## 1.7 Les échanges client-serveur

Revenons à notre schéma de départ qui illustrait les acteurs d'une application web :




Nous nous intéressons ici aux échanges entre la machine cliente et la machine serveur. Ceux-ci se font au travers d'un réseau et il est bon de rappeler la structure générale des échanges entre deux machines distantes.

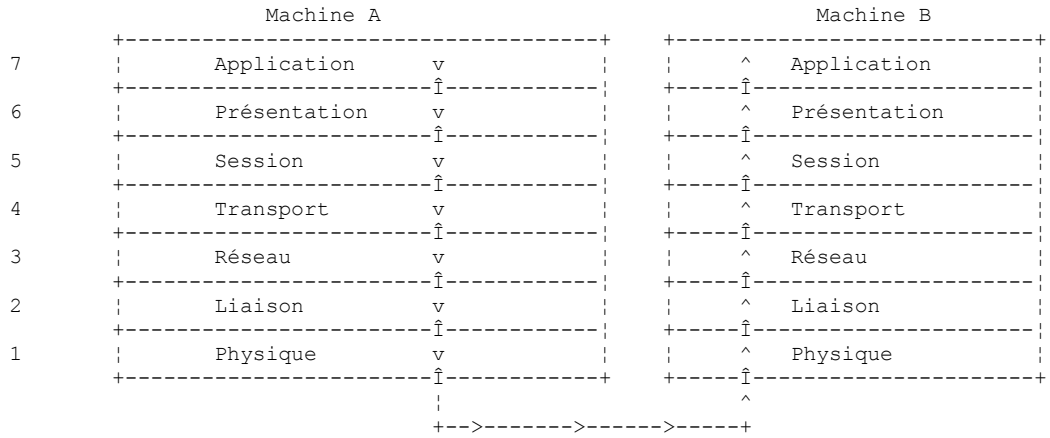
## 1.7.1 Le modèle OSI

Le modèle de réseau ouvert appelé **OSI** (**O**pen **S**ystems **I**nterconnection Reference Model) défini par l'**ISO** (**I**nternational **S**tandards **O**rganisation) décrit un réseau idéal où la communication entre machines peut être représentée par un modèle à sept couches :

```
        |-------------------------------------|
7       |       Application                   |
        |-------------------------------------|
6       |       Présentation                  |
        |-------------------------------------|
5       |       Session                       |
        |-------------------------------------|
4       |       Transport                     |
        |-------------------------------------|
3       |       Réseau                        |
        |-------------------------------------|
2       |       Liaison                       |
        |-------------------------------------|
1       |       Physique                      |
        |-------------------------------------|
```

Chaque couche reçoit des services de la couche inférieure et offre les siens à la couche supérieure. Supposons que deux applications situées sur des machines A et B différentes veulent communiquer : elles le font au niveau de la couche *Application*. Elles n'ont pas besoin de connaître tous les détails du fonctionnement du réseau : chaque application remet l'information qu'elle souhaite transmettre à la couche du dessous : la couche *Présentation*. L'application n'a donc à connaître que les règles d'interfaçage avec la couche *Présentation*. Une fois l'information dans la couche *Présentation*, elle est passée selon d'autres règles à la couche *Session* et ainsi de suite, jusqu'à ce que l'information arrive sur le support physique et soit transmise physiquement à la machine destination. Là, elle subira le traitement inverse de celui qu'elle a subi sur la machine expéditeur.

A chaque couche, le processus expéditeur chargé d'envoyer l'information, l'envoie à un processus récepteur sur l'autre machine apartenant à la même couche que lui. Il le fait selon certaines règles que l'on appelle le **protocole** de la couche. On a donc le schéma de communication final suivant :




Le rôle des différentes couches est le suivant :

**Physique** Assure la transmission de bits sur un support physique. On trouve dans cette couche des équipements terminaux de traitement des données (E.T.T.D.) tels que terminal ou ordinateur, ainsi que des équipements de terminaison de circuits de données (E.T.C.D.) tels que modulateur/démodulateur, multiplexeur, concentrateur. Les points d'intérêt à ce niveau sont :

. le choix du codage de l'information (analogique ou numérique)
. le choix du mode de transmission (synchrone ou asynchrone).

**Liaison de données** Masque les particularités physiques de la couche Physique. Détecte et corrige les erreurs de transmission.

**Réseau** Gère le chemin que doivent suivre les informations envoyées sur le réseau. On appelle cela le *routage* : déterminer la route à suivre par une information pour qu'elle arrive à son destinataire.

**Transport** Permet la communication entre deux applications alors que les couches précédentes ne permettaient que la communication entre machines. Un service fourni par cette couche peut être le multiplexage : la couche transport pourra utiliser une même connexion réseau (de machine à machine) pour transmettre des informations appartenant à plusieurs applications.

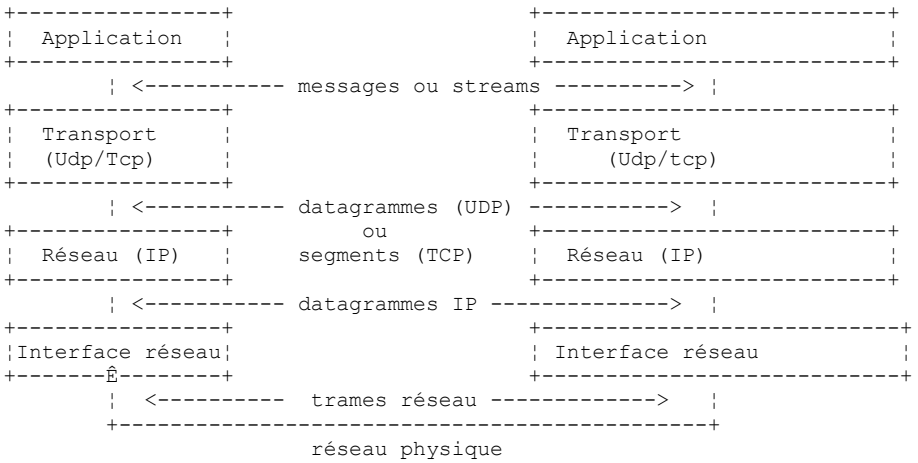
**Session** On va trouver dans cette couche des services permettant à une application d'ouvrir et de maintenir une session de travail sur une machine distante.

**Présentation** Elle vise à uniformiser la représentation des données sur les différentes machines. Ainsi des données provenant d'une machine A, vont être "habillées" par la couche *Présentation* de la machine A, selon un format standard avant d'être envoyées sur le réseau. Parvenues à la couche *Présentation* de la machine destinatrice B qui les reconnaîtra grâce à leur format standard, elles seront habillées d'une autre façon afin que l'application de la machine B les reconnaisse.

**Application** A ce niveau, on trouve les applications généralement proches de l'utilisateur telles que la messagerie électronique ou le transfert de fichiers.

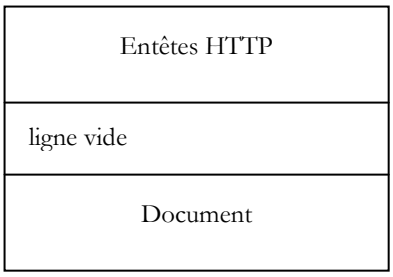
## 1.7.2 Le modèle TCP/IP

Le modèle OSI est un modèle idéal. La suite de protocoles TCP/IP s'en approche sous la forme suivante :




- l'**interface réseau** (la carte réseau de l'ordinateur) assure les fonctions des couches 1 et 2 du modèle OSI
- la couche **IP** (Internet Protocol) assure les fonctions de la couche 3 (réseau)
- la couche **TCP** (Transfer Control Protocol) ou **UDP** (User Datagram Protocol) assure les fonctions de la couche 4 (transport). Le protocole TCP s'assure que les paquets de données échangés par les machines arrivent bien à destination. Si ce n'est pas les cas, il renvoie les paquets qui se sont égarés. Le protocole UDP ne fait pas ce travail et c'est alors au développeur d'applications de le faire. C'est pourquoi sur l'internet qui n'est pas un réseau fiable à 100%, c'est le protocole TCP qui est le plus utilisé. On parle alors de réseau **TCP-IP**.
- la couche **Application** recouvre les fonctions des niveaux 5 à 7 du modèle OSI.

Les applications web se trouvent dans la couche *Application* et s'appuient donc sur les protocoles TCP-IP. Les couches *Application* des machines clientes et serveur s'échangent des messages qui sont confiées aux couches 1 à 4 du modèle pour être acheminées à destination. Pour se comprendre, les couches application des deux machines doivent "parler" un même langage ou protocole. Celui des applications Web s'appelle **HTTP** (HyperText Transfer Protocol). C'est un protocole de type texte, c.a.d. que les machines échangent des lignes de texte sur le réseau pour se comprendre. Ces échanges sont normalisés, c.a.d. que le client dispose d'un certain nombre de messages pour indiquer exactement ce qu'il veut au serveur et ce dernier dispose également d'un certain nombre de messages pour donner au client sa réponse. Cet échange de messages a la forme suivante :




Client --> Serveur

Lorsque le client fait sa demande au serveur web, il envoie

1. des lignes de texte au format HTTP pour indiquer ce qu'il veut
2. une ligne vide
3. optionnellement un document

Serveur --> Client

Lorsque le serveur fait sa réponse au client, il envoie

1. des lignes de texte au format HTTP pour indiquer ce qu'il envoie
2. une ligne vide
3. optionnellement un document

Les échanges ont donc la même forme dans les deux sens. Dans les deux cas, il peut y avoir envoi d'un document même s'il est rare qu'un client envoie un document au serveur. Mais le protocole HTTP le prévoit. C'est ce qui permet par exemple aux abonnés d'un fournisseur d'accès de télécharger des documents divers sur leur site personnel hébergé chez ce fournisseur d'accès. Les documents échangés peuvent être quelconques. Prenons un navigateur demandant une page web contenant des images :

1. le navigateur se connecte au serveur web et demande la page qu'il souhaite. Les ressources demandées sont désignées de façon unique par des URL (Uniform Resource Locator). Le navigateur n'envoie que des entêtes HTTP et aucun document.
2. le serveur lui répond. Il envoie tout d'abord des entêtes HTTP indiquant quel type de réponse il envoie. Ce peut être une erreur si la page demandée n'existe pas. Si la page existe, le serveur dira dans les entêtes HTTP de sa réponse qu'après ceux-ci il va envoyer un document **HTML** (HyperText Markup Language). Ce document est une suite de lignes de texte au format HTML. Un texte HTML contient des balises (marqueurs) donnant au navigateur des indications sur la façon d'afficher le texte.
3. le client sait d'après les entêtes HTTP du serveur qu'il va recevoir un document HTML. Il va analyser celui-ci et s'apercevoir peut-être qu'il contient des références d'images. Ces dernières ne sont pas dans le document HTML. Il fait donc une nouvelle demande au même serveur web pour demander la première image dont il a besoin. Cette demande est identique à celle faite en 1, si ce n'est que la resource demandée est différente. Le serveur va traiter cette demande en envoyant à son client l'image demandée. Cette fois-ci, dans sa réponse, les entêtes HTTP préciseront que le document envoyé est une image et non un document HTML.
4. le client récupère l'image envoyée. Les étapes 3 et 4 vont être répétées jusqu'à ce que le client (un navigateur en général) ait tous les documents lui permettant d'afficher l'intégralité de la page.

## 1.7.3 Le protocole HTTP

Découvrons le protocole HTTP sur des exemples. Que s'échangent un navigateur et un serveur web ?

### *1.7.3.1 La réponse d'un serveur HTTP*

Nous allons découvrir ici comment un serveur web répond aux demandes de ses clients. Le service Web ou service HTTP est un service TCP-IP qui travaille habituellement sur le port 80. Il pourrait travailler sur un autre port. Dans ce cas, le navigateur client serait obligé de préciser ce port dans l'URL qu'il demande. Une URL a la forme générale suivante :

**protocole://machine[:port]/chemin/infos**

avec

| | |
|---|---|
| **protocole** | http pour le service web. Un navigateur peut également servir de client à des services ftp, news, telnet, .. |
| **machine** | nom de la machine où officie le service web |
| **port** | port du service web. Si c'est 80, on peut omettre le n° du port. C'est le cas le plus fréquent |
| **chemin** | chemin désignant la ressource demandée |
| **infos** | informations complémentaires données au serveur pour préciser la demande du client |

Que fait un navigateur lorsqu'un utilisateur demande le chargement d'une URL ?

1. il ouvre une communication TCP-IP avec la machine et le port indiqués dans la partie **machine[:port]** de l'URL. Ouvrir une communication TCP-IP, c'est créer un "tuyau" de communication entre deux machines. Une fois ce tuyau créé, toutes les informations échangées entre les deux machines vont passer dedans. La création de ce tuyau TCP-IP n'implique pas encore le protocole HTTP du Web.
2. le tuyau TCP-IP créé, le client va faire sa demande au serveur Web et il va la faire en lui envoyant des lignes de texte (des commandes) au format HTTP. Il va envoyer au serveur la partie **chemin/infos** de l'URL
3. le serveur lui répondra de la même façon et dans le même tuyau
4. l'un des deux partenaires prendra la décision de fermer le tuyau. Cela dépend du protocole HTTP utilisé. Avec le protocole HTTP 1.0, le serveur ferme la connexion après chacune de ses réponses. Cela oblige un client qui doit faire plusieurs demandes pour obtenir les différents documents constituant une page web à ouvrir une nouvelle connexion à chaque demande, ce qui a un coût. Avec le protocole HTTP/1.1, le client peut dire au serveur de garder la connexion ouverte jusqu'à ce qu'il lui dise de la fermer. Il peut donc récupérer tous les documents d'une page web avec une seule connexion et fermer lui-même la connexion une fois le dernier document obtenu. Le serveur détectera cette fermeture et fermera lui aussi la connexion.

Pour découvrir les échanges entre un client et un serveur web, nous allons utiliser un client TCP générique. C'est un programme qui peut être client de tout service ayant un protocole de communication à base de lignes de texte comme c'est le cas du protocole HTTP. Ces lignes de texte seront tapées par l'utilisateur au clavier. Cela nécessite qu'il connaisse le protocole de communication du service qu'il cherche à atteindre. La réponse du serveur est ensuite affichée à l'écran. Le programme a été écrit en Java et on le trouvera en annexe. On l'utilise ici dans une fenêtre Dos sous windows et on l'appelle de la façon suivante :

**java clientTCPgenerique machine port**

avec

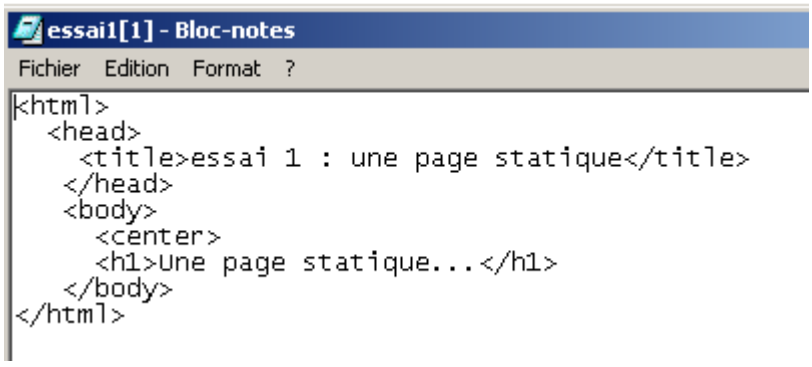
**machine** nom de la machine où officie le service à contacter
**port** port où le service est délivré

Avec ces deux informations, le programme va ouvrir une connexion TCP-IP avec la machine et le port désignés. Cette connexion va servir aux échanges de lignes de texte entre le client et le serveur web. Les lignes du client sont tapées par l'utilisateur au clavier et envoyées au serveur. Les lignes de texte renvoyées par le serveur comme réponse sont affichées à l'écran. Un dialogue peut donc s'instaurer directement entre l'utilisateur au clavier et le serveur web. Essayons sur les exemples déjà présentés. Nous avions créé la page HTML statique suivante :

```
<html>
  <head>
    <title>essai 1 : une page statique</title>
   </head>
   <body>
     <center>
     <h1>Une page statique...</h1>
   </body>
</html>
```

que nous visualisons dans un navigateur :


On voit que l'URL demandée est : *http://localhost:81/essais/essai1.html*. La machine du service web est donc *localhost* (=machine locale) et le port 81. Si on demande à voir le texte HTML de cette page Web (Affichage/Source) on retrouve le texte HTML initialement créé :




Maintenant utilisons notre client TCP générique pour demander la même URL :

```
Dos>java clientTCPgenerique localhost 81

Commandes :
GET /essais/essai1.html HTTP/1.0

<-- HTTP/1.1 200 OK
<-- Date: Mon, 08 Jul 2002 08:07:46 GMT
<-- Server: Apache/1.3.24 (Win32) PHP/4.2.0
```

```
<-- Last-Modified: Mon, 08 Jul 2002 08:00:30 GMT
<-- ETag: "0-a1-3d29469e"
<-- Accept-Ranges: bytes
<-- Content-Length: 161
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
<--   <head>
<--     <title>essai 1 : une page statique</title>
<--    </head>
<--    <body>
<--      <center>
<--      <h1>Une page statique...</h1>
<--    </body>
<-- </html>
```

Au lancement du client par la commande java clientTCPgenerique localhost 81 un tuyau a été créé entre le programme et le serveur web opérant sur la même machine (localhost) et sur le port 81. Les échanges client-serveur au format HTTP peuvent commencer. Rappelons que ceux-ci ont trois composantes :

1. entêtes HTTP
2. ligne vide
3. données facultatives

Dans notre exemple, le client n'envoie qu'une demande:

```
GET /essais/essai1.html HTTP/1.0
```

Cette ligne a trois composantes :

| | |
|---|---|
| `GET` | commande HTTP pour demander une ressource. Il en existe d'autres :<br>HEAD demande une ressource mais en se limitant aux entêtes HTTP de la réponse du serveur. La ressource elle-même n'est pas envoyée.<br>PUT permet au client d'envoyer un document au serveur |
| `/essais/essai1.html` | ressource demandée |
| `HTTP/1.0` | niveau du protocole HTTP utilisé. Ici le 1.0. Cela signifie que le serveur fermera la connexion dès qu'il aura envoyé sa réponse |

Les entêtes HTTP doivent toujours être suivis d'une ligne vide. C'est ce qui a été fait ici par le client. C'est comme cela que le client ou le serveur sait que la partie HTTP de l'échange est terminé. Ici pour le client c'est terminé. Il n'a pas de document à envoyer. Commence alors la réponse du serveur composée dans notre exemple de toutes les lignes commençant par le signe <--. Il envoie d'abord une série d'entêtes HTTP suivie d'une ligne vide :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 08 Jul 2002 08:07:46 GMT
<-- Server: Apache/1.3.24 (Win32) PHP/4.2.0
<-- Last-Modified: Mon, 08 Jul 2002 08:00:30 GMT
<-- ETag: "0-a1-3d29469e"
<-- Accept-Ranges: bytes
<-- Content-Length: 161
<-- Connection: close
<-- Content-Type: text/html
<--
```

| | |
|---|---|
| `HTTP/1.1 200 OK` | le serveur dit<br>• qu'il comprend le protocole HTTP version 1.1<br>• qu'il a la ressource demandée (code 200, message OK) |
| `Date: ...` | la date/heure de la réponse |
| `Server:` | le serveur s'identifie. Ici c'est un serveur Apache |
| `Last-Modified:` | date de dernière modification de la ressource demandée par le client |
| `ETag:` | ... |
| `Accept-Ranges: bytes` | unité de mesure des données envoyées. Ici l'octet (byte) |
| `Content-Length: 161` | nombre de bytes du document qui va être envoyé après les entêtes HTTP. Ce nombre est en fait la taille en octets du fichier *essai1.html* :<br>`E:\data\serge\web\essais>dir essai1.html` |

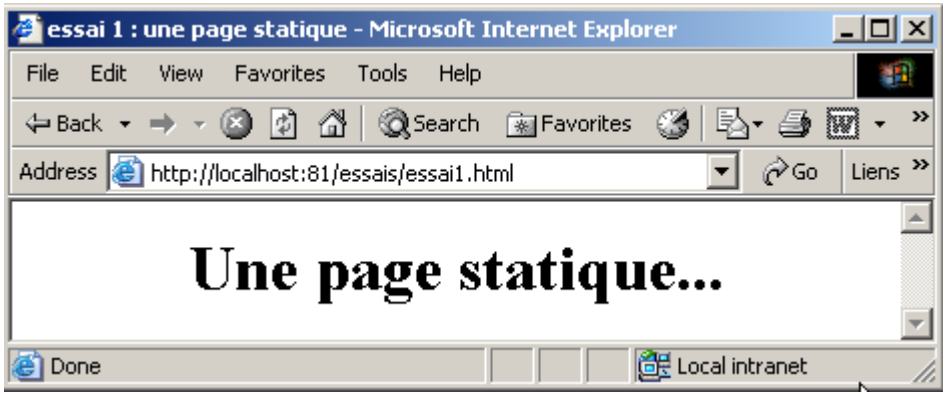
```
08/07/2002  10:00                   161 essai1.html
```

`Connection: close` le serveur dit qu'il fermera la connexion une fois le document envoyé
`Content-type: text/html` le serveur dit qu'il va envoyer du texte (text) au format HTML (html).

Le client reçoit ces entêtes HTTP et sait maintenant qu'il va recevoir 161 octets représentant un document HTML. Le serveur envoie ces 161 octets immédiatement derrière la ligne vide qui signalait la fin des entêtes HTTP :

```
<-- <html>
<--   <head>
<--     <title>essai 1 : une page statique</title>
<--    </head>
<--    <body>
<--      <center>
<--      <h1>Une page statique...</h1>
<--    </body>
<-- </html>
```

On reconnaît là, le fichier HTML construit initialement. Si notre client était un navigateur, après réception de ces lignes de texte, il les interpréterait pour présenter à l'utilisateur au clavier la page suivante :

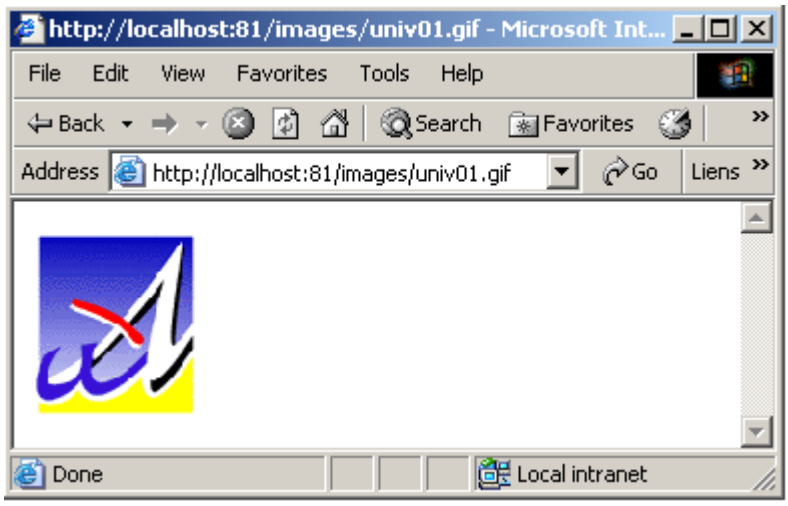



Utilisons une nouvelle fois notre client TCP générique pour demander la même ressource mais cette fois-ci avec la commande HEAD qui demande seulement les entêtes de la réponse :

```
Dos>java.bat clientTCPgenerique localhost 81
Commandes :
HEAD /essais/essai1.html HTTP/1.1
Host: localhost:81

<-- HTTP/1.1 200 OK
<-- Date: Mon, 08 Jul 2002 09:07:25 GMT
<-- Server: Apache/1.3.24 (Win32) PHP/4.2.0
<-- Last-Modified: Mon, 08 Jul 2002 08:00:30 GMT
<-- ETag: "0-a1-3d29469e"
<-- Accept-Ranges: bytes
<-- Content-Length: 161
<-- Content-Type: text/html
<--
```

Nous obtenons le même résultat que précédemment sans le document HTML. Notons que dans sa demande HEAD, le client a indiqué qu'il utilisait le protocole HTTP version 1.1. Cela l'oblige à envoyer un second entête HTTP précisant le couple *machine:port* que le client veut interroger : `Host: localhost:81`.

Maintenant demandons une image aussi bien avec un navigateur qu'avec le client TCP générique. Tout d'abord avec un navigateur :




Le fichier *univ01.gif* a 3167 octets :

```
E:\data\serge\web\images>dir univ01.gif
14/04/2000  13:37                 3 167 univ01.gif
```

Utilisons maintenant le client TCP générique :

```
E:\data\serge\JAVA\SOCKETS\client générique>java clientTCPgenerique localhost 81
Commandes :
HEAD /images/univ01.gif HTTP/1.1
host: localhost:81

<-- HTTP/1.1 200 OK
<-- Date: Tue, 09 Jul 2002 13:53:24 GMT
<-- Server: Apache/1.3.24 (Win32) PHP/4.2.0
<-- Last-Modified: Fri, 14 Apr 2000 11:37:42 GMT
<-- ETag: "0-c5f-38f70306"
<-- Accept-Ranges: bytes
<-- Content-Length: 3167
<-- Content-Type: image/gif
<--
```

On notera les points suivants dans la réponse du serveur :

```
HEAD
```

- nous ne demandons que les entêtes HTTP de la ressource. En effet, une image est un fichier binaire et non un fichier texte et son affichage à l'écran en tant que texte ne donne rien de lisible.

```
Content-Length: 3167
```

- c'est la taille du fichier *univ01.gif*

```
Content-Type: image/gif
```

- le serveur indique à son client qu'il va lui envoyer un document de type *image/gif*, c.a.d. une image au format GIF. Si l'image avait été au format JPEG, le type du document aurait été *image/jpeg*. Les types des documents sont standardisés et sont appelés des types MIME (Multi-purpose Mail Internet Extension).

## *1.7.3.2 La demande d'un client HTTP*

Maintenant, posons-nous la question suivante : si nous voulons écrire un programme qui "parle" à un serveur web, quelles commandes doit-il envoyer au serveur web pour obtenir une ressource donnée ? Nous avons dans les exemples précédents obtenu un début de réponse. Nous avons rencontré trois commandes :

```
GET ressource protocole
```

- pour demander une ressource donnée selon une version donnée du protocole HTTP. Le serveur envoie une réponse au format HTTP suivie d'une ligne vide suivie de la ressource demandée

```
HEAD ressource protocole
```

- idem si ce n'est qu'ici la réponse se limite aux entêtes HTTP et de la ligne vide

```
host: machine:port
```

- pour préciser (protocole HTTP 1.1) la machine et le port du serveur web interrogé

Il existe d'autres commandes. Pour les découvrir, nous allons maintenant utiliser un serveur TCP générique. C'est un programme écrit en Java et que vous trouverez lui aussi en annexe. Il est lancé par : **java serveurTCPgenerique portEcoute**, où *portEcoute* est le port sur lequel les clients doivent se connecter. Le programme **serveurTCPgenerique**

- affiche à l'écran les commandes envoyées par les clients
- leur envoie comme réponse les lignes de texte tapées au clavier par un utilisateur. C'est donc ce dernier qui fait office de serveur. Dans notre exemple, l'utilisateur au clavier jouera le rôle d'un service web.

Simulons maintenant un serveur web en lançant notre serveur générique sur le port 88 :

```
Dos> java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
```

Prenons maintenant un navigateur et demandons l'URL *http://localhost:88/exemple.html*. Le navigateur va alors se connecter sur le port *88* de la machine *localhost* puis demander la page */exemple.html* :


Regardons maintenant la fenêtre de notre serveur qui affiche ce que le client lui a envoyé (certaines lignes spécifiques au fonctionnement du programme *serveurTCPgenerique* ont été omises par souci de simplification) :

```
Dos>java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
...
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2 914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

Les lignes précédées du signe <-- sont celles envoyées par le client. On découvre ainsi des entêtes HTTP que nous n'avions pas encore rencontrés :

`Accept:`
`Accept-language:`
`Accept-Encoding:`
`User-Agent:`
`Connection:`

- liste de types MIME de documents que le navigateur sait traiter.
- la langue acceptée de préférence pour les documents.
- le type d'encodage des documents que le navigateur sait traiter
- identité du client
- *Close* : le serveur fermera la connexion après avoir donné sa réponse
- *Keep-Alive* : la connexion restera ouverte après réception de la réponse du serveur. Cela permettra au navigateur de demander les autres documents nécessaires à la construction de la page sans avoir à recréer une connexion.

Les entêtes HTTP envoyés par le navigateur se terminent par une ligne vide comme attendu.

Elaborons une réponse à notre client. L'utilisateur au clavier est ici le véritable serveur et il peut élaborer une réponse à la main. Rappelons-nous la réponse faite par un serveur Web dans un précédent exemple :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 08 Jul 2002 08:07:46 GMT
<-- Server: Apache/1.3.24 (Win32) PHP/4.2.0
<-- Last-Modified: Mon, 08 Jul 2002 08:00:30 GMT
<-- ETag: "0-a1-3d29469e"
<-- Accept-Ranges: bytes
<-- Content-Length: 161
<-- Connection: close
```

```
<-- Content-Type: text/html
<--
<-- <html>
<--   <head>
<--     <title>essai 1 : une page statique</title>
<--    </head>
<--    <body>
<--      <center>
<--      <h1>Une page statique...</h1>
<--    </body>
<-- </html>
```

Essayons d'élaborer à la main (au clavier) une réponse analogue. Les lignes commençant par --> *:* sont envoyées au client :

```
...
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
--> : HTTP/1.1 200 OK
--> : Server: serveur tcp generique
--> : Connection: close
--> : Content-Type: text/html
--> :
--> : <html>
--> :   <head><title>Serveur generique</title></head>
--> :   <body>
--> :     <center>
--> :       <h2>Reponse du serveur generique</h2>
--> :     </center>
--> :    </body>
--> : </html>
fin
```

La commande *fin* est propre au fonctionnement du programme *serveurTCPgenerique*. Elle arrête l'exécution du programme et clôt la connexion du serveur au client. Nous nous sommes limités dans notre réponse aux entêtes HTTP suivants :

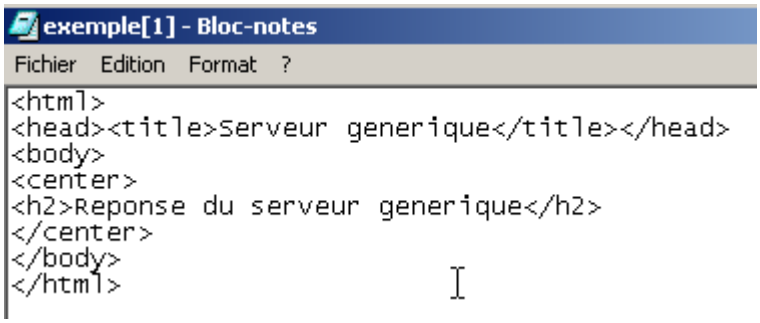
```
HTTP/1.1 200 OK
--> : Server: serveur tcp generique
--> : Connection: close
--> : Content-Type: text/html
--> :
```

Nous ne donnons pas la taille du fichier que nous allons envoyer (*Content-Length*) mais nous contentons de dire que nous allons fermer la connexion (*Connection: close*) après envoi de celui-ci. Cela est suffisant pour le navigateur. En voyant la connexion fermée, il saura que la réponse du serveur est terminée et affichera la page HTML qui lui a été envoyée. Cette dernière est la suivante :

```
--> : <html>
--> :   <head><title>Serveur generique</title></head>
--> :   <body>
--> :     <center>
--> :       <h2>Reponse du serveur generique</h2>
--> :     </center>
--> :    </body>
--> : </html>
```

Le navigateur affiche alors la page suivante :


Si ci-dessus, on fait *View/Source* pour voir ce qu'a reçu le navigateur, on obtient :




c'est à dire exactement ce qu'on a envoyé depuis le serveur générique.

## 1.8 Le langage HTML

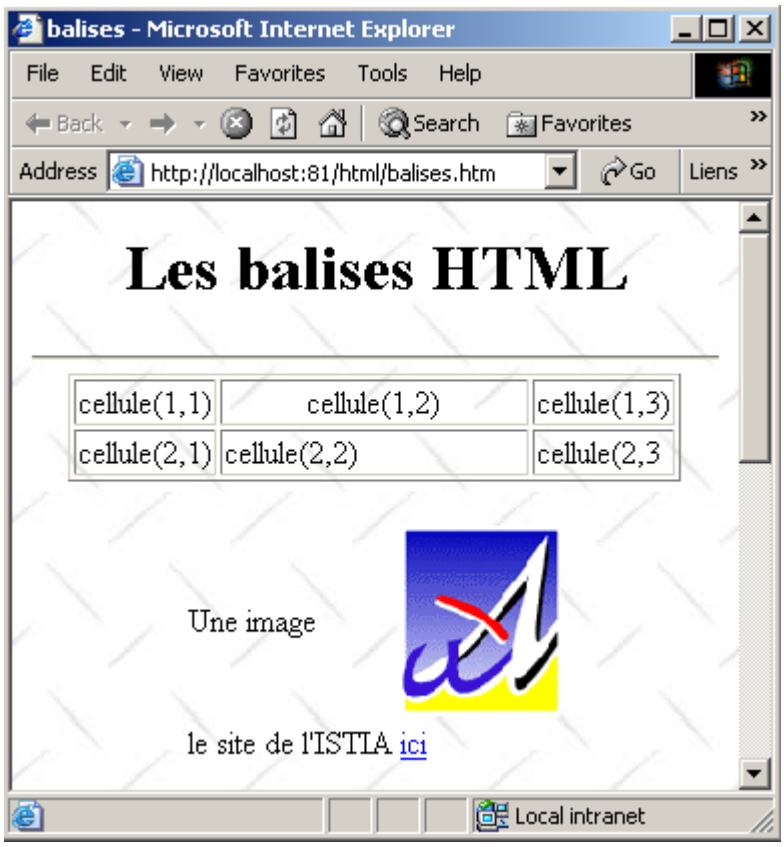
Un navigateur Web peut afficher divers documents, le plus courant étant le document HTML (HyperText Markup Language). Celui-ci est un texte formaté avec des balises de la forme *<balise>texte</balise>*. Ainsi le texte *<B>important</B>* affichera le texte important en gras. Il existe des balises seules telles que la balise *<hr>* qui affiche une ligne horizontale. Nous ne passerons pas en revue les balises que l'on peut trouver dans un texte HTML. Il existe de nombreux logiciels WYSIWYG permettant de construire une page web sans écrire une ligne de code HTML. Ces outils génèrent automatiquement le code HTML d'une mise en page faite à l'aide de la souris et de contrôles prédéfinis. On peut ainsi insérer (avec la souris) dans la page un tableau puis consulter le code HTML généré par le logiciel pour découvrir les balises à utiliser pour définir un tableau dans une page Web. Ce n'est pas plus compliqué que cela. Par ailleurs, la connaissance du langage HTML est indispensable puisque les applications web dynamiques doivent générer elles-mêmes le code HTML à envoyer aux clients web. Ce code est généré par programme et il faut bien sûr savoir ce qu'il faut générer pour que le client ait la page web qu'il désire.

Pour résumer, il n'est nul besoin de connaître la totalité du langage HTML pour démarrer la programmation Web. Cependant cette connaissance est nécessaire et peut être acquise au travers de l'utilisation de logiciels WYSIWYG de construction de pages Web tels que Word, FrontPage, DreamWeaver et des dizaines d'autres. Une autre façon de découvrir les subtilités du langage HTML est de parcourir le web et d'afficher le code source des pages qui présentent des caractéristiques intéressantes et encore inconnues pour vous.

### 1.8.1 Un exemple

Considérons l'exemple suivant, créé avec FrontPage Express, un outil gratuit livré avec Internet Explorer. Le code généré par Frontpage a été ici épuré. Cet exemple présente quelques éléments qu'on peut trouver dans un document web tels que :

- un tableau
- une image
- un lien




Un document HTML a la forme générale suivante :

```
<html>
  <head>
    <title>Un titre</title>
    ...
  </head>
  <body attributs>
    ...
  </body>
</html>
```

L'ensemble du document est encadré par les balises **<html>...</html>**. Il est formé de deux parties :

1. **<head>...</head>** : c'est la partie non affichable du document. Elle donne des renseignements au navigateur qui va afficher le document. On y trouve souvent la balise **<title>...</title>** qui fixe le texte qui sera affiché dans la barre de titre du navigateur. On peut y trouver d'autres balises notamment des balises définissant les mots clés du document, mot clés utilisés ensuite par les moteurs de recherche. On peut trouver également dans cette partie des scripts, écrits le plus souvent en javascript ou vbscript et qui seront exécutés par le navigateur.
2. **<body attributs>...</body>** : c'est la partie qui sera affichée par le navigateur. Les balises HTML contenues dans cette partie indiquent au navigateur la forme visuelle "souhaitée" pour le document. Chaque navigateur va interpréter ces balises à sa façon. Deux navigateurs peuvent alors visualiser différemment un même document web. C'est généralement l'un des casse-têtes des concepteurs web.

Le code HTML de notre document exemple est le suivant :

```
<html>

  <head>
    <title>balises</title>
  </head>

  <body background="/images/standard.jpg">
    <center>
      <h1>Les balises HTML</h1>
      <hr>
    </center>
```

```
    <table border="1">
      <tr>
        <td>cellule(1,1)</td>
        <td valign="middle" align="center" width="150">cellule(1,2)</td>
        <td>cellule(1,3)</td>
      </tr>
      <tr>
        <td>cellule(2,1)</td>
        <td>cellule(2,2)</td>
        <td>cellule(2,3</td>
      </tr>
    </table>
```

```
    <table border="0">
      <tr>
        <td>Une image</td>
        <td><img border="0" src="/images/univ01.gif" width="80" height="95"></td>
      </tr>
      <tr>
        <td>le site de l'ISTIA</td>
        <td><a href="http://istia.univ-angers.fr">ici</a></td>
      </tr>
    </table>
  </body>
</html>
```

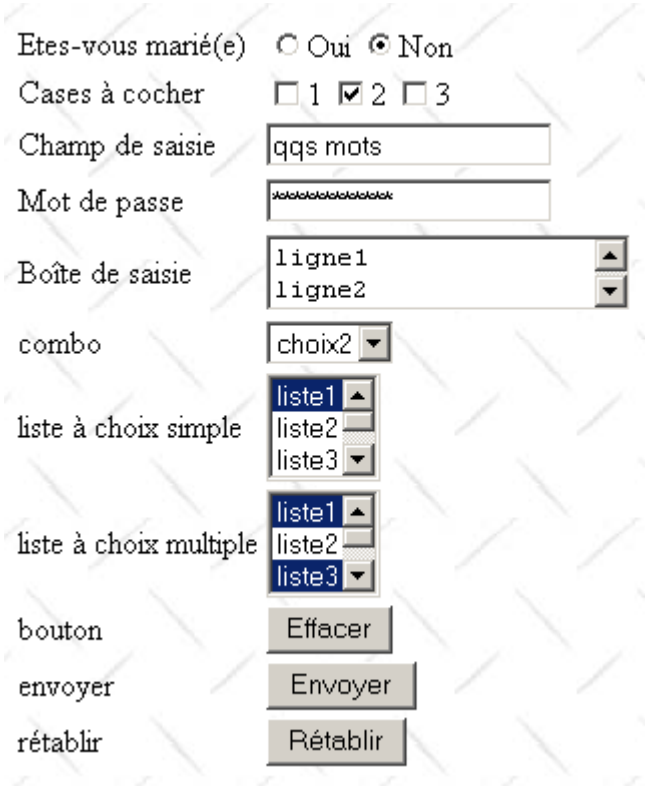
Ont été mis en relief dans le code les seuls points qui nous intéressent :

| Elément | balises et exemples HTML |
|---|---|
| titre du document | <*title*>*balises*</*title*><br>*balises* apparaîtra dans la barre de titre du navigateur qui affichera le document |
| barre horizontale | <*hr*> : affiche un trait horizontal |
| tableau | <*table attributs*>....</*table*> : pour définir le tableau<br><*tr attributs*>...</*tr*> : pour définir une ligne<br><*td attributs*>...</*td*> : pour définir une cellule<br><br>**exemples :**<br><*table border="1"*>...</*table*> : l'attribut border définit l'épaisseur de la bordure du tableau<br><*td valign="middle" align="center" width="150"*>*cellule(1,2)*</*td*> : définit une cellule dont le contenu sera cellule(1,2). Ce contenu sera centré verticalement (valign="middle") et horizontalement (align="center"). La cellule aura une largeur de 150 pixels (width="150") |
| image | <*img border="0" src="/images/univ01.gif" width="80" height="95"*> : définit une image sans bordure (border=0"), de hauteur 95 pixels (height="95"), de largeur 80 pixels (width="80") et dont le fichier source est */images/univ01.gif* sur le serveur web (src="/images/univ01.gif"). Ce lien se trouve sur un document web qui a été obtenu avec l'URL *http://localhost:81/html/balises.htm*. Aussi, le navigateur demandera-t-il l'URL *http://localhost:81/images/univ01.gif* pour avoir l'image référencée ici. |
| lien | <*a href="http://istia.univ-angers.fr"*>*ici*</*a*> : fait que le texte *ici* sert de lien vers l'URL *http://istia.univ-angers.fr*. |
| fond de page | <*body background="/images/standard.jpg"*> : indique que l'image qui doit servir de fond de page se trouve à l'URL */images/standard.jpg* du serveur web. Dans le contexte de notre exemple, le navigateur demandera l'URL *http://localhost:81/images/standard.jpg* pour obtenir cette image de fond. |

On voit dans ce simple exemple que pour construire l'intéralité du document, le navigateur doit faire trois requêtes au serveur :

1. *http://localhost:81/html/balises.htm* pour avoir le source HTML du document
2. *http://localhost:81/images/univ01.gif* pour avoir l'image *univ01.gif*
3. *http://localhost:81/images/standard.jpg* pour obtenir l'image de fond *standard.jpg*

L'exemple suivant présente un formulaire Web créé lui aussi avec FrontPage.




Le code HTML généré par FrontPage et un peu épuré est le suivant :

```
<html>

  <head>
    <title>balises</title>
    <script language="JavaScript">
      function effacer(){
        alert("Vous avez cliqué sur le bouton Effacer");
       }//effacer
    </script>
  </head>

  <body background="/images/standard.jpg">

    <form method="POST" >

      <table border="0">
        <tr>
          <td>Etes-vous marié(e)</td>
          <td>
            <input type="radio" value="Oui" name="R1">Oui
            <input type="radio" name="R1" value="non" checked>Non
          </td>
        </tr>
        <tr>
          <td>Cases à cocher</td>
          <td>
            <input type="checkbox" name="C1" value="un">1
            <input type="checkbox" name="C2" value="deux" checked>2
            <input type="checkbox" name="C3" value="trois">3
          </td>
        </tr>
        <tr>
          <td>Champ de saisie</td>
          <td>
            <input type="text" name="txtSaisie" size="20" value="qqs mots">
          </td>
        </tr>
        <tr>
          <td>Mot de passe</td>
          <td>
            <input type="password" name="txtMdp" size="20" value="unMotDePasse">
          </td>
```

```
        </tr>
        <tr>
          <td>Boîte de saisie</td>
          <td>
            <textarea rows="2" name="areaSaisie" cols="20">
ligne1
ligne2
ligne3
</textarea>
          </td>
        </tr>
        <tr>
          <td>combo</td>
          <td>
              <select size="1" name="cmbValeurs">
                <option>choix1</option>
                <option selected>choix2</option>
                <option>choix3</option>
              </select>
          </td>
        </tr>
        <tr>
          <td>liste à choix simple</td>
          <td>
              <select size="3" name="lst1">
                <option selected>liste1</option>
                <option>liste2</option>
                <option>liste3</option>
                <option>liste4</option>
                <option>liste5</option>
              </select>
          </td>
        </tr>
        <tr>
          <td>liste à choix multiple</td>
          <td>
              <select size="3" name="lst2" multiple>
                <option>liste1</option>
                <option>liste2</option>
                <option selected>liste3</option>
                <option>liste4</option>
                <option>liste5</option>
              </select>
          </td>
        </tr>
        <tr>
          <td>bouton</td>
          <td>
              <input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()">
          </td>
        </tr>
        <tr>
          <td>envoyer</td>
          <td>
              <input type="submit" value="Envoyer" name="cmdRenvoyer">
          </td>
        </tr>
        <tr>
          <td>rétablir</td>
          <td>
              <input type="reset" value="Rétablir" name="cmdRétablir">
          </td>
        </tr>
      </table>
      <input type="hidden" name="secret" value="uneValeur">

    </form>
  </body>
</html>
```

L'association contrôle visuel <--> balise HTML est le suivant :

| Contrôle | balise HTML |
|---|---|
| formulaire | <form method="POST" > |
| champ de saisie | <input type="text" name="txtSaisie" size="20" value="qqs mots"> |

| | |
|---|---|
| champ de saisie cachée | <input type="password" name="txtMdp" size="20" value="unMotDePasse"> |
| champ de saisie multilignes | <textarea rows="2" name="areaSaisie" cols="20"><br>ligne1<br>ligne2<br>ligne3<br></textarea> |
| boutons radio | <input type="radio" value="Oui" name="R1">Oui<br><input type="radio" name="R1" value="non" checked>Non |
| cases à cocher | <input type="checkbox" name="C1" value="un">1<br><input type="checkbox" name="C2" value="deux" checked>2<br><input type="checkbox" name="C3" value="trois">3 |
| Combo | <select size="1" name="cmbValeurs"><br><option>choix1</option><br><option selected>choix2</option><br><option>choix3</option><br></select> |
| liste à sélection unique | <select size="3" name="lst1"><br><option selected>liste1</option><br><option>liste2</option><br><option>liste3</option><br><option>liste4</option><br><option>liste5</option><br></select> |
| liste à sélection multiple | <select size="3" name="lst2" multiple><br><option>liste1</option><br><option>liste2</option><br><option selected>liste3</option><br><option>liste4</option><br><option>liste5</option><br></select> |
| bouton de type submit | <input type="submit" value="Envoyer" name="cmdRenvoyer"> |
| bouton de type reset | <input type="reset" value="Rétablir" name="cmdRétablir"> |
| bouton de type button | <input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()"> |

Passons en revue ces différents contrôles.

### *1.8.1.1 Le formulaire*

| | |
|---|---|
| formulaire | *<form method="POST" >* |
| balise HTML | **<form name="..." method="..." action="...">...</form>** |
| attributs | **name="frmexemple"** : nom du formulaire<br>**method="..."** : méthode utilisée par le navigateur pour envoyer au serveur web les valeurs récoltées dans le formulaire<br>**action="..."** : URL à laquelle seront envoyées les valeurs récoltées dans le formulaire. |

Un formulaire web est entouré des balises *<form>...</form>*. Le formulaire peut avoir un nom (*name="xx"*). C'est le cas pour tous les contrôles qu'on peut trouver dans un formulaire. Ce nom est utile si le document web contient des scripts qui doivent référencer des éléments du formulaire. Le but d'un formulaire est de rassembler des

informations données par l'utilisateur au clavier/souris et d'envoyer celles-ci à une URL de serveur web. Laquelle ? Celle référencée dans l'attribut *action="URL"*. Si cet attribut est absent, les informations seront envoyées à l'URL du document dans lequel se trouve le formulaire. Ce serait le cas dans l'exemple ci-dessus. Jusqu'à maintenant, nous avons toujours vu le client web comme "demandant" des informations à un serveur web, jamais lui "donnant" des informations. Comment un client web fait-il pour donner des informations (celles contenues dans le formulaire) à un serveur web ? Nous y reviendrons dans le détail un peu plus loin. Il peut utiliser deux méthodes différentes appelées POST et GET. L'attribut *method="méthode"*, avec méthode égale à GET ou POST, de la balise *<form>* indique au navigateur la méthode à utiliser pour envoyer les informations recueillies dans le formulaire à l'URL précisée par l'attribut *action="URL"*. Lorsque l'attribut *method* n'est pas précisé, c'est la méthode GET qui est prise par défaut.

## *1.8.1.2 Champ de saisie*


`champ de saisie`

```
<input type="text" name="txtSaisie" size="20" value="qqs mots">
<input type="password" name="txtMdp" size="20" value="unMotDePasse">
```

`balise HTML`

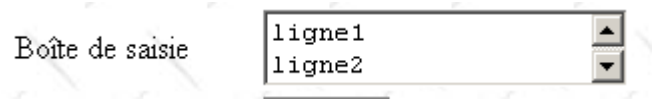
**<input type="..." name="..." size=".." value="..">**

La balise **input** existe pour divers contrôles. C'est l'attribut *type* qui permet de différentier ces différents contrôles entre eux.

`attributs`

**type="text"** : précise que c'est un champ de saisie
**type="password"** : les caractères présents dans le champ de saisie sont remplacés par des caractères *. C'est la seule différence avec le champ de saisie normal. Ce type de contrôle convient pour la saisie des mots de passe.
**size="20"** : nombre de caractères visibles dans le champ - n'empêche pas la saisie de davantage de caractères
**name="txtSaisie"** : nom du contrôle
**value="qqs mots"** : texte qui sera affiché dans le champ de saisie.

## *1.8.1.3 Champ de saisie multilignes*




`champ de saisie multilignes`

```
<textarea rows="2" name="areaSaisie" cols="20">
ligne1
ligne2
ligne3
</textarea>
```

`balise HTML`

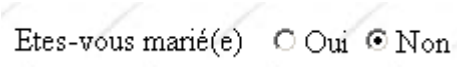
**<textarea ...>texte</textarea>**

affiche une zone de saisie multilignes avec au départ **texte** dedans

`attributs`

**rows="2"** : nombre de lignes
**cols="'20"** : nombre de colonnes
**name="areaSaisie"** : nom du contrôle

## *1.8.1.4 Boutons radio*




`boutons radio`

```
<input type="radio" value="Oui" name="R1">Oui
<input type="radio" name="R1" value="non" checked>Non
```

balise HTML **<input type="radio" attribut2="valeur2" ....>texte**
affiche un bouton radio avec **texte** à côté.

attributs **name="radio"** : nom du contrôle. Les boutons radio portant le même nom forment un groupe de boutons exclusifs les uns des autres : on ne peut cocher que l'un d'eux.
**value="valeur"** : valeur affectée au bouton radio. Il ne faut pas confondre cette valeur avec le texte affiché à côté du bouton radio. Celui-ci n'est destiné qu'à l'affichage.
**checked** : si ce mot clé est présent, le bouton radio est coché, sinon il ne l'est pas.

### *1.8.1.5 Cases à cocher*

cases à cocher

```
<input type="checkbox" name="C1" value="un">1
<input type="checkbox" name="C2" value="deux" checked>2
<input type="checkbox" name="C3" value="trois">3
```


balise HTML **<input type="checkbox" attribut2="valeur2" ....>texte**
affiche une case à cocher avec **texte** à côté.

attributs **name="C1"** : nom du contrôle. Les cases à cocher peuvent porter ou non le même nom. Les cases portant le même nom forment un groupe de cases associées.
**value="valeur"** : valeur affectée à la case à cocher. Il ne faut pas confondre cette valeur avec le texte affiché à côté du bouton radio. Celui-ci n'est destiné qu'à l'affichage.
**checked** : si ce mot clé est présent, le bouton radio est coché, sinon il ne l'est pas.

### *1.8.1.6 Liste déroulante (combo)*

Combo

```
<select size="1" name="cmbValeurs">
  <option>choix1</option>
  <option selected>choix2</option>
  <option>choix3</option>
</select>
```


balise HTML **<select size=".." name="..">**
**<option [selected]>...</option>**
**...**
**</select>**
affiche dans une liste les textes compris entre les balises <option>...</option>

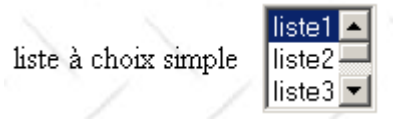
attributs **name="cmbValeurs"** : nom du contrôle.
**size="1"** : nombre d'éléments de liste visibles. *size="1"* fait de la liste l'équivalent d'un combobox.
**selected** : si ce mot clé est présent pour un élément de liste, ce dernier apparaît sélectionné dans la liste. Dans notre exemple ci-dessus, l'élément de liste *choix2* apparaît comme l'élément sélectionné du combo lorsque celui-ci est affiché pour la première fois.

### *1.8.1.7 Liste à sélection unique*

liste à sélection unique

```
<select size="3" name="lst1">
  <option selected>liste1</option>
  <option>liste2</option>
  <option>liste3</option>
  <option>liste4</option>
```

```
  <option>liste5</option>
</select>
```

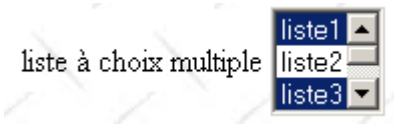



balise HTML
```
<select size=".." name="..">
  <option [selected]>...</option>
  ...
</select>
```
affiche dans une liste les textes compris entre les balises *<option>...</option>*

attributs
les mêmes que pour la liste déroulante n'affichant qu'un élément. Ce contrôle ne diffère de la liste déroulante précédente que par son attribut size>1.

## *1.8.1.8 Liste à sélection multiple*

liste à sélection unique
```
<select size="3" name="lst2" multiple>
  <option selected>liste1</option>
  <option>liste2</option>
  <option selected>liste3</option>
  <option>liste4</option>
  <option>liste5</option>
</select>
```




balise HTML
```
<select size=".." name=".." multiple>
  <option [selected]>...</option>
  ...
</select>
```
affiche dans une liste les textes compris entre les balises *<option>...</option>*

attributs
**multiple** : permet la sélection de plusieurs éléments dans la liste. Dans l'exemple ci-dessus, les éléments *liste1* et *liste3* sont tous deux sélectionnés.

## *1.8.1.9 Bouton de type button*

bouton de type button
```
<input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()">
```


balise HTML
**<input type="button" value="..." name="..." onclick="effacer()" ....>**

attributs
**type="button"** : définit un contrôle bouton. Il existe deux autres types de bouton, les types *submit* et *reset*.
**value="Effacer"** : le texte affiché sur le bouton
**onclick="fonction()"** : permet de définir une fonction à exécuter lorsque l'utilisateur clique sur le bouton. Cette fonction fait partie des scripts définis dans le document web affiché. La syntaxe précédente est une syntaxe *javascript*. Si les scripts sont écrits en vbscript, il faudrait écrire **onclick="fonction"** sans les parenthèses. La syntaxe devient identique s'il faut passer des paramètres à la fonction : **onclick="fonction(val1, val2,...)"**

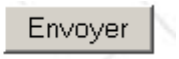
Dans notre exemple, un clic sur le bouton *Effacer* appelle la fonction javascript *effacer* suivante :

```
<script language="JavaScript">
 function effacer(){
  alert("Vous avez cliqué sur le bouton Effacer");
 }//effacer
</script>
```

La fonction *effacer* affiche un message :


### 1.8.1.10 Bouton de type submit

`bouton de type submit` *<input type="submit" value="Envoyer" name="cmdRenvoyer">*

envoyer 


`balise HTML` **<input type="submit" value="Envoyer" name="cmdRenvoyer">**

`attributs` **type="submit"** : définit le bouton comme un bouton d'envoi des données du formulaire au serveur web. Lorsque le client va cliquer sur ce bouton, le navigateur va envoyer les données du formulaire à l'URL définie dans l'attribut **action** de la balise **<form>** selon la méthode définie par l'attribut **method** de cette même balise.
**value="Envoyer"** : le texte affiché sur le bouton

### 1.8.1.11 Bouton de type reset

`bouton de type reset` *<input type="reset" value="Rétablir" name="cmdRétablir">*

rétablir 


`balise HTML` **<input type="reset" value="Rétablir" name="cmdRétablir">**

`attributs` **type="reset"** : définit le bouton comme un bouton de réinitialisation du formulaire. Lorsque le client va cliquer sur ce bouton, le navigateur va remettre le formulaire dans l'état où il l'a reçu.
**value="Rétablir"** : le texte affiché sur le bouton

### 1.8.1.12 Champ caché

`champ caché` *<input type="hidden" name="secret" value="uneValeur">*

`balise HTML` **<input type="hidden" name="..." value="...">**

`attributs` **type="hidden"** : précise que c'est un champ caché. Un champ caché fait partie du formulaire mais n'est pas présenté à l'utilisateur. Cependant, si celui-ci demandait à son navigateur l'affichage du code source, il verrait la présence de la balise <input type="hidden" value="..."> et donc la valeur du champ caché.

**value="uneValeur"** : valeur du champ caché.

Quel est l'intérêt du champ caché ? Cela peut permettre au serveur web de garder des informations au fil des requêtes d'un client. Considérons une application d'achats sur le web. Le client achète un premier article *art1* en quantité *q1* sur une première page d'un catalogue puis passe à une nouvelle page du catalogue. Pour se souvenir que le client a acheté *q1* articles *art1*, le serveur peut mettre ces deux informations dans un champ caché du formulaire web de la nouvelle page. Sur cette nouvelle page, le client achète *q2* articles *art2*. Lorsque les données de ce second formulaire vont être envoyées au serveur (submit), celui-ci va non seulement recevoir l'information (*q2*,*art2*) mais aussi (*q1*,*art1*) qui fait partie également partie du formulaire en tant que champ caché non modifiable par l'utilisateur. Le serveur web va alors mettre dans un nouveau champ caché les informations (*q1*,*art1*) et (*q2*,*art2*) et envoyer une nouvelle page de catalogue. Et ainsi de suite.

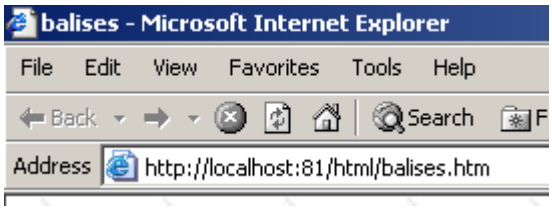
## 1.8.2 Envoi à un serveur web par un client web des valeurs d'un formulaire

Nous avons dit dans l'étude précédente que le client web disposait de deux méthodes pour envoyer à un serveur web les valeurs d'un formulaire qu'il a affiché : les méthodes GET et POST. Voyons sur un exemple la différence entre les deux méthodes. Nous reprenons l'exemple précédent et le traitons de la façon suivante :

1. un navigateur demande l'URL de l'exemple à un serveur web
2. une fois le formulaire obtenu, nous le remplissons
3. avant d'envoyer les valeurs du formulaire au serveur web en cliquant sur le bouton *Envoyer* de type *submit*, nous arrêtons le serveur web et le remplaçons par le serveur TCP générique déjà utilisé précédemment. Rappelons que celui-ci affiche à l'écran les lignes de texte que lui envoie le client web. Ainsi nous verrons ce qu'envoie exactement le navigateur.

Le formulaire est rempli de la façon suivante :


L'URL utilisée pour ce document est la suivante :




## 1.8.2.1 Méthode GET

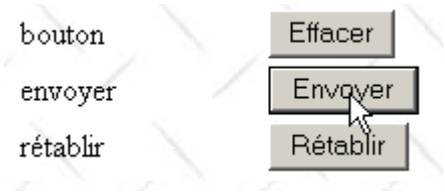
Le document HTML est programmé pour que le navigateur utilise la méthode GET pour envoyer les valeurs du formulaire au serveur web. Nous avons donc écrit :

```
    <form method="GET" >
```

Nous arrêtons le serveur web et lançons notre serveur TCP générique sur le port 81 :

```
E:\data\serge\JAVA\SOCKETS\serveur générique>java serveurTCPgenerique 81
Serveur générique lancé sur le port 81
```

Maintenant, nous revenons dans notre navigateur pour envoyer les données du formulaire au serveur web à l'aide du bouton *Envoyer* :




Voici alors ce que reçoit le serveur TCP générique :

```
<-- GET /html/balises.htm?R1=Oui&C1=un&C2=deux&txtSaisie=programmation+web&txtMdp=ceciestsecret&area
Saisie=les+bases+de+la%0D%0Aprogrammation+web&cmbValeurs=choix3&lst1=liste3&lst2=liste1&lst2=liste3&
cmdRenvoyer=Envoyer&secret=uneValeur HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, application/vnd
.ms-powerpoint, application/vnd.ms-excel, */*
<-- Referer: http://localhost:81/html/balises.htm
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
<-- Host: localhost:81
<-- Connection: Keep-Alive
<--
```

Tout est dans le premier entête HTTP envoyé par le navigateur :

```
<-- GET /html/balises.htm?R1=Oui&C1=un&C2=deux&txtSaisie=programmation+web&txtMdp=ceciestsecret&area
Saisie=les+bases+de+la%0D%0Aprogrammation+web&cmbValeurs=choix3&lst1=liste3&lst2=liste1&lst2=liste3&
cmdRenvoyer=Envoyer&secret=uneValeur HTTP/1.1
```

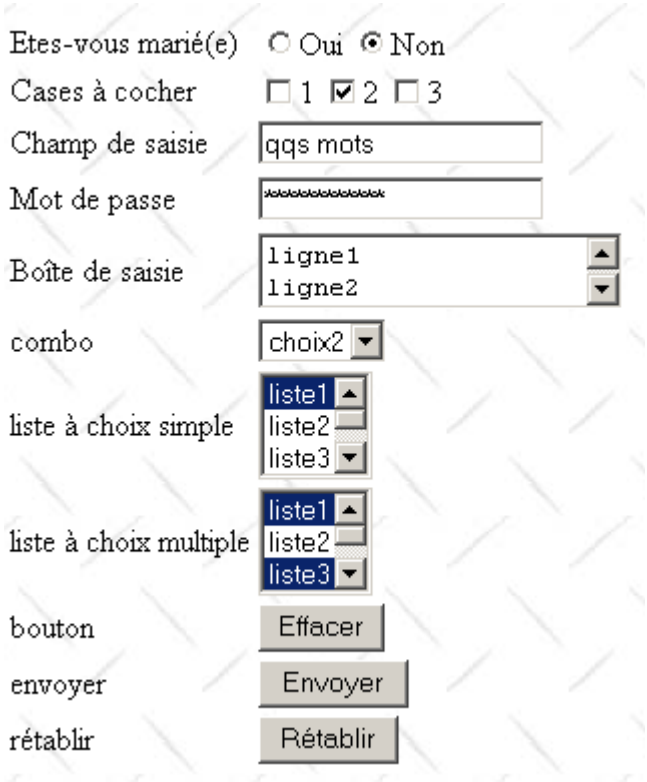
On voit qu'il est beaucoup plus complexe que ce qui avait été rencontré jusqu'à maintenant. On y retrouve la syntaxe *GET URL HTTP/1.1* mais sous une forme particulière *GET URL?param1=valeur1&param2=valeur2&... HTTP/1.1* où les *parami* sont les noms des contrôles du formulaire web et valeuri les valeurs qui leur sont associées. Examinons-les de plus près. Nous présentons ci-dessous un tableau à trois colonnes :

- colonne 1 : reprend la définition d'un contrôle HTML de l'exemple
- colonne 2 : donne l'affichage de ce contrôle dans un navigateur
- colonne 3 : donne la valeur envoyée au serveur par le navigateur pour le contrôle de la colonne 1 sous la forme qu'elle a dans la requête GET de l'exemple

| contrôle HTML | visuel | valeur(s) renvoyée(s) |
|---|---|---|

| | | |
|---|---|---|
| <input type="radio" value="Oui" name="R1">Oui<br><input type="radio" name="R1" value="non" checked>Non | Etes-vous marié(e) ◉ Oui ○ Non | **R1=Oui**<br>- la valeur de l'attribut *value* du bouton radio coché par l'utilisateur. |
| <input type="checkbox" name="C1" value="un">1<br><input type="checkbox" name="C2" value="deux" checked>2<br><input type="checkbox" name="C3" value="trois">3 | Cases à cocher ☑ 1 ☑ 2 ☐ 3 | **C1=un**<br>**C2=deux**<br>- valeurs des attributs *value* des cases cochées par l'utilisateur |
| <input type="text" name="txtSaisie" size="20" value="qqs mots"> | Champ de saisie programmation web | **txtSaisie=programmation+web**<br>- texte tapé par l'utilisateur dans le champ de saisie. Les espaces ont été remplacés par le signe + |
| <input type="password" name="txtMdp" size="20" value="unMotDePasse"> | Mot de passe ********** | **txtMdp=ceciestsecret**<br>- texte tapé par l'utilisateur dans le champ de saisie |
| <textarea rows="2" name="areaSaisie" cols="20"><br>ligne1<br>ligne2<br>ligne3<br></textarea> | Boîte de saisie les bases de la programmation web | **areaSaisie=les+bases+de+la%0D%0A programmation+web**<br>- texte tapé par l'utilisateur dans le champ de saisie. %OD%OA est la marque de fin de ligne. Les espaces ont été remplacés par le signe + |
| <select size="1" name="cmbValeurs"><br><option>choix1</option><br><option selected>choix2</option><br><option>choix3</option><br></select> | combo choix3 | **cmbValeurs=choix3**<br>- valeur choisie par l'utilisateur dans la liste à sélection unique |
| <select size="3" name="lst1"><br><option selected>liste1</option><br><option>liste2</option><br><option>liste3</option><br><option>liste4</option><br><option>liste5</option><br></select> | liste à choix simple liste1 liste2 liste3 | **lst1=liste3**<br>- valeur choisie par l'utilisateur dans la liste à sélection unique |
| <select size="3" name="lst2" multiple><br><option selected>liste1</option><br><option>liste2</option><br><option selected>liste3</option><br><option>liste4</option><br><option>liste5</option><br></select> | liste à choix multiple liste1 liste2 liste3 | **lst2=liste1**<br>**lst2=liste3**<br>- valeurs choisies par l'utilisateur dans la liste à sélection multiple |
| <input type="submit" value="Envoyer" name="cmdRenvoyer"> | | **cmdRenvoyer=Envoyer**<br>- nom et attribut *value* du bouton qui a servi à envoyer les données du formulaire au serveur |
| <input type="hidden" name="secret" value="uneValeur"> | | **secret=uneValeur**<br>- attribut *value* du champ caché |

Refaisons la même chose mais cette fois-ci en gardant le serveur web pour élaborer la réponse et voyons quelle est cette dernière. La page renvoyée par le serveur Web est la suivante :



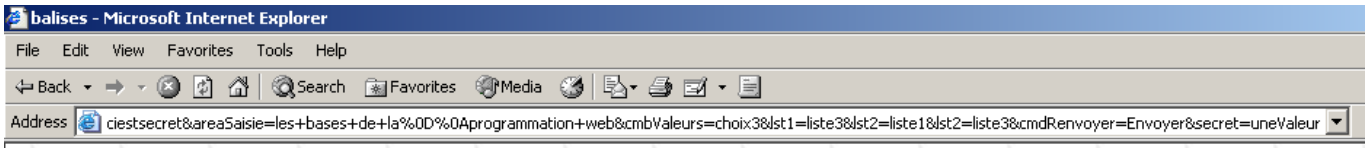

C'est exactement la même que celle reçue initialement avant le remplissage du formulaire. Pour comprendre pourquoi, il faut regarder de nouveau l'URL demandée par le navigateur lorsque l'utilisateur appuie sur le bouton *Envoyer* :

```
<-- GET /html/balises.htm?R1=Oui&C1=un&C2=deux&txtSaisie=programmation+web&txtMdp=ceciestsecret&area
Saisie=les+bases+de+la%0D%0Aprogrammation+web&cmbValeurs=choix3&lst1=liste3&lst2=liste1&lst2=liste3&
cmdRenvoyer=Envoyer&secret=uneValeur HTTP/1.1
```

L'URL demandée est */html/balises.htm*. On passe de plus à cette URL des valeurs qui sont celles du formulaire. Pour l'instant, l'URL */html/balises.htm* qui est une page statique n'utilise pas ces valeurs. Si bien que le GET précédent est équivalent à

```
<-- GET /html/balises.htm HTTP/1.1
```

et c'est pourquoi le serveur nous a renvoyé de nouveau la page initiale. On remarquera que le navigateur affiche bien lui l'URL complète qui a été demandée :




## *1.8.2.2 Méthode POST*

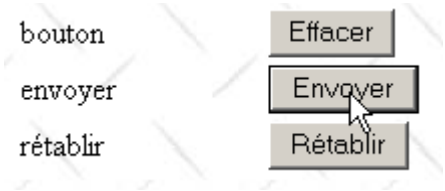
Le document HTML est programmé pour que le navigateur utilise maintenant la méthode POST pour envoyer les valeurs du formulaire au serveur web :

```
    <form method="POST" >
```

Nous arrêtons le serveur web et lançons le serveur TCP générique (déjà rencontré mais un peu modifié pour l'occasion) sur le port 81 :

```
E:\data\serge\JAVA\SOCKETS\serveur générique>java serveurTCPgenerique2 81
Serveur générique lancé sur le port 81
```

Maintenant, nous revenons dans notre navigateur pour envoyer les données du formulaire au serveur web à l'aide du bouton Envoyer :




Voici alors ce que reçoit le serveur TCP générique :

```
<-- POST /html/balises.htm HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, application/vnd
.ms-powerpoint, application/vnd.ms-excel, */*
<-- Referer: http://localhost:81/html/balises.htm
<-- Accept-Language: fr
<-- Content-Type: application/x-www-form-urlencoded
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
<-- Host: localhost:81
<-- Content-Length: 210
<-- Connection: Keep-Alive
<-- Cache-Control: no-cache
<--
<--
R1=Oui&C1=un&C2=deux&txtSaisie=programmation+web&txtMdp=ceciestsecret&areaSaisie=les+bases+de+la%0D%0Aprog
rammation+web&cmbValeurs=choix3&lst1=liste3&lst2=liste1&lst2=liste3&cmdRenvoyer=Envoyer&secret=uneValeur
```

Par rapport à ce que nous connaissons déjà, nous notons les changements suivants dans la requête du navigateur :

1. L'entête HTTP initial n'est plus GET mais **POST**. La syntaxe est POST URL HTTP/1.1 où URL est l'URL demandée par le navigateur. En même temps, POST signifie que le navigateur a des données à transmettre au serveur.
2. La ligne **Content-Type: application/x-www-form-urlencoded** indique quel type de données va envoyer le navigateur. Ce sont des données de formulaire (x-www-form) codées (urlencoded). Ce codage fait que certains caractères des données transmises sont transformées afin d'éviter au serveur des erreurs d'interprétation. Ainsi, l'espace est remplacé par +, la marque de fin de ligne par %OD%OA,... De façon générale, tous les caractères contenus dans les données et susceptibles d'une interprétation erronée par le serveur (&, +, %, ...) sont transformés en %XX où XX est leur code hexadécimal.
3. La ligne **Content-Length: 210** indique au serveur combien de caractères le client va lui envoyer une fois les entêtes HTTP terminés, c.a.d. après la ligne vide signalant la fin des entêtes.
4. Les données (210 caractères) :
   **R1=Oui&C1=un&C2=deux&txtSaisie=programmation+web&txtMdp=ceciestsecret&areaSaisie=les+bases+de+la%0D%0Aprogrammation+web&cmbValeurs=choix3&lst1=liste3&lst2=liste1&lst2=liste3&cmdRenvoyer=Envoyer&secret=uneValeur**

On remarque que les données transmises par POST le sont au même format que celle transmises par GET.

Y-a-t-il une méthode meilleure que l'autre ? Nous avons vu que si les valeurs d'un formulaire étaient envoyées par le navigateur avec la méthode GET, le navigateur affichait dans son champ *Adresse* l'URL demandée sous la forme *URL?param1=val1&param2=val2&...*. On peut voir cela comme un avantage ou un inconvénient :

- un avantage si on veut permettre à l'utilisateur de placer cette URL paramétrée dans ses liens favoris
- un inconvénient si on ne souhaite pas que l'utilisateur ait accès à certaines informations du formulaire tels, par exemple, les champs cachés

Par la suite, nous utiliserons quasi exclusivement la méthode POST dans nos formulaires.

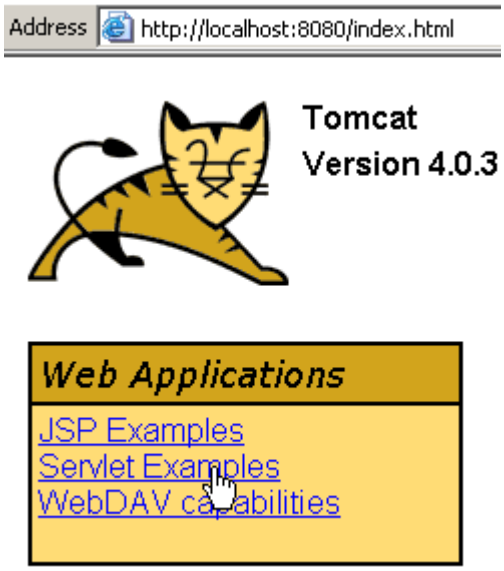
### *1.8.2.3 Récupération des valeurs d'un formulaire Web*

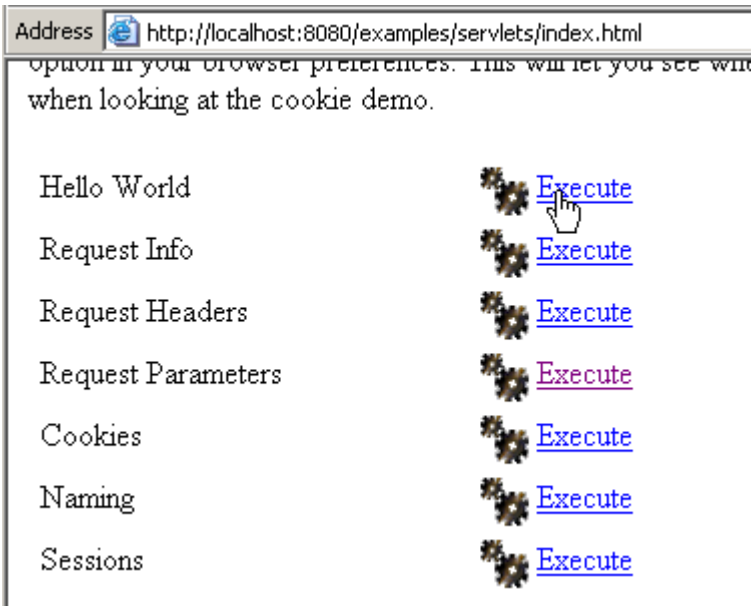
Une page statique demandée par un client qui envoie de plus des paramètres par POST ou GET ne peut en aucune façon récupérer ceux-ci. Seul un programme peut le faire et c'est lui qui se chargera alors de générer une réponse au client, une réponse qui sera dynamique et généralement fonction des paramètres reçus. C'est le domaine de la programmation web, domaine que nous abordons

plus en détail dans le chapitre suivant avec la présentation des technologies Java de programmation web : les **servlets** et les **pages JSP**.

# 2. Introduction aux servlets Java et pages JSP

On trouvera dans ce chapitre divers exemples de servlets et pages JSP. Ils ont été testés avec le serveur Tomcat. Celui-ci travaille sur le port 8080. En suivant les liens de la page d'accueil, on a accès à des exemples de servlets et pages JSP. Les exemples ci-dessous sont pour la plupart tirés des exemples de Tomcat. Pour les tester, il suffit de lancer Tomcat, de demander l'URL *http://localhost:8080* avec un navigateur et de suivre le lien des servlets.




## 2.1 Servlets Java

### 2.1.1 Envoyer un contenu HTML à un client Web

Nous examinons l'exemple *Hello World* ci-dessus. La servlet est la suivante :

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

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

A l'exécution de cette servlet, on obtient l'affichage suivant :


On notera les points suivants :

- il faut importer des classes spéciales pour les servlets :

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

La bibliothèque *javax.servlet* n'est pas toujours livrée en standard avec le jdk. Dans ce cas, on peut la récupérer directement sur le site de Sun.

- une servlet dérive de la classe *HttpServlet*

```
public class HelloWorld extends HttpServlet {
```

- Une requête GET faite à la servlet est traitée par la méthode *doGet*

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
```

- De même une requête POST faite à la servlet est traitée par la méthode doPost

```
    public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
```

- L'objet *HttpServletRequest request* est l'objet qui nous donne accès à la requête faite par le client Web. La réponse de la servlet sera faite via l'objet *HttpServletResponse response*

- L'objet *response* nous permet de fixer les entêtes http qui seront envoyés au client. Par exemple l'entête *Content-type: text/html* est ici fixé par :

```
        response.setContentType("text/html");
```

- Pour envoyer la réponse au client, la servlet utilise un flux de sortie que lui délivre l'objet *response* :

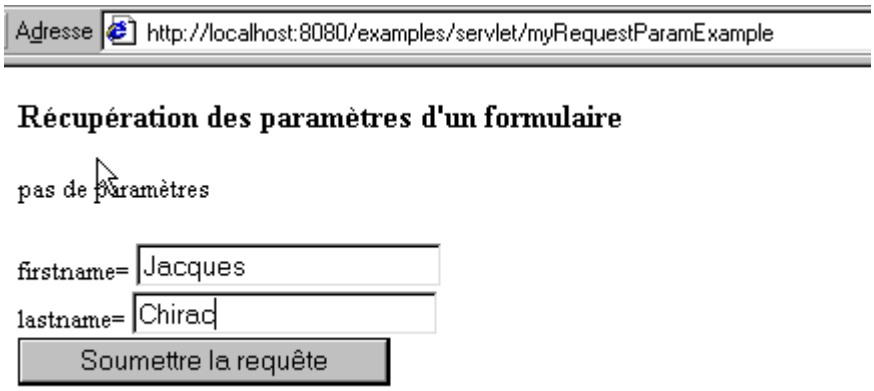
```
        PrintWriter out = response.getWriter();
```

- Une foix ce flux de sortie obtenu, le code HTML est écrit dedans et donc envoyé au client :

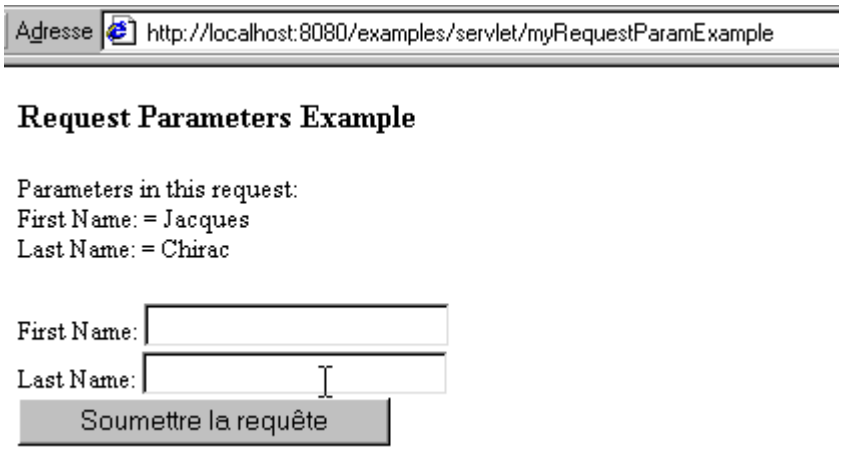
```
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello World!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Hello World!</h1>");
        out.println("</body>");
        out.println("</html>");
```

## 2.1.2 Récupérer les paramètres envoyés par un client web

L'exemple suivant montre comment une servlet peut récupérer des paramètres envoyés par le client web. Un formulaire de saisie :




La réponse envoyée par la servlet :




Le code source de la servlet est le suivant :

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

public class myRequestParamExample extends HttpServlet {

    String title="Récupération des paramètres d'un formulaire";

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h3>" + title + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        if (firstName != null || lastName != null) {
            out.println("firstname= " + firstName + "<br>");
            out.println("lastname= " + lastName);
        } else {
            out.println("pas de paramètres");
        }
        out.println("<P>");
        out.print("<form action=\"RequestParamExample\" method=\"POST\">");
        out.println("firstname= <input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("lastname= <input type=text size=20 name=lastname>");
        out.println("<br>");
```

```
        out.println("<input type=submit>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

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

On notera les nouveautés suivantes par rapport à l'exemple précédent :

- Les paramètres envoyés par le navigateur sont récupérés de la façon suivante :

```
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
```

La méthode *request.getParameter("nomParamètre")* rend le pointeur *null*, si le paramètre *nomParamètre* ne fait pas partie des paramètres envoyés par le client web.

- Le formulaire précise que le navigateur doit envoyer les paramètres par la méthode POST

```
        out.print("<form action=\"RequestParamExample\" method=\"POST\">");
```

- Les paramètres reçus seront traités par la méthode *doPost* de la servlet. Ici, cette méthode se contente d'appeler la méthode *doGet*. Ainsi cette servlet traite les valeurs du formulaire qu'elles soient envoyées par un GET ou un POST.

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

## 2.1.3 Récupérer les entêtes http envoyés par un client web

La servlet qui suit montre comment récupérer les entêtes http envoyés par le client web :


Le code source de la servlet est le suivant :

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

public class RequestHeaderExample extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println(name + " = " + value);
        }
    }
}
```

Points à noter :

- C'est l'objet *request* et sa méthode *getHeaderNames* qui nous donne accès aux entêtes http envoyés par le navigateur sous la forme d'une énumération :

```
Enumeration e = request.getHeaderNames();
```

- La méthode *request.getHeader("entête")* permet d'obtenir un entête http précis. L'exemple ci-dessus nous donne quelques-uns d'entre-eux. On se souviendra que les entêtes présentés ici sont envoyés par le navigateur. Le serveur a lui aussi ses propres entêtes http qui reprennent parfois ceux du navigateur. Les entêtes http envoyés par le navigateur ont pour objectif de renseigner le serveur sur les capacités du navigateur.

| Entête | Signification |
|---|---|
| `User-Agent` | identité du navigateur |
| `Accept` | les formats MIME acceptés par le navigateur. Ainsi *image/gif* signifie que le navigateur sait traiter les images au format GIF |
| `Host` | au format *hote:port*. Indique quelle machine et quel port le navigateur veut contacter. |
| `Accept-Encoding` | format d'encodage accepté par le navigateur pour les documents envoyés par le serveur. Ainsi si un serveur a un document sous une forme normale non compressée et une au format compressé gzip et que le navigateur a indiqué qu'il savait traiter le format gzip, alors le serveur pourra envoyer le document au format gzip pour économiser de la bande passante. |
| `Accept-language` | langues acceptées par le navigateur. Si un serveur dispose d'un même document en plusieurs langues, il en enverra un dont la langue est acceptée par le navigateur. |
| `Referer` | l'URL qui a été demandée par le navigateur |
| `Connection` | le mode de connexion demandée par le navigateur. *Keep-alive* veut dire que le serveur ne doir pas couper la connexion après avoir servi la page demandée au navigateur. Si ce dernier découvre que la page reçue contient des liens sur des images par exemples, il pourra faire de nouvelles requêtes au serveur pour les demander sans avoir besoin de créer une nouvelle connexion. C'est le navigateur qui prendra alors l'initiative de fermer la connexion lorsqu'il aura reçu tous les éléments de la page. |

## 2.1.4 Récupérer des informations d'environnement

La servlet qui suit montre comment accéder à des informations d'environnement d'exécution de la servlet. Certaines de celles-ci sont envoyées sous forme d'entêtes http par le navigateur et sont donc récupérables par la méthode précédente.


**Request Information Example**

| | |
|---|---|
| Method: | GET |
| Request URI: | /examples/servlet/RequestInfoExample |
| Protocol: | HTTP/1.1 |
| Path Info: | null |
| Remote Address: | 127.0.0.1 |

Le code de la servlet est le suivant :

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

public class RequestInfo extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Request Information Example</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h3>Request Information Example</h3>");
        out.println("Method: " + request.getMethod());
        out.println("Request URI: " + request.getRequestURI());
        out.println("Protocol: " + request.getProtocol());
        out.println("PathInfo: " + request.getPathInfo());
        out.println("Remote Address: " + request.getRemoteAddr());
        out.println("</body>");
        out.println("</html>");
    }


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

Les informations sont ici obtenues par diverses méthodes :

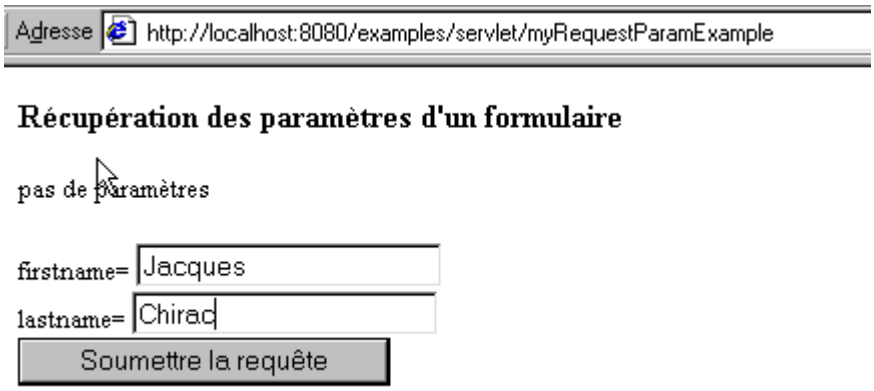
```
        out.println("Method: " + request.getMethod());
        out.println("Request URI: " + request.getRequestURI());
        out.println("Protocol: " + request.getProtocol());
        out.println("PathInfo: " + request.getPathInfo());
        out.println("Remote Address: " + request.getRemoteAddr());
```

Une liste de certaines des méthodes disponibles et leur signification est la suivante :

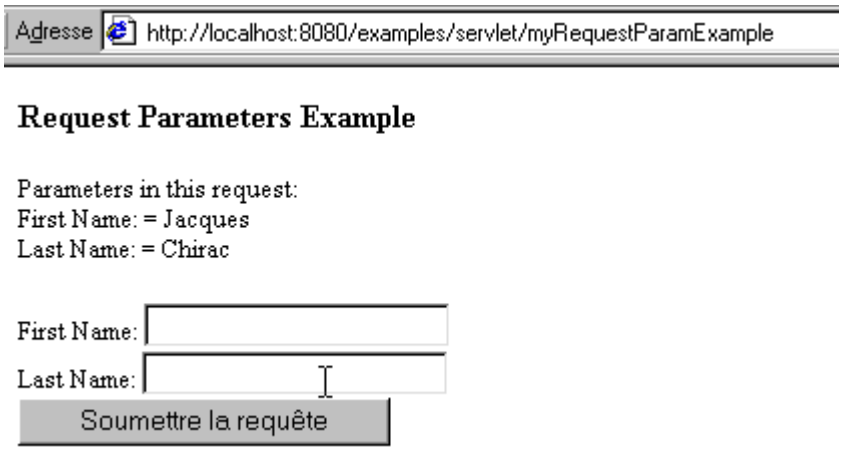
| méthode | signification |
|---|---|
| `getServerName()` | le nom du serveur Web |
| `getServerPort()` | le port de travail du serveur web |
| `getMethod()` | la méthode GET ou POST utilisée par le navigateur pour faire sa requête |
| `getRemoteHost()` | le nom de la machine client à partir de laquelle le navigateur a fait sa requête |
| `getRemoteAddr()` | l'adresse IP de cette même machine |
| `getContentType()` | le type de contenu envoyé par le navigateur (entête http *Content-type*) |
| `getContentLength()` | le nombre de caractères envoyés par le navigateur (entête http *Content-length*) |
| `getProtocol()` | la version du protocole http demandée par le navigateur |
| `getRequestURI()` | l'URI demandée par le navigateur. Correspond à la partie de l'URL placée après l'identification *hote:port* dans *http://hote:port/URI* |

## 2.1.5 Créer une servlet avec JBuilder, la déployer avec Tomcat

Nous décrivons maintenant comment créer et exécuter une servlet Java. On utilisera deux outils : Jbuilder pour compiler la servlet et Tomcat pour l'exécuter. Tomcat pourrait seul suffire. Néanmoins il offre des capacités de débogage limitées. Nous reprenons l'exemple développé précédemment qui affiche les paramètres reçus par le serveur. La servlet envoie tout d'abord le formulaire de saisie suivant :




La réponse envoyée par la servlet :




Le code source de la servlet est le suivant :

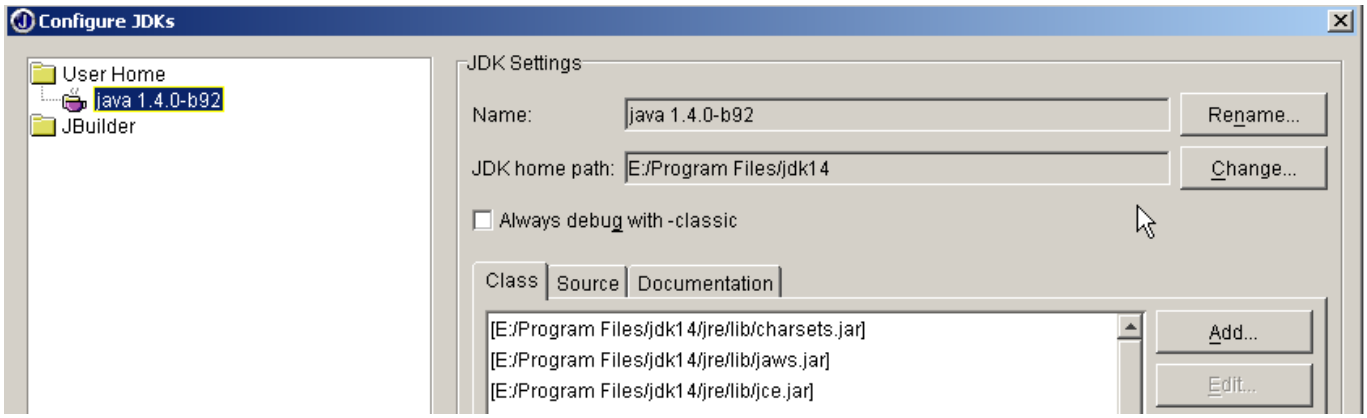
```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class myRequestParamExample extends HttpServlet {

    String title="Récupération des paramètres d'un formulaire";

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h3>" + title + "</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        if (firstName != null || lastName != null) {
            out.println("firstname= " + firstName + "<br>");
            out.println("lastname= " + lastName);
        } else {
```

```
            out.println("pas de paramètres");
        }
        out.println("<P>");
        out.print("<form action=\"RequestParamExample\" method=\"POST\">");
        out.println("firstname= <input type=text size=20 name=firstname>");
        out.println("<br>");
        out.println("lastname= <input type=text size=20 name=lastname>");
        out.println("<br>");
        out.println("<input type=submit>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }

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

}
```

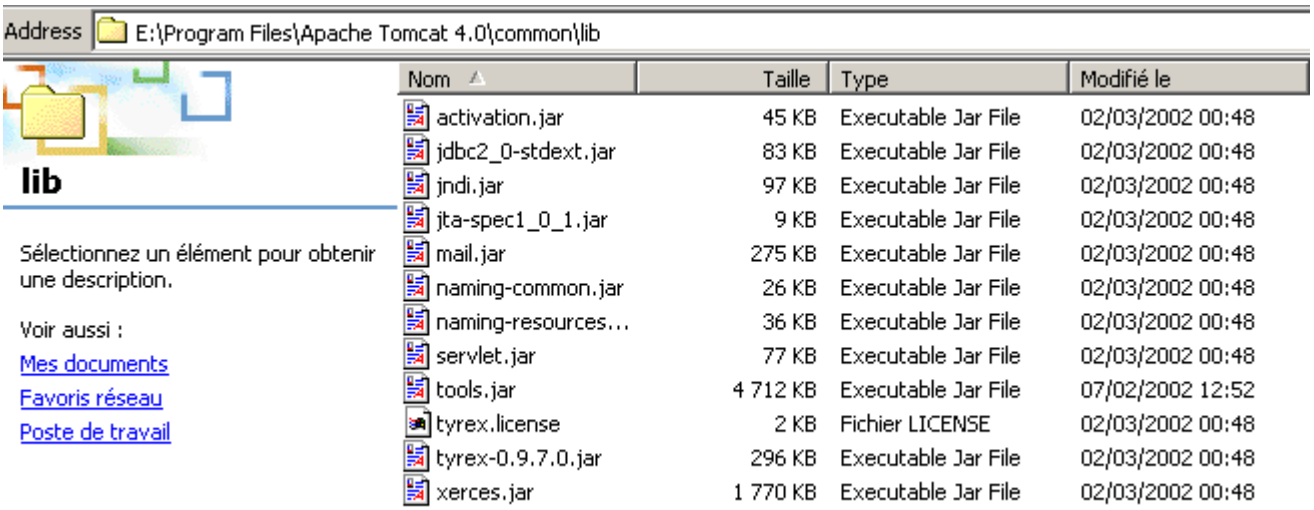
- créez un projet *myRequestParamExample* avec Jbuilder et y inclure le programme *myRequestParamExample.java* précédent.

- à la compilation, on peut rencontrer le problème suivant : votre JBuilder n'a pas forcément la bibliothèque *javax.servlet* nécessaire à la compilation des servlets. Dans ce cas, il vous faut configurer Jbuilder pour qu'il utilise des bibliothèques de classes supplémentaires. La méthode est décrite dans les annexes de ce document pour JBuilder 7. Nous la reprenons partiellement ici :

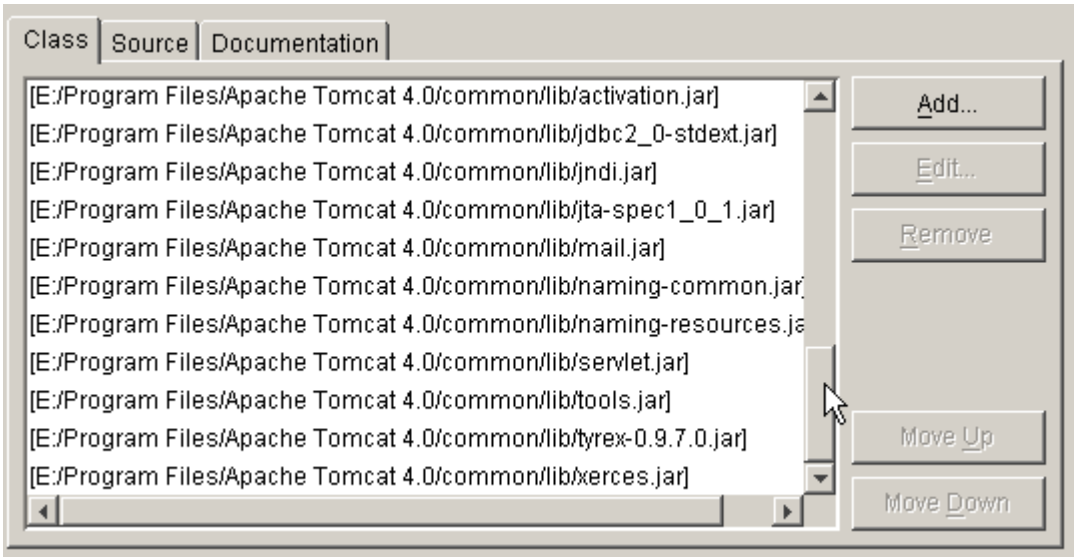
- activer l'option *Tools/Configure JDKs* ou (Options/Configurer les JDK)




Dans la partie *JDK Settings* ci-dessus, on a normalement dans le champ *Name* un JDK 1.3.1. Si vous avez un JDK plus récent, utilisez le bouton *Change* pour désigner le répertoire d'installation de ce dernier. Ci-dessus, on a désigné le répertoire *E:\Program Files\jdk14* où était installé un JDK 1.4. Désormais, JBuilder utilisera ce JDK pour ses compilations et exécutions. Dans la partie (Class, Source, Documentation) on a la liste de toutes les bibliothèques de classes qui seront explorées par JBuilder, ici les classes du JDK 1.4. Les classes de celui-ci ne suffisent pas pour faire du développement web en Java. Pour ajouter d'autres bibliothèques de classes on utilise le bouton *Add* et on désigne les fichiers *.jar* supplémentaires que l'on veut utiliser. Les fichiers .jar sont des bibliothèques de classes. Tomcat 4.x amène avec lui toutes les bibliothèques de classes nécessaires au développement web. Elles se trouvent dans *<tomcat>\common\lib* où *<tomcat>* est le répertoire d'installation de Tomcat :



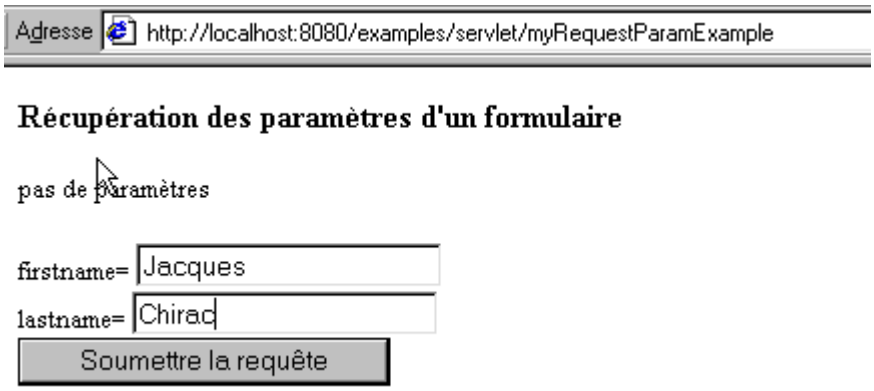

Avec le bouton *Add*, on va ajouter ces bibliothèques, une à une, à la liste des bibliothèques explorées par JBuilder :




A partir de maintenant, on peut compiler des programmes java conformes à la norme J2EE, notamment les servlets Java. Jbuilder ne sert qu'à la compilation, l'exécution étant ultérieurement assurée par Tomcat.

- maintenant vous pouvez compiler le programme *myRequestParamExample.java* et produire la servlet *myRequestParamExample.class*. Où placer cette servlet ? Si la configuration initiale de Tomcat n'a pas été changée, les *.class* des servlets doivent être placés dans *<tomcat>\webapps\examples\WEB-INF\classes* (Tomcat 4.x).

- vérifiez que Tomcat est lancé et avec un navigateur demandez l'URL *http://localhost:8080/examples/servlet/myRequestParamExample* :




## 2.1.6 Exemples

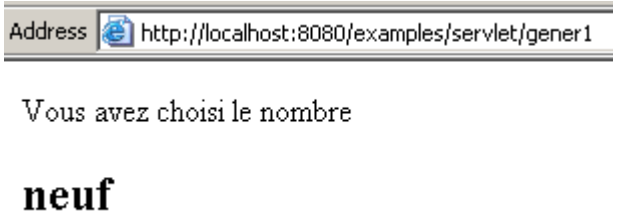
Pour les exemples qui suivent, nous avons utilisé la méthode décrite précédemment :

- compilation du source *XX.java* de la servlet avec Jbuilder
- déploiement de la servlet *XX.class* dans *<tomcat>\webapps\examples\WEB-INF\classes*
- Tomcat lancé, demander avec un navigateur l'URL *http://localhost:8080/examples/servlet/XX*

### *2.1.6.1 Génération dynamique de formulaire - 1*

Nous prenons comme exemple la génération d'un formulaire n'ayant qu'un contrôle : une liste. Le contenu de cette liste est construit dynamiquement avec des valeurs prises dans un tableau. Dans la réalité, elles sont souvent prises dans une base de données. Le formulaire est le suivant :


Si sur l'exemple ci-dessus, on fait *Envoyer*, on obtient la réponse suivante :




On remarquera que l'URL qui fait la réponse est la même que celle qui affiche le formulaire. Ici on a une servlet qui traite elle-même la réponse au formulaire qu'elle a envoyé. C'est un cas courant. Le code HTML du formulaire est le suivant :

```
<html>
  <head><title>Génération de formulaire</title></head>
  <body>
    <h3>Choisissez un nombre</h3><hr>
    <form method="POST">
      <select name="cmbValeurs" size="1">
        <option>zéro</option>
        <option>un</option>
        <option>deux</option>
```

```
        <option>trois</option>
        <option>quatre</option>
        <option>cinq</option>
        <option>six</option>
        <option>sept</option>
        <option>huit</option>
        <option>neuf</option>
      </select>
      <input type="submit" value="Envoyer">
    </form>
  </body>
</html>
```

On notera que les valeurs envoyées par le formulaire le sont par la méthode POST. Le code HTML de la réponse :

```
<html>
  <head><title>Voici ma réponse</title></head>
  <body>
    Vous avez choisi le nombre<h2>neuf</h2>
  </body>
</html>
```

Le code de la servlet qui génère ce formulaire et cette réponse est le suivant :

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

public class gener1 extends HttpServlet{
    // variables d'instance
    private String title="Génération d'un formulaire";
    private final String[] valeurs={"zéro","un","deux","trois","quatre","cinq","six",
      "sept","huit","neuf"};
    private final String HTML1=
        "<html>" +
         "<head>" +
           "<title>Génération de formulaire</title>"+
         "</head>" +
         "<body>" +
            "<h3>Choisissez un nombre</h3>"+
            "<hr>" +
            "<form method=\"POST\">";
   private final String HTML2="<input type=\"submit\" value=\"Envoyer\">";
   private final String HTML3="</form>\n</body>\n</html>";

    // GET
    public void doGet(HttpServletRequest request,HttpServletResponse response)
          throws IOException, ServletException{

      // on indique au client le type de document envoyé
      response.setContentType("text/html");
      // on envoie le formulaire
      PrintWriter out=response.getWriter();
      // début
      out.println(HTML1);
      // combo
      out.println("<select name=\"cmbValeurs\" size=\"1\">");
      for (int i=0;i<valeurs.length;i++){
        out.println("<option>"+valeurs[i]+"</option>");
      }//for
      out.println("</select>");
      // fin formulaire
      out.println(HTML2+HTML3);
    }//GET

   // POST
   public void doPost(HttpServletRequest request,HttpServletResponse response)
          throws IOException, ServletException{

      // on récupère le choix de l'utilisateur
      String choix=request.getParameter("cmbValeurs");
      if(choix==null) doGet(request,response);

      // on prépare la réponse
      String réponse="<html><head><title>Voici ma réponse</title></head>";
      réponse+="<body>Vous avez choisi le nombre <h2>"+choix+"</h2></body></html>";
      // on indique au client le type de document envoyé
      response.setContentType("text/html");
      // on envoie le formulaire
      PrintWriter out=response.getWriter();
      out.println(réponse);
```

```
    }//POST
}//classe
```

La méthode *doGet* sert à générer le formulaire. Il y a une partie dynamique qui est le contenu de la liste, contenu provenant ici d'un tableau. La méthode *doPost* sert à générer la réponse. Ici la seule partie dynamique est la valeur du choix fait par l'utilisateur dans la liste du formulaire. Cette valeur est obtenue par *request.getParameter("cmbValeurs")* où *cmbValeurs* est le nom HTML de la liste :

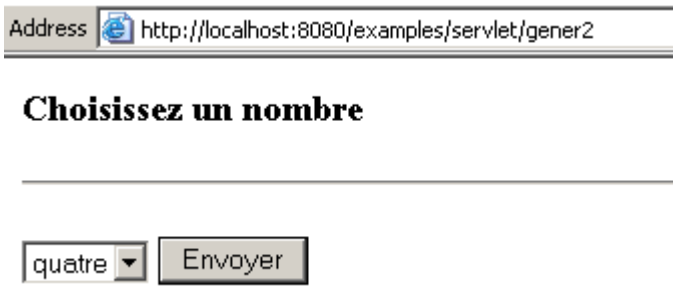
```
      <select name="cmbValeurs" size="1">
```

On notera pour terminer les points suivants :

- le navigateur envoie les valeurs du formulaire à la servlet qui a généré le formulaire parce que la balise <*form*> n'a pas d'attribut <*action*>. Dans ce cas, le navigateur envoie les données saisies dans le formulaire à l'URL qui l'a fourni.
- la balise <*form*> précise que les données du formulaire doivent être envoyées par la méthode POST. C'est pour cela que ces valeurs sont récupérées par la méthode *doPost* de la servlet.

## *2.1.6.2 Génération dynamique de formulaire - 2*

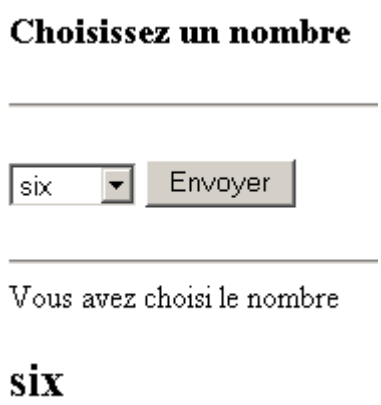
Nous reprenons l'exemple précédent en le modifiant de la façon suivante. Le formulaire proposé est toujours le même :




La réponse est elle différente :


Dans la réponse, on renvoie le formulaire, le nombre choisi par l'utilisateur étant indiqué dessous. Par ailleurs, ce nombre est celui qui apparaît comme sélectionné lorsque la liste est affichée. L'utilisateur peut alors choisir un autre nombre :


puis faire *Envoyer*. Il obtient la réponse suivante :




Le code la servlet appelée *gener2.java* est la suivante :

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

public class gener2 extends HttpServlet{
    // variables d'instance
    private String title="Génération d'un formulaire";
    private final String[] valeurs={"zéro","un","deux","trois","quatre","cinq","six",
      "sept","huit","neuf"};
    private final String HTML1=
        "<html>" +
          "<head>" +
            "<title>Génération de formulaire</title>"+
          "</head>" +
          "<body>" +
             "<h3>Choisissez un nombre</h3>"+
             "<hr>" +
             "<form method=\"POST\">";
    private final String HTML2="<input type=\"submit\" value=\"Envoyer\"></form>\n";
    private final String HTML3="</body>\n</html>";

    // GET
    public void doGet(HttpServletRequest request,HttpServletResponse response)
          throws IOException, ServletException{

      // on récupère l'éventuel choix de l'utilisateur
      String choix=request.getParameter("cmbValeurs");
      if(choix==null) choix="";

      // on indique au client le type de document envoyé
      response.setContentType("text/html");
      // on envoie le formulaire
      PrintWriter out=response.getWriter();
      // début
      out.println(HTML1);
      // combo
      out.println("<select name=\"cmbValeurs\" size=\"1\">");
```

```
      String selected="";
      for (int i=0;i<valeurs.length;i++){
        if(valeurs[i].equals(choix)) selected="selected"; else selected="";
        out.println("<option "+selected+">"+valeurs[i]+"</option>");
      }//for
      out.println("</select>");
      // suite formulaire
      out.println(HTML2);
      if(! choix.equals("")){
        // on affiche le choix de l'utilisateur
        out.println("<hr>Vous avez choisi le nombre <h2>"+choix+"</h2>");
      }//if
      // fin du formulaire
      out.println(HTML3);
    }//GET

    // POST
    public void doPost(HttpServletRequest request,HttpServletResponse response)
          throws IOException, ServletException{

      // on renvoie sur GET
      doGet(request,response);
    }//POST
  }//classe
```

La méthode *doGet* fait tout : elle élabore le formulaire qu'elle envoie au client et traite les valeurs que celui-ci renvoie. Les points à noter sont les suivants :

- on vérifie si le paramètre *cmbValeurs* a une valeur.
- si c'est le cas, lorsqu'on élabore le contenu de la liste, on compare chaque élément de celle-ci au choix de l'utilisateur pour mettre l'attribut *selected* à l'élément choisi par l'utilisateur : <*option selected*>*élément*</*option*>. Par ailleurs, on affiche sous le formulaire la valeur du choix.

## *2.1.6.3 Génération dynamique de formulaire - 3*

Nous reprenons le même problème que précédemment mais cette fois-ci les valeurs sont prises dans une base de données. Celle-ci est dans notre exemple une base MySQL :

- la base s'appelle **dbValeurs**
- son propriétaire est **admDbValeurs** ayant le mot de passe **mdpDbValeurs**
- la base a une unique table appelée **tvaleurs**
- cette table n'a qu'un champ entier appelé **valeur**

```
E:\Program Files\EasyPHP\mysql\bin>mysql --database=dbValeurs --user=admDbValeurs --password=mdpDbVa
leurs

mysql> show tables;
+---------------------+
| Tables_in_dbValeurs |
+---------------------+
| tvaleurs            |
+---------------------+
1 row in set (0.00 sec)

mysql> describe tvaleurs;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| valeur | int(11) |      |     | 0       |       |
+--------+---------+------+-----+---------+-------+

mysql> select * from tvaleurs;
+--------+
| valeur |
+--------+
|      0 |
|      1 |
|      2 |
|      3 |
|      4 |
|      6 |
|      5 |
```

```
|       7 |
|       8 |
|       9 |
+---------+
10 rows in set (0.00 sec)
```

La base MySQL **dbValeurs** a été rendu accessible par un pilote ODBC pour MySQL. Son nom DSN (Data Source Name) est **odbc-valeurs**. Le code de la servlet est le suivant :

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

public class gener3 extends HttpServlet{
    // le titre de la page
    private final String title="Génération d'un formulaire";
    // la base de données des valeurs de liste
    private final String DSNValeurs="odbc-valeurs";
    private final String admDbValeurs="admDbValeurs";
    private final String mdpDbValeurs="mdpDbValeurs";
    // valeurs de liste
    private String[] valeurs=null;
    // msg d'erreur
    private String msgErreur=null;
    // code HTML
    private final String HTML1=
        "<html>" +
          "<head>" +
            "<title>Génération de formulaire</title>"+
          "</head>" +
          "<body>" +
             "<h3>Choisissez un nombre</h3>"+
             "<hr>" +
             "<form method=\"POST\">";
    private final String HTML2="<input type=\"submit\" value=\"Envoyer\"></form>\n";
    private final String HTML3="</body>\n</html>";

    // GET
    public void doGet(HttpServletRequest request,HttpServletResponse response)
          throws IOException, ServletException{

      // on indique au client le type de document envoyé
      response.setContentType("text/html");
      // flux de sortie
      PrintWriter out=response.getWriter();

      // l'initialisation de la servlet s'est-elle bien passée ?
      if (msgErreur!=null){
        // il y a eu une erreur - on génère une page d'erreur
        out.println("<html><head><title>"+title+"</title></head>");
        out.println("<body><h3>Application indisponible ("+msgErreur+
                    ")</h3></body></html>");
        return;
      }//if

      // on récupère l'éventuel choix de l'utilisateur
      String choix=request.getParameter("cmbValeurs");
      if(choix==null) choix="";

      // on envoie le formulaire
      // début
      out.println(HTML1);
      // combo
      out.println("<select name=\"cmbValeurs\" size=\"1\">");
      String selected="";
      for (int i=0;i<valeurs.length;i++){
        if(valeurs[i].equals(choix)) selected="selected"; else selected="";
        out.println("<option "+selected+">"+valeurs[i]+"</option>");
      }//for
      out.println("</select>");
      // suite formulaire
      out.println(HTML2);
      if(! choix.equals("")){
        // on affiche le choix de l'utilisateur
        out.println("<hr>Vous avez choisi le nombre <h2>"+choix+"</h2>");
      }//if
      // fin du formulaire
      out.println(HTML3);
    }//GET
```

```
  // POST
  public void doPost(HttpServletRequest request,HttpServletResponse response)
       throws IOException, ServletException{

    // on renvoie sur GET
    doGet(request,response);
  }//POST

  // initialisation de la servlet
  public void init(){
    // remplit le tableau des valeurs à partir d'une base de données ODBC
    // de nom DSN : DSNvaleurs
    Connection connexion=null;
    Statement st=null;
    ResultSet rs=null;
    try{
      // connexion à la base ODBC
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      connexion=DriverManager.getConnection("jdbc:odbc:"+DSNValeurs,admDbValeurs,mdpDbValeurs);
      // objet Statement
      st=connexion.createStatement();
      // exécution requête select pour récupérer les valeurs
      rs=st.executeQuery("select valeur from Tvaleurs");
      // les valeurs sont récupérées et mises dans un tableau dynamique
      ArrayList lstValeurs=new ArrayList();
      while(rs.next()){
        // on enregistre la valeur dans la liste
        lstValeurs.add(rs.getString("valeur"));
      }//while
      // transformation liste --> tableau
      valeurs=new String[lstValeurs.size()];
      for (int i=0;i<lstValeurs.size();i++){
        valeurs[i]=(String)lstValeurs.get(i);
      }
    }catch(Exception ex){
      // problème
      msgErreur=ex.getMessage();
    }finally{
      try{rs.close();}catch(Exception ex){}
      try{st.close();}catch(Exception ex){}
      try{connexion.close();}catch(Exception ex){}
    }//try
  }//init
}//classe
```

Les points importants à noter sont les suivants :

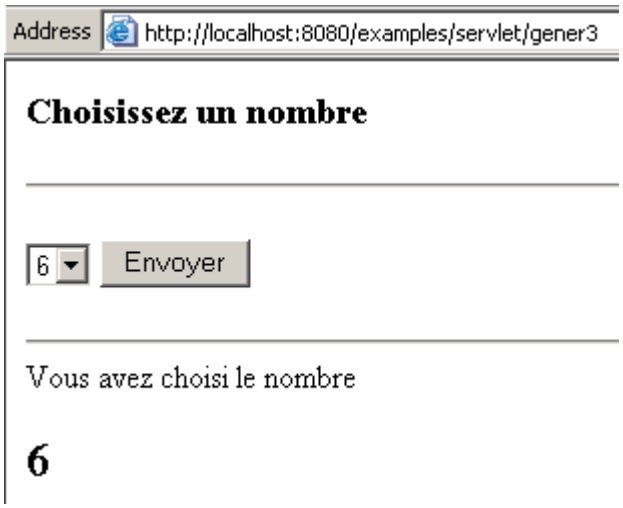
1. Une servlet peut être initialisée par une méthode dont la signature doit être **public void init()**. Cette méthode n'est exécutée qu'au chargement initial de la servlet
2. Une fois chargée, une servlet reste en mémoire tout le temps. Cela signifie que lorsqu'elle a servi un client, elle n'est pas déchargée. Elle répond ainsi plus vite aux requêtes des clients.
3. Dans notre servlet, une liste de valeurs doit être cherchée dans une base de données. Cette liste ne changeant pas au cours du temps, la méthode **init** est le moment idéal pour la récupérer. La base n'est ainsi accédée qu'une fois par la servlet, au moment du chargement initial de celle-ci, et non pas à chaque requête d'un client.
4. L'accès à une base de données peut échouer. La méthode *init* de notre servlet positionne un message d'erreur *msgErreur* en cas d'échec. Ce message est testé dans la méthode *doGet* et s'il y a eu erreur *doGet* génère une page la signalant.
5. L'écriture de la méthode **init** utilise un accès classique à une base de données avec les pilotes Odbc-Jdbc. Si besoin est, le lecteur est invité à revoir les méthodes d'accès aux bases de données JDBC.

Lorsqu'on exécute la servlet et que le serveur MySQL n'a pas été lancé, on a la page d'erreur suivante :


**Application indisponible ([Microsoft][ODBC Driver Manager] Data source name not found and no default driver specified)**

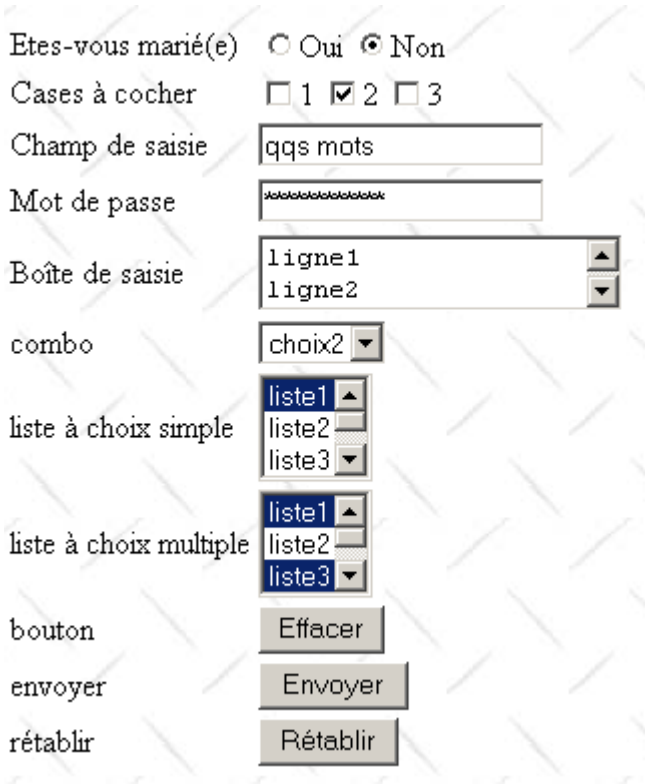
Si maintenant, on lance le serveur MySQL, on obtient la page :


Si on choisit le nombre 6 et qu'on "envoie" :




### *2.1.6.4 Récupérer les valeurs d'un formulaire*

Nous reprenons un exemple déjà rencontré, celui du formulaire web suivant :



Le code HTML du formulaire **balises2.htm** est le suivant :

```
<html>

  <head>
    <title>balises</title>
    <script language="JavaScript">
      function effacer(){
        alert("Vous avez cliqué sur le bouton Effacer");
      }//effacer
    </script>
  </head>

  <body background="/images/standard.jpg">
...
```

```
    <form method="POST" action="http://localhost:8080/examples/servlet/parameters">
```

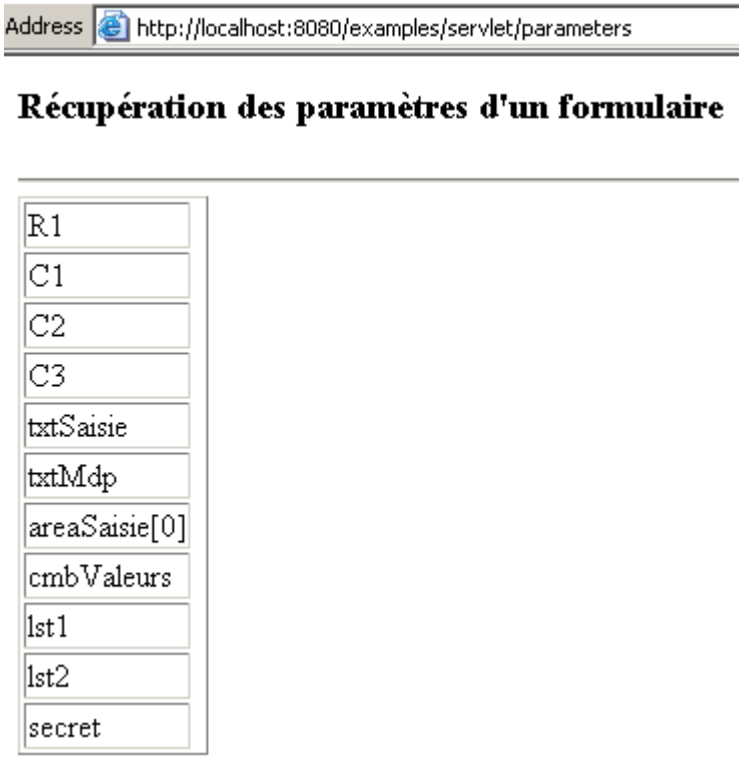
```
      <table border="0">
        <tr>
          <td>Etes-vous marié(e)</td>
          <td>
           <input type="radio" value="Oui" name="R1">Oui
           <input type="radio" name="R1" value="non" checked>Non
          </td>
        </tr>
        <tr>
          <td>Cases à cocher</td>
          <td>
           <input type="checkbox" name="C1" value="un">1
           <input type="checkbox" name="C2" value="deux" checked>2
           <input type="checkbox" name="C3" value="trois">3
          </td>
        </tr>
        <tr>
          <td>Champ de saisie</td>
          <td>
           <input type="text" name="txtSaisie" size="20" value="qqs mots">
          </td>
        </tr>
        <tr>
          <td>Mot de passe</td>
          <td>
           <input type="password" name="txtMdp" size="20" value="unMotDePasse">
          </td>
        </tr>
        <tr>
          <td>Boîte de saisie</td>
          <td>
            <textarea rows="2" name="areaSaisie" cols="20">
ligne1
ligne2
ligne3
</textarea>
          </td>
        </tr>
        <tr>
          <td>combo</td>
          <td>
              <select size="1" name="cmbValeurs">
                <option>choix1</option>
                <option selected>choix2</option>
                <option>choix3</option>
              </select>
          </td>
        </tr>
        <tr>
          <td>liste à choix simple</td>
          <td>
              <select size="3" name="lst1">
                <option selected>liste1</option>
                <option>liste2</option>
                <option>liste3</option>
                <option>liste4</option>
                <option>liste5</option>
              </select>
          </td>
        </tr>
        <tr>
          <td>liste à choix multiple</td>
          <td>
              <select size="3" name="lst2" multiple>
                <option selected>liste1</option>
                <option>liste2</option>
```

```
                <option selected>liste3</option>
                <option>liste4</option>
                <option>liste5</option>
              </select>
            </td>
          </tr>
          <tr>
            <td>bouton</td>
            <td>
                <input type="button" value="Effacer" name="cmdEffacer" onclick="effacer()">
            </td>
          </tr>
          <tr>
            <td>envoyer</td>
            <td>
                <input type="submit" value="Envoyer" name="cmdRenvoyer">
            </td>
          </tr>
          <tr>
            <td>rétablir</td>
            <td>
                <input type="reset" value="Rétablir" name="cmdRétablir">
            </td>
          </tr>
        </table>
        <input type="hidden" name="secret" value="uneValeur">
      </form>
    </body>
</html>
```

La balise *<form>* du formulaire a été définie comme suit :

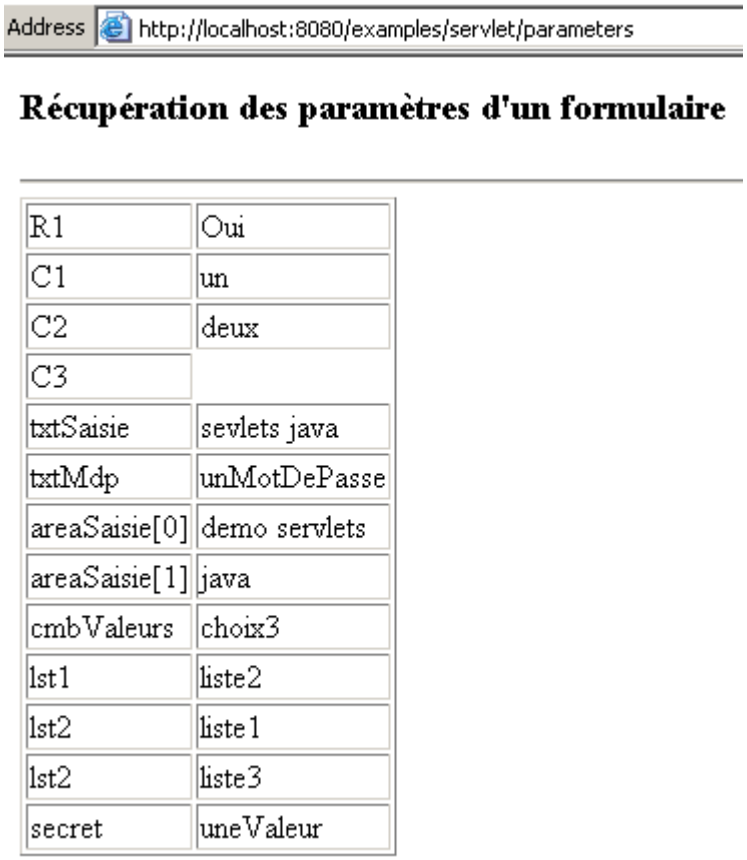
```
    <form method="POST" action="http://localhost:8080/examples/servlet/parameters">
```

Le navigateur "postera" les valeurs du formulaire à l'URL *http://localhost:8080/examples/servlet/parameters* qui est l'URL d'une servlet gérée par Tomcat et qui affiche les valeurs du formulaire précédent. Si on appelle la servlet *parameters* directement, on a les résultats suivants :




Si le formulaire *balises2.htm* saisi est celui-ci :


et qu'on appuie sur le bouton Envoyer (de type *submit*), la servlet *parameters* est cette fois appelée avec des paramètres. Elle renvoie alors la réponse suivante :




On retrouve bien dans cette réponse, les valeurs saisies dans le formulaire. Le code de la servlet est la suivante :

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

public class parameters extends HttpServlet{
    // variables d'instance
    String title="Récupération des paramètres d'un formulaire";

    private String getParameter(HttpServletRequest request, String contrôle){
      // rend la valeur request.getParameter(contrôle) ou "" si elle n'existe pas
      String valeur=request.getParameter(contrôle);
      if(valeur==null) return ""; else return valeur;
    }//getParameter

    // GET
    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException, ServletException
    {
      // on commence par récupérer les paramètres du formulaire
      String R1=getParameter(request,"R1");
      String C1=getParameter(request,"C1");
      String C2=getParameter(request,"C2");
      String C3=getParameter(request,"C3");
      String txtSaisie=getParameter(request,"txtSaisie");
      String txtMdp=getParameter(request,"txtMdp");
      String areaSaisie=getParameter(request,"areaSaisie");
      String[] lignes=areaSaisie.split("\\r\\n");
      String cmbValeurs=getParameter(request,"cmbValeurs");
      String lst1=getParameter(request,"lst1");
      String[] lst2=request.getParameterValues("lst2");
      String secret=getParameter(request,"secret");

      // on indique le contenu du document
        response.setContentType("text/html");
      // on envoie le document
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h3>" + title + "</h3>");
        out.println("<hr>");
        out.println("<table border=\"1\">");
        out.println("<tr><td>R1</td><td>"+R1+"</td></tr>");
        out.println("<tr><td>C1</td><td>"+C1+"</td></tr>");
        out.println("<tr><td>C2</td><td>"+C2+"</td></tr>");
        out.println("<tr><td>C3</td><td>"+C3+"</td></tr>");
        out.println("<tr><td>txtSaisie</td><td>"+txtSaisie+"</td></tr>");
        out.println("<tr><td>txtMdp</td><td>"+txtMdp+"</td></tr>");
        for(int i=0;i<lignes.length;i++)
          out.println("<tr><td>areaSaisie["+i+"]</td><td>"+lignes[i]+"</td></tr>");
        out.println("<tr><td>cmbValeurs</td><td>"+cmbValeurs+"</td></tr>");
        out.println("<tr><td>lst1</td><td>"+lst1+"</td></tr>");
        if(lst2==null)
          out.println("<tr><td>lst2</td><td></td></tr>");
        else
          for(int i=0;i<lst2.length;i++)
            out.println("<tr><td>lst2</td><td>"+lst2[i]+"</td></tr>");
        out.println("<tr><td>secret</td><td>"+secret+"</td></tr>");
        out.println("</body>");
        out.println("</html>");
    }

    // POST
    public void doPost(HttpServletRequest request,HttpServletResponse response)
        throws IOException, ServletException
    {
      // renvoie sur GET
      doGet(request,response);
    }
}
```

On retrouve dans ce code les techniques présentées précédemment dans un autre exemple. On notera deux points :

1. le contrôle *lst2* est une liste à sélection multiple et donc plusieurs éléments peuvent être sélectionnés. C'est le cas dans notre exemple où les éléments *liste1* et *liste3* ont été sélectionnés. Les valeurs de *lst2* ont été transmises par le navigateur au serveur sous la forme *lst2=liste1&lst2=liste3*. La servlet Java peut récupérer ces valeurs dans un tableau avec la méthode *getParameterValues* : ici *request.getParameterValues("lst2")* fournit un tableau de 2 chaînes de caractères ["liste1","liste3"].

2. le contrôle *areaSaisie* est un champ de saisie multilignes. *request.getParameter("areaSaisie")* donne le contenu du champ sous la forme d'une unique chaîne de caractères. Si dans celle-ci, on veut récupérer les différentes lignes qui la forment on pourra utiliser la fonction *split* de la classe *String*. Le code suivant

```
String areaSaisie=getParameter(request,"areaSaisie");
String[] lignes=areaSaisie.split("\\r\\n");
```

récupère les lignes du champ de saisie. Ces lignes sont terminées par les caractères \r\n (0D0A).

Pour faire les tests on a :

- construit et compilé la servlet *parameters* avec JBuilder comme il a été expliqué précédemment
- placé la classe générée dans *<tomcat>\webapps\examples\WEB-INF\classes* où *<tomcat>* est le répertoire d'installation de Tomcat.
- demandé l'URL *http://localhost:81/html/balises2.htm* dont le code a été présenté plus haut
- rempli le formulaire et appuyé sur le bouton *Envoyer*.

## *2.1.6.5 Récupérer les entêtes HTTP d'un client web*

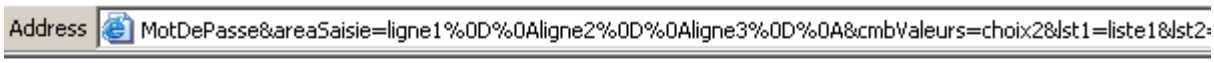
Nous reprenons le même exemple que précédemment mais en réponse au client web qui a envoyé les valeurs du formulaire, nous lui envoyons les entêtes HTTP qu'il a envoyés en même temps. Nous introduisons un seul changement dans notre formulaire :

```
    <form method="GET" action="http://localhost:8080/examples/servlet/headers">
```

Les valeurs du formulaires seront envoyées par la méthode GET à une servlet java appelée *headers* placée dans *<tomcat>\webapps\examples\WEB-INF\classes*. La servlet *headers* a été construite et compilée avec JBuilder :

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

public class headers extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
    {
      // on fixe la nature du document
        response.setContentType("text/html");
      // on obtient un flux d'écriture
        PrintWriter out = response.getWriter();
      // affichage liste des entêtes HTTP
        Enumeration e = request.getHeaderNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = request.getHeader(name);
            out.println("<b>"+name + "</b> = " + value + "<br>");
        }
    }//GET

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

On demande l'URL *http://localhost:81/html/balises2.htm* et on fait *Envoyer* sans modifier le formulaire. On obtient la réponse suivante :




**accept** = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, application/vn
**referer** = http://localhost:81/html/balises3.htm
**accept-language** = fr
**accept-encoding** = gzip, deflate
**user-agent** = Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
**host** = localhost:8080
**connection** = Keep-Alive

On remarquera l'URL paramétrée présente dans le champ *Address* du navigateur qui montre la façon (GET) utilisée pour transmettre les paramètres. Nous reprenons le même exemple mais en modifiant la façon d'envoyer les paramètres (POST) :

```
    <form method="POST" action="http://localhost:8080/examples/servlet/headers">
```

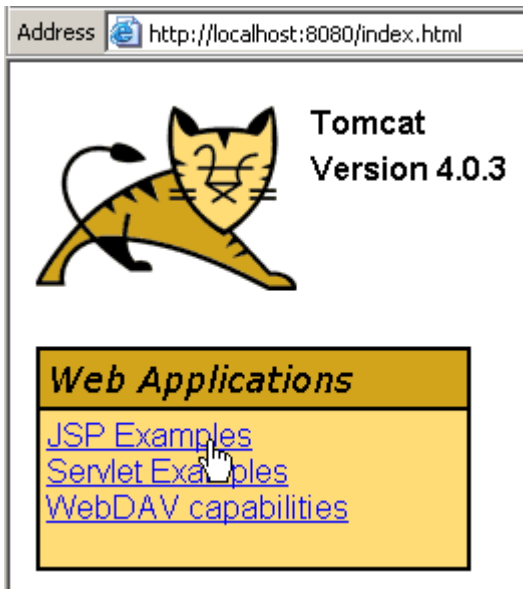
On obtient la nouvelle réponse suivante :

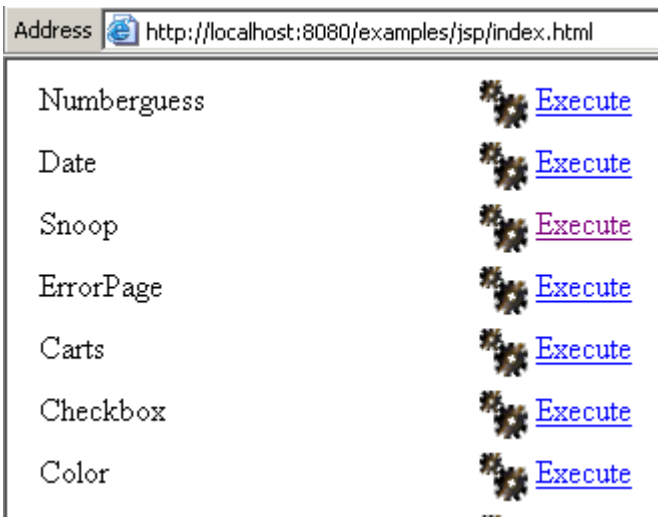

**accept** = image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, application/vnd.ms-powerpoint, application/vnd.ms-excel, */*
**referer** = http://localhost:81/html/balises3.htm
**accept-language** = fr
**content-type** = application/x-www-form-urlencoded
**accept-encoding** = gzip, deflate
**user-agent** = Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)
**host** = localhost:8080
**content-length** = 192
**connection** = Keep-Alive
**cache-control** = no-cache

On remarquera les entêtes HTTP *content-type* et *content-length* caractéristiques d'un envoi par POST. Par ailleurs, on notera que dans le champ *Address* du navigateur, les valeurs du formulaire n'apparaissent plus.

## 2.2 Pages JSP

Les pages JSP (Java Server Pages) sont une autre façon d'écrire des applications serveurs web. En fait ces pages JSP sont traduites en servlets avant d'être exécutées et on retrouve alors la technologie des servlets. Les pages JSP permettent de mieux mettre en relief la structure des pages HTML générées. Nous présentons ci-dessous des exemples dont certains sont accessibles en suivant le lien JSP de la page d'accueil de Tomcat :



### 2.2.1 Récupérer des informations d'environnement

Nous reprenons ici un exemple déjà traité avec une servlet : afficher les variables d'environnement d'une servlet. C'est l'exemple *snoop* des exemples JSP :


# Request Information

JSP Request Method: GET
Request URI: /examples/jsp/snp/snoop.jsp
Request Protocol: HTTP/1.1
Servlet path: /jsp/snp/snoop.jsp
Path info: null
Path translated: null
Query string: null
Content length: -1
Content type: null
Server name: localhost
Server port: 8080
Remote user: null
Remote address: 127.0.0.1
Remote host: 127.0.0.1
Authorization scheme: null

Le code source de la page JSP se trouve dans *<tomcat>\jakarta-tomcat\examples\jsp\snp\snoop.jsp* (Tomcat 3.x) ou *<tomcat>\examples\jsp\snp\snoop.jsp* (Tomcat 4.x)

```
<html>
<!--
  Copyright (c) 1999 The Apache Software Foundation.  All rights
  reserved.
-->

  <body bgcolor="white">
    <h1> Request Information </h1>
    <font size="4">
      JSP Request Method: <%= request.getMethod() %>
      <br>
      Request URI: <%= request.getRequestURI() %>
      <br>
      Request Protocol: <%= request.getProtocol() %>
      <br>
      Servlet path: <%= request.getServletPath() %>
      <br>
      Path info: <%= request.getPathInfo() %>
      <br>
      Path translated: <%= request.getPathTranslated() %>
      <br>
      Query string: <%= request.getQueryString() %>
      <br>
      Content length: <%= request.getContentLength() %>
      <br>
      Content type: <%= request.getContentType() %>
      <br>
      Server name: <%= request.getServerName() %>
      <br>
      Server port: <%= request.getServerPort() %>
      <br>
      Remote user: <%= request.getRemoteUser() %>
      <br>
```

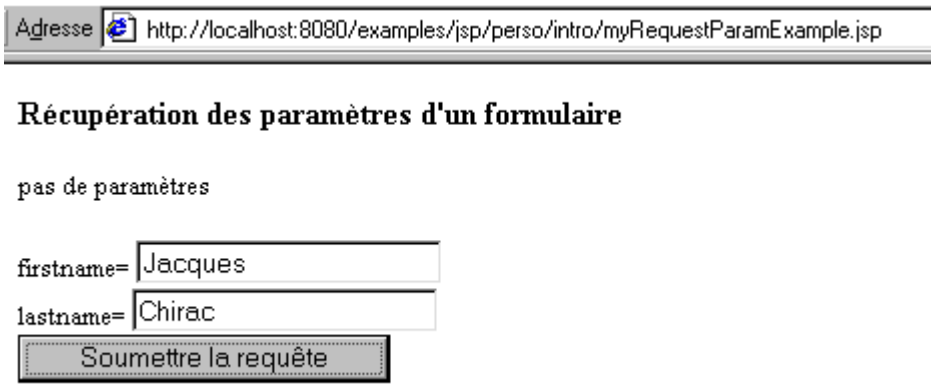
```
      Remote address: <%= request.getRemoteAddr() %>
      <br>
      Remote host: <%= request.getRemoteHost() %>
      <br>
      Authorization scheme: <%= request.getAuthType() %>
      <hr>
      The browser you are using is <%= request.getHeader("User-Agent") %>
      <hr>
    </font>
  </body>
</html>
```

On note les points suivants :

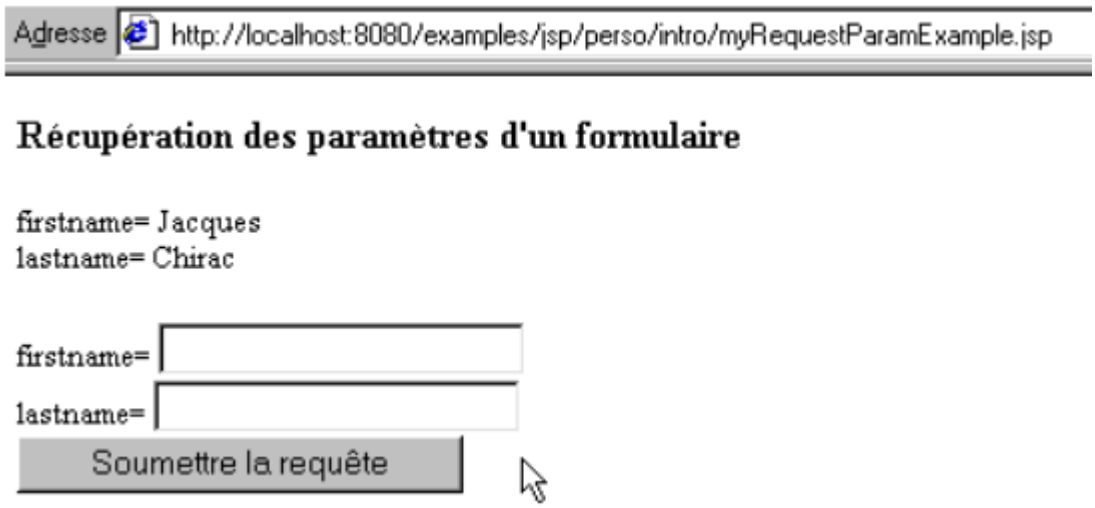
- on a là un code qui ressemble fort à du HTML. On y trouve cependant des balises <%= *expression* %> qui sont propres au langage JSP. Le compilateur JSP remplace dans le texte HTML l'intégralité de la balise par la valeur de *expression*.
- cet exemple utilise les méthodes de l'objet Java *request* qui est l'objet *request* déjà rencontré dans l'étude des servlets. C'est donc un objet *HttpServletRequest*. Ainsi la balise <%= *request.getRemoteHost()* %> sera remplacée dans le code HTML par le nom de la machine du client web qui a fait la requête.
- on peut arriver au même résultat avec une servlet mais ici la structure de la page web est plus évidente.

## 2.2.2 Récupérer les paramètres envoyés par le client web

Nous reprenons ici l'exemple déjà étudié avec une servlet. Il est présenté un formulaire au navigateur :




En réponse à la requête ci-dessus, le navigateur reçoit la page suivante :




Le code de la page JSP est le suivant :

```
<%
  // variables locales à la procédure principale
  String title="Récupération des paramètres d'un formulaire";
```

```
  String firstName = request.getParameter("firstname");
  String lastName = request.getParameter("lastname");
%>
<!-- code HTML -->
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body bgcolor="white">
    <h3><%= title %></h3>
    <%
      if (firstName != null || lastName != null) {
        out.println("firstname= " + firstName + "<br>");
        out.println("lastname= " + lastName);
      } else {
        out.println("pas de paramètres");
      }
    %>
    <P>
    <form method="POST">
      firstname= <input type="text" size="20" name="firstname">
      <br>
      lastname= <input type="text" size="20" name="lastname">
      <br>
      <input type="submit">
    </form>
  </body>
</html>
```

- Si on retrouve la balise ***<%= expression %>*** déjà rencontrée dans l'exemple précédent, une nouvelle balise apparaît ***<%*** *instructions Java;* ***%>***. La balise <% introduit du code Java. Ce code se termine à la rencontre de la balise de fermeture de code %>.

- L'ensemble du code précédent (HTML + JSP) va faire l'objet d'une conversion en servlet Java. Il sera enfermé dans une unique méthode, appelée méthode principale de la page JSP. C'est pourquoi, les variables java déclarées au début de la page JSP sont connues dans les autres portions de code JSP qui parsèment le code HTML : ces variables et portions de code feront partie de la même méthode Java. Mais si notre code JSP devait contenir des méthodes, les variables *title*, *firstname* et *lastname* n'y seraient pas connues à cause de l'étanchéité entre méthodes. Il faudrait en faire des variables globales ou les passer en paramètres aux méthodes. Nous y reviendrons.

- Pour inclure des parties dynamiques dans le code HTML deux méthodes sont possibles : <%= *expression* %> ou *out.println(expression)*. L'objet *out* est un flux de sortie analogue à celui de même nom rencontré dans les exemples de servlets mais pas du même type : c'est un objet *JspWriter* et non un *PrintWriter*. Il permet d'écrire dans le flux HTML avec le méthodes *print* et *println*.

- La page JSP reflète mieux la structure de la page HTML générée que la servlet équivalente.

## 2.2.3 Les balises JSP

Voici une liste de balises qu'on peut rencontrer dans une page JSP et leur signification.

| balise | signification |
|---|---|
| `<!-- commentaire -->` | commentaire HTML. Est envoyé au client. |
| `<%-- commentaire --%>` | commentaire JSP. N'est pas envoyé au client. |
| `<%! déclarations, méthodes %>` | déclare des variables globales et méthodes. Les variables seront connues dans toutes les méthodes |
| `<%= expression %>` | la valeur de expression sera intégrée dans la page HTML à la place de la balise |
| `<% code Java %>` | contient du code Java qui fera partie de la méthode principale de la page JSP |
| `<%@ page attribut1=valeur1 attribut2=valeur2 … %>` | fixe des attributs pour la page JSP. Par exemple :<br>*import="java.util.*,java.sql.*"* pour préciser les bibliothèques nécessaires à la page JSP<br>*extends="uneClasseParent"* pour faire dériver la page JSP d'une autre classe |

## 2.2.4 Les objets implicites JSP

Dans les exemples précédents, nous avons rencontré deux objets non déclarés : *request* et *out*. Ce sont deux des objets qui sont automatiquement définis dans la servlet dans laquelle est convertie la page JSP. On les appelle des objets implicites ou prédéfinis. Il en existe d'autres mais ce sont les plus utilisés avec l'obet *response* :

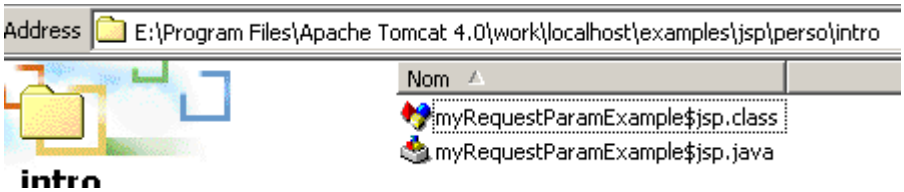
| objet | signification |
|---|---|
| `HttpServletRequest request` | l'objet à partir duquel on a accès à la requête du client Web (*getParameter, getParameterNames, getParameterValues*) |
| `HttpServletResponse response` | l'objet avec lequel on peut construire la réponse du serveur Web à son client. Permet de fixer les entêtes http à envoyer au client Web. |
| `JspWriter out` | le flux de sortie qui nous permet d'envoyer du code HTML au client (*print,println*) |

## 2.2.5 La transformation d'une page JSP en servlet

Reprenons le code JSP de *myRequestParamExample.jsp* :

```
<%
  // variables locales à la procédure principale
  String title="Récupération des paramètres d'un formulaire";
  String firstName = request.getParameter("firstname");
  String lastName = request.getParameter("lastname");
%>

<!-- code HTML -->
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body bgcolor="white">
    <h3><%= title %></h3>
    <%
      if (firstName != null || lastName != null) {
        out.println("firstname= " + firstName + "<br>");
        out.println("lastname= " + lastName);
      } else {
        out.println("pas de paramètres");
      }
    %>
    <P>
    <form method="POST">
      firstname= <input type="text" size="20" name="firstname">
      <br>
      lastname= <input type="text" size="20" name="lastname">
      <br>
      <input type="submit">
    </form>
  </body>
</html>
```

Lorsque le navigateur demande cette page JSP au serveur Tomcat, celui-ci va la transformer en servlet. Si l'URL demandée est *http://localhost:8080/examples/jsp/perso/intro/myRequestParamExample.jsp*, Tomcat 4.x va placer la servlet générée dans le répertoire *<tomcat>\work\localhost\examples\jsp\perso\intro* :




On retrouve dans ce nom l'URL *http://localhost:8080/examples/jsp/perso/intro/myRequestParamExample.jsp* de la page JSP. On voit ci-dessus qu'on a accès au code java de la servlet générée pour la page JSP. Dans notre exemple, c'est le suivant :

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;


public class myRequestParamExample$jsp extends HttpJspBase {
```

```
    static {
    }
    public myRequestParamExample$jsp( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws org.apache.jasper.runtime.JspException {
    }

    public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {

        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();


                  // variables locales à la procédure principale
                  String title="Récupération des paramètres d'un formulaire";
                  String firstName = request.getParameter("firstname");
                  String lastName = request.getParameter("lastname");

                out.write("\r\n\r\n<!-- code HTML -->\r\n<html>\r\n  <head>\r\n    <title>");
                out.print( title );
                out.write("</title>\r\n  </head>\r\n  <body bgcolor=\"white\">\r\n    <h3>");
                out.print( title );
                out.write("</h3>\r\n    ");

                    if (firstName != null || lastName != null) {
                      out.println("firstname= " + firstName + "<br>");
                      out.println("lastname= " + lastName);
                    } else {
                      out.println("pas de paramÃ¨tres");
                    }

                out.write("\r\n    <P>\r\n    <form method=\"POST\">\r\n      firstname= <input
type=\"text\" size=\"20\" name=\"firstname\">\r\n      <br>\r\n      lastname= <input type=\"text\"
size=\"20\" name=\"lastname\">\r\n      <br>\r\n      <input type=\"submit\">\r\n    </form>\r\n
</body>\r\n</html>\r\n");

        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(t);
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

Le code généré est assez complexe. Nous ne retiendrons que les points suivants :

- La méthode principale de la servlet est la suivante :

```
    public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {
```

C'est cette méthode qui est lancée au départ de la servlet. On voit qu'elle reçoit deux paramètres : la requête *request* du client et un objet *response* pour générer sa réponse au client web.

- Dans la méthode principale un objet *JspWriter out* est déclaré puis initialisé. C'est lui qui va permettre d'envoyer du code HTML au client par des instructions *out.print("codeHTML")*.

```
        JspWriter out = null;
...
            out = pageContext.getOut();
```

- Le code Java

```
<%
  // variables locales à la procédure principale
  String title="Récupération des paramètres d'un formulaire";
  String firstName = request.getParameter("firstname");
  String lastName = request.getParameter("lastname");
%>
```

a été repris intégralement dans la méthode principale **_jspService** de la servlet. Il en est de même pour tout code situé dans les balises <%… %>

- Le code HTML de la page JSP fait l'objet d'instructions *out.print("codeHTML")* ou *out.write(...)*. Par exemple

```
            out.write("</title>\r\n  </head>\r\n  <body bgcolor=\"white\">\r\n    <h3>");
```

- Dans cet exemple, il n'y a pas d'autres méthodes que la méthode principale **_jspService**.

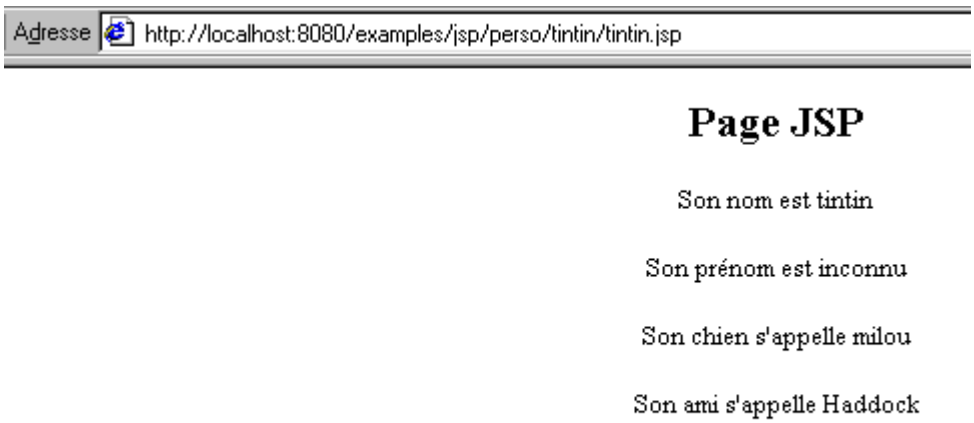
## 2.2.6 Les méthodes et variables globales d'une page JSP

Considérons la page JSP suivante :

```
<%!
  // la balise précédente démarre la partie variables et méthodes globales
  // cette partie sera reprise sans modification dans la servlet

  // une variabe globale
  String prenom="inconnu";

  // une méthode
  private String sonChien(){
    return "milou";
  }//sonChien

  // une autre méthode
  private void afficheAmi(JspWriter out) throws Exception{
    out.println("<p>Son ami s'appelle Haddock</p>");
  }//afficheAmi

  // fin de la partie globale de la servlet
%>

<%
  // la balise précédente indique que le code qui suit sera enregistré
  // dans la méthode principale de la servlet

  // variable locale à la méthode principale
  String nom="tintin";
%>


<%-- code HTML --%>
<html>
  <head>
    <title>Page JSP</title>
  </head>
  <body>
    <center>
      <h2>Page JSP</h2>
      <p>Son nom est <%= nom %></p>
      <p>Son prénom est <%= prenom %></p>
      <p>Son chien s'appelle <%= sonChien() %></p>
      <%
        // le nom de son ami
```

```
        afficheAmi(out);
      %>
    </center>
  </body>
</html>
```

Cette page JSP génère la page Web suivante :




Intéressons-nous à la façon dont sont générées les quatre lignes ci-dessus :

```
      <p>Son nom est <%= nom %></p>
      <p>Son prénom est <%= prenom %></p>
      <p>Son chien s'appelle <%= sonChien() %></p>
      <%
        // le nom de son ami
        afficheAmi(out);
      %>
```

Les lignes ci-dessus sont dans une balise `<%..%>` et feront donc partie de la méthode principale *_jspService* de la servlet qui sera générée. Comment ont-elles accès aux variables *nom*, *prenom* et méthodes *sonChien* et *afficheAmi* ?

`nom (tintin)` est une variable locale à la méthode principale de la page JSP et donc connue dans celle-ci

`prenom (inconnu)` est une variable globale de la page JSP et donc connue dans la méthode principale

`sonChien (milou)` est une méthode publique de la page JSP et donc accessible de la méthode principale

`afficheAmi (Haddock)` est une méthode publique de la page JSP et donc accessible de la méthode principale. On remarquera qu'on passe l'objet *out* en paramètre à la méthode. C'est ici obligatoire. En effet, l'objet *out* est déclaré et initialisé dans la méthode principale de la servlet et n'est pas une variable globale.

Voyons maintenant le code de la servlet java générée à partir de cette page JSP, une fois débarassé du code inutile :

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import org.apache.jasper.runtime.*;
```

```
public class tintin$jsp extends HttpJspBase {
```

```
          // la balise précédente démarre la partie variables et méthodes globales
          // cette partie sera reprise sans modification dans la servlet

          // une variabe globale
          String prenom="inconnu";

          // une méthode
          private String sonChien(){
            return "milou";
          }//sonChien
```

```
          // une autre méthode
          private void afficheAmi(JspWriter out) throws Exception{
            out.println("<p>Son ami s'appelle Haddock</p>");
          }//afficheAmi

          // fin de la partie globale de la servlet
```

```
    static {
    }
    public tintin$jsp( ) {
    }

    private static boolean _jspx_inited = false;

    public final void _jspx_init() throws org.apache.jasper.runtime.JspException {
    }
```

```
    public void _jspService(HttpServletRequest request, HttpServletResponse  response)
        throws java.io.IOException, ServletException {
```

```
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        String  _value = null;
        try {

            if (_jspx_inited == false) {
                synchronized (this) {
                    if (_jspx_inited == false) {
                        _jspx_init();
                        _jspx_inited = true;
                    }
                }
            }
            _jspxFactory = JspFactory.getDefaultFactory();
            response.setContentType("text/html;charset=ISO-8859-1");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                "", true, 8192, true);

            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();

                out.write("  \r\n\r\n");
```

```
                  // la balise précédente indique que le code qui suit sera enregistré
                  // dans la méthode principale de la servlet

                  // variable locale à la méthode principale
                  String nom="tintin";
```

```
                  out.write("\r\n\r\n\r\n");
                  out.write("\r\n<html>\r\n  <head>\r\n    <title>Page JSP</title>\r\n  </head>\r\n
<body>\r\n    <center>\r\n      <h2>Page JSP</h2>\r\n      <p>Son nom est ");
                  out.print( nom );
                  out.write("</p>\r\n      <p>Son prÃ©nom est ");
                  out.print( prenom );
                  out.write("</p>\r\n      <p>Son chien s'appelle ");
                  out.print( sonChien() );
                  out.write("</p>\r\n      ");
```

```
                        // le nom de son ami
                        afficheAmi(out);
```

```
                  out.write("\r\n    </center>\r\n  </body>\r\n</html>\r\n");
```

```
        } catch (Throwable t) {
            if (out != null && out.getBufferSize() != 0)
                out.clearBuffer();
            if (pageContext != null) pageContext.handlePageException(t);
        } finally {
            if (_jspxFactory != null) _jspxFactory.releasePageContext(pageContext);
        }
    }
}
```

On voit ci-dessus que le code Java qui était situé entre les balise JSP <%! .. %> a été repris intégralement et ne fait pas partie de la méthode principale **_jspService** de la servlet. Les variables déclarées dans cette partie sont alors des variables d'instance donc globales aux méthodes et c'est également là qu'on peut définir des méthodes autres que *_jspService*.

```
// cette partie sera reprise sans modification dans la servlet

// une variabe globale
String prenom="inconnu";

// une méthode
private String sonChien(){
  return "milou";
}//sonChien

// une autre méthode
private void afficheAmi(JspWriter out) throws Exception{
  out.println("<p>Son ami s'appelle Haddock</p>");
}//afficheAmi

// fin de la partie globale de la servlet
```

## 2.2.7 Déploiement et débogage des pages JSP au sein du serveur Tomcat

Lorsqu'on veut construire une page JSP et l'utiliser avec le serveur Tomcat, se pose la question de l'endroit où placer la page dans l'arborescence du serveur. Il y a différentes façons de faire sur lesquelles nous reviendrons. Pour l'instant, la plus simple est de placer la page JSP dans un dossier de l'arborescence *<tomcat>\webapps\examples\jsp* (Tomcat 4.x) où *<tomcat>* est le répertoire d'installation de Tomcat. Ainsi l'URL de l'exemple précédent était *http://localhost:8080/examples/jsp/perso/tintin/tintin.jsp*. Cela signifie que la page *tintin.jsp* était dans le dossier *<tomcat>\webapps\examples\jsp\perso\tintin*.

Une page JSP est traduite en fichier source java, fichier ensuite compilé par Tomcat lorsque l'URL de la page JSP est demandée par un navigateur. Des erreurs de compilation peuvent apparaître. Tomcat 4.x les signale dans sa réponse au navigateur. Il indique notamment les lignes du fichier .java qui sont erronées. Les erreurs peuvent avoir diverses sources :

1. le code JSP de la page est erroné (erreurs dans les balises jsp utilisées par exemple)
2. le code Java inclus dans la page JSP est erroné

La première cause peut être éliminée en vérifiant le code JSP de la page. La seconde peut l'être en vérifiant le code Java. On pourra le faire en compilant directement le fichier .java généré pour la page JSP avec un outil tel que JBuilder qui offre des possibilités de débogage plus évoluées que celles de Tomcat.

## 2.2.8 Exemples

Nous reprenons l'exemple déjà traité avec une servlet où un utilisateur choisit un nombre dans une liste et le serveur lui dit quel nombre il a choisi tout en lui renvoyant la même liste avec comme élément sélectionné l'élément choisi par l'utilisateur :


Pour construire cette page, on a récupéré le code de la servlet qu'on a modifié de la façon suivante :

- on a conservé tel quel le code Java qui ne produisait pas du code HTML
- le code Java qui produisait du code HTML a été transformé en un mix code HTML, code JSP

On obtient alors la page JSP suivante :

```
<%@ page import="java.sql.*, java.util.*" %>
<%!
    // variables globales de l'application
    // le titre de la page
    private final String title="Génération d'un formulaire";
    // la base de données des valeurs de liste
    private final String DSNValeurs="odbc-valeurs";
    private final String admDbValeurs="admDbValeurs";
    private final String mdpDbValeurs="mdpDbValeurs";
    // valeurs de liste
    private String[] valeurs=null;
    // msg d'erreur
    private String msgErreur=null;

    // initialisation de la page JSP - n'est exécutée qu'une seule fois
    public void jspInit(){
      // remplit le tableau des valeurs à partir d'une base de données ODBC
      // de nom DSN : DSNvaleurs
      Connection connexion=null;
      Statement st=null;
      ResultSet rs=null;
      try{
        // connexion à la base ODBC
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        connexion=DriverManager.getConnection("jdbc:odbc:"+DSNValeurs,admDbValeurs,mdpDbValeurs);
        // objet Statement
        st=connexion.createStatement();
        // exécution requête select pour récupérer les valeurs
        rs=st.executeQuery("select valeur from Tvaleurs");
        // les valeurs sont récupérées et mises dans un tableau dynamique
        ArrayList lstValeurs=new ArrayList();
        while(rs.next()){
          // on enregistre la valeur dans la liste
          lstValeurs.add(rs.getString("valeur"));
        }//while
        // transformation liste --> tableau
        valeurs=new String[lstValeurs.size()];
        for (int i=0;i<lstValeurs.size();i++){
          valeurs[i]=(String)lstValeurs.get(i);
        }
      }catch(Exception ex){
        // problème
        msgErreur=ex.getMessage();
      }finally{
        try{rs.close();}catch(Exception ex){}
        try{st.close();}catch(Exception ex){}
        try{connexion.close();}catch(Exception ex){}
      }//try
    }//init
%>
<%
   // code de _jspService exécuté à chaque requête cliente
  // y-at-il eu une erreur lors de l'initialisation de la page JSP ?
  if(msgErreur!=null){
%>
   <!-- code HTML -->
    <html>
      <head>
        <title>Erreur</title>
       </head>
       <body>
        <h3>Application indisponible (<%= msgErreur %></h3>
       </body>
    </html>
<%
    // fin de jspService
    return;
  }//if

  // on récupère l'éventuel choix de l'utilisateur
```

```
  String choix=request.getParameter("cmbValeurs");
  if(choix==null) choix="";
%>

  <%-- pas d'erreur - code HTML de la page normale --%>
    <html>
      <head>
        <title><%= title %></title>
       </head>
       <body>
        <h3>Choisissez une valeur</h3>
        <form method="POST">
          <select name="cmbValeurs">
            <%
              // affichage dynamique des valeurs
              String selected="";
              for (int i=0;i<valeurs.length;i++){
                if(valeurs[i].equals(choix)) selected="selected"; else selected="";
                out.println("<option "+selected+">"+valeurs[i]+"</option>");
              }//for
             %>
           </select>
           <input type="submit" value="Envoyer">
         </form>
<%
         // y-avait-il une valeur choisie ?
        if(! choix.equals("")){
         // on affiche le choix de l'utilisateur
%>
        <hr>Vous avez choisi le nombre<h2><%= choix %></h2>
<%
        }//if
%>
       </body>
    </html>
```

Notons les points suivants :

- les instructions *import* de la servlet ont fait l'objet d'une directive <% page import="..." %>
- la balise **<%! ... %>** encadrent les variables globales et les méthodes java de l'aplication
- la méthode *init* de la servlet qui est exécutée une seule fois, au moment du chargement de la servlet, s'appelle pour une page JSP : jspInit. Ces deux méthodes ont le même rôle. On a donc repris ici intégralement le code de la méthode *init* de la servlet.
- les variables d'instance de la servlet, celles qui doivent être connues dans plusieurs méthodes ont été reprises à l'identique. Il s'agit essentiellement des variables **title**, **valeurs** et **msgErreur** qui sont ensuite utilisées dans le code JSP.
- les balises <% ... %> encadrent du code Java qui sera inclus dans la méthode **_jspService** exécutée au moment d'une requête d'un client.
- comme pour la servlet, la méthode *_jspService* commencera par vérifier la valeur de la variable *msgErreur* pour savoir si elle doit générer une page d'erreur. S'il y a erreur, elle génère la page d'erreur et s'arrête (*return*).
- s'il n'y a pas d'erreur, elle génère le formulaire avec la liste de valeurs
- ceci fait, elle vérifie si l'utilisateur avait choisi un nombre, auquel cas elle affiche ce nombre dans la page générée

Qu'a-t-on gagné vis à vis de la servlet ? Sans doute une meilleure vue du code HTML généré. Mais il reste encore beaucoup de code Java qui "pollue" cette vue. Nous verrons ultérieurement, une autre méthode appelée délégation où on pourra mettre l'essentiel du code Java dans une servlet, la page JSP ne conservant elle que le code HTML et JSP. On sépare alors nettement la partie traitement de la partie présentation.

## 2.3 Déploiement d'une application web au sein du serveur Tomcat

Nous présentons maintenant la façon de déployer des applications web Java avec le serveur Tomcat. Si ce qui va être dit est propre à ce serveur, le déploiement d'une application web Java au sein d'un autre conteneur J2EE présentera des caractéristiques proches de celles qui vont être décrites maintenant.

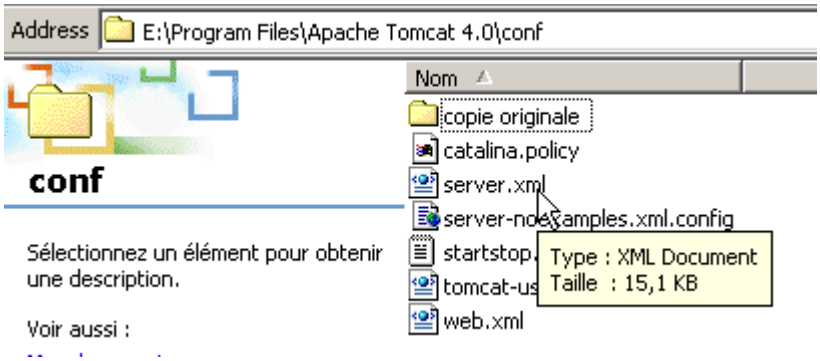
### 2.3.1 Les fichiers de configuration server.xml et web.xml

Jusqu'à maintenant, pour tester nos servlets et pages JSP, nous avons placé

- les servlets dans le dossier *<tomcat>\webapps\examples\WEB-INF\classes*. Elles étaient alors accessibles via l'URL *http://localhost:8080/examples/servlet/nomServlet*

- les pages JSP dans l'arborescence *<tomcat>\webapps\examples\jsp*. Elles étaient alors accessibles via l'URL *http://localhost:8080/examples/jsp/nomPageJSP*

Nous n'avons jamais expliqué pourquoi c'était ainsi. La configuration du serveur Tomcat est faite dans un fichier texte appelé *server.xml* qui se trouve dans le dossier *<tomcat>\conf* :



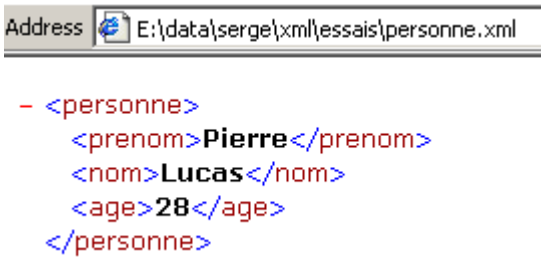

Ce fichier texte est en fait un fichier XML (eXtended Markup Language). Un document XML est un document texte contenant des balises comme l'est un document HTML. Cependant alors que les balises du langage HTML sont bien définies, celles du langage XML ne le sont pas. Ainsi le document suivant est un document XML :

```
<personne>
  <prenom>Pierre</prenom>
  <nom>Lucas</nom>
  <age>28</age>
</personne>
```

Un document XML est simplement un document "balisé" et qui suit certaines règles de balisage :

- un texte balisé à la forme **<xx att1="val1" att2="val2" ....>texte</xx>**
- une balise peut être seule et avoir la forme **<xx att1="val1" att2="val2" ..../>**

Les champs **atti** sont appelés attributs de la balise **xx** et les champs **vali** sont les valeurs associées à ces attributs. Certains documents HTML ne sont pas des documents XML valides. Par exemple la balise HTML <br> n'est pas une balise XML valide. Elle devrait s'écrire <br/> pour en être une afin de respecter la règle qui veut que toute balise XML doit être fermée. Une variante de HTML appelée XHTML a été créée afin de faire de tout document XHTML un document XML valide. Certains des navigateurs récents sont capables d'afficher des fichiers XML. Ainsi si nous appelons *personne.xml* le document XML présenté en exemple ci-dessus et que nous le visualisons avec IE6, nous obtenons l'affichage suivant :




IE6 reconnaît les balises et les colore. Il reconnaît également la structure du document grâce aux balises. Ainsi si nous appelons *personne2.xml* le document suivant :

```
<personne><prenom>Pierre</prenom><nom>Lucas</nom><age>28</age></personne>
```

et le visualisons avec IE6, nous obtenons le même affichage :

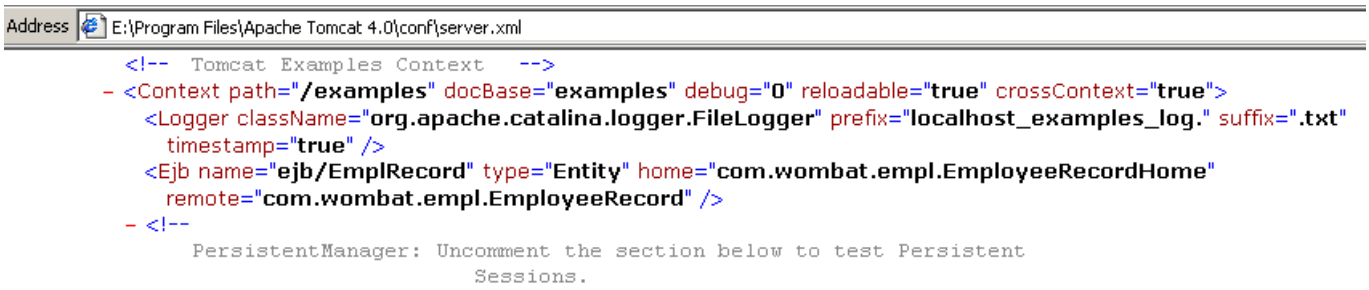

```
- <personne>
    <prenom>Pierre</prenom>
    <nom>Lucas</nom>
    <age>28</age>
  </personne>
```

IE6 a reconnu correctement la structure et le contenu du document. Tout l'intérêt du document XML repose dans cette propriété : il est facile de retrouver la structure et le contenu d'un document XML. Cela se fait avec un programme appelé un **parseur XML**. Les documents XML ont tendance à devenir la norme dans les échanges de documents sur le web. Prenons une machine A devant envoyer un document DOC à une machine B. Le document DOC est construit à partir des informations contenues dans une base de données DB-A. La machine B elle doit stocker le document DOC dans une base de données DB-B. L'échange pourra se faire de la façon suivante :

- la machine A récupère les données dans la base DB-A et encapsule celles-ci dans un document texte XML
- le document XML est envoyé à la machine B à travers le réseau
- la machine B analyse le document reçu avec un parseur XML et en récupère et la structure et les données (comme l'a fait IE6 dans notre exemple). Elle peut alors stocker les données reçues dans la base DB-B

Nous n'en dirons pas plus sur le langage XML qui mérite un livre à lui tout seul.

Ici donc, Tomcat est configuré par le fichier XML *server.xml*. Si nous visualisons celui-ci avec IE6, nous obtenons un document complexe. Nous nous attarderons simplement sur les lignes suivantes :




```
    <!--  Tomcat Examples Context  -->
  - <Context path="/examples" docBase="examples" debug="0" reloadable="true" crossContext="true">
      <Logger className="org.apache.catalina.logger.FileLogger" prefix="localhost_examples_log." suffix=".txt"
        timestamp="true" />
      <Ejb name="ejb/EmplRecord" type="Entity" home="com.wombat.empl.EmployeeRecordHome"
        remote="com.wombat.empl.EmployeeRecord" />
    - <!--
          PersistentManager: Uncomment the section below to test Persistent
                             Sessions.
```

C'est la balise **<Context ...>** qui nous intéresse ici. Elle sert à définir des applications web. Deux de ses attributs sont à noter :

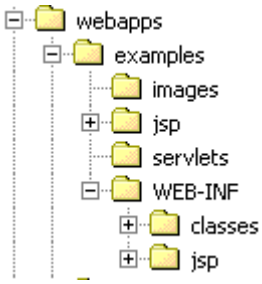
- **path** : c'est le nom de l'application web
- **docBase** : c'est le dossier dans lequel elle se trouve. Ici c'est un nom relatif : *examples*. Relatif à quel dossier ? La réponse se trouve également dans le fichier *server.xml* dans la ligne qui suit :


```
    -->
    <!--  Define the default virtual host  -->
  + <Host name="localhost" debug="0" appBase="webapps" unpackWARs="true">
```

La ligne ci-dessus définit le serveur web :

- **name** : nom du serveur web
- **appBase** : racine de l'arborescence des documents qu'il distribue. De nouveau, on a un nom relatif : *webapps*. Il est relatif au répertoire d'installation du serveur Tomcat <*tomcat*>. Ainsi donc, il s'agit du dossier <*tomcat*>*webapps*.

L'application web **examples** a ses documents dans le dossier *examples* (cf *docBase* plus haut). Ce nom est relatif à la racine de l'arborescence web du serveur, c.a.d. <*tomcat*>*webapps*. Il s'agit donc du dossier <*tomcat*>*webapps**examples*. Allons-voir de plus près ce dossier :




Nous y retrouvons le dossier *WEB-INF\classes* dans lequel nous avons rangé nos servlets pour les tester. Le dossier WEB-INF contient un fichier appelé *web.xml* :


Ce fichier sert à configurer l'application web *examples*. Nous n'entrerons pas dans les détails de ce fichier trop complexe pour le moment. Nous nous attarderons simplement sur les quelques lignes suivantes :

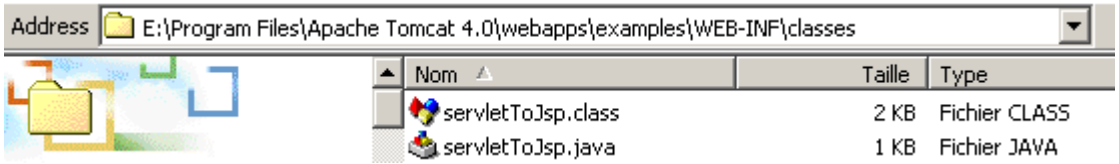
```
<servlet>
  <servlet-name>
      servletToJsp
  </servlet-name>
  <servlet-class>
      servletToJsp
  </servlet-class>
</servlet>
```

La balise <*servlet*> sert à définir une servlet au sein d'une application web. Rappelons ici que l'application web en question est *examples*. La balise *servlet* contient ici deux autres balises :

- <*servlet-name*>***servletToJsp***</*servlet-name*> : définit le nom de la servlet
- <*servlet-name*>***servletToJsp***</*servlet-name*> : définit le nom de la classe à exécuter lorsque la servlet est demandée. Dans cet exemple, la servlet et sa classe portent le même nom. Ce n'est pas obligatoire.

Comment la servlet *servletToJsp* est-elle demandée par un navigateur au serveur Tomcat ?

- le navigateur demande l'URL *http://localhost:8080/examples/servlet/servletToJsp*
- Tomcat analyse le chemin de la servlet */examples/servlet/servletToJsp*. Il interprète la première partie du chemin */examples* comme le nom d'une application web et cherche dans son fichier de configuration *server.xml* où les documents de cette application ont été rangés. Nous l'avons vu précédemment, c'est dans le dossier <*tomcat*>*webapps**examples*.
- Tomcat utilise le reste du chemin de la servlet pour localiser celle-ci dans l'application web *examples*. Ce chemin */servlet/servletToJsp* indique qu'il doit exécuter la servlet portant le nom *servletToJsp*. Tomcat va alors lire le fichier *web.xml* de configuration de l'application *examples* qu'il va trouver dans <*tomcat*>*webapps**examples**WEB-INF*. Il va trouver dans ce fichier que la servlet *servletToJsp* est liée à la classe Java *servletToJsp* (cf fichier *web.xml* ci-dessus). Il va alors chercher cette classe dans le dossier *WEB-INF\classes* de l'application web *examples*, c.a.d. dans <*tomcat*>*webapps**examples**WEB-INF**classes* et l'exécutera.



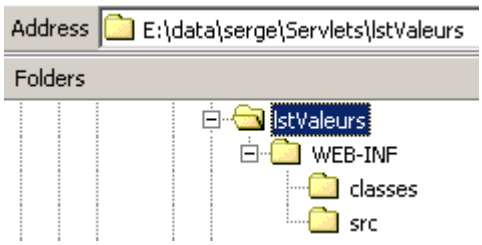

## 2.3.2 Exemple : déploiement de l'application web liste

Nous reprenons une servlet déjà étudiée et qui présentait à l'utilisateur une liste de nombres parmi lesquel il en choisissait un. La servlet lui confirmait ensuite le nombre qu'il avait choisi :

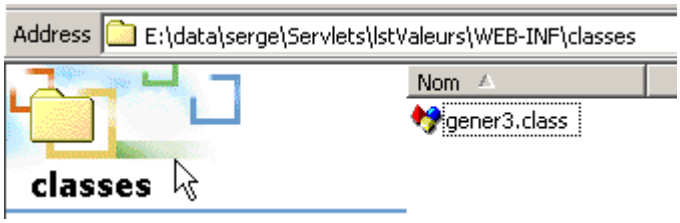

Comme le montre le champ *Address* du navigateur ci-dessus, le fichier classe de la servlet s'appelait *gener3*. D'après les explications qui ont été données précédemment :

- l'URL */examples/servlet/gener3* montre qu'il s'agit d'une servlet appelée *gener3* de l'application web *examples*
- dans le fichier *web.xml* de l'aplication *examples*, on ne trouvera rien qui parle d'une servlet *gener3*. Comment Tomcat l'a-t-il alors trouvée ? Ayant parcouru la totalité du fichier *web.xml*, je ne peux répondre avec certitude... La question reste posée...

Nous choisissons de déployer la servlet **gener3.class** sous le nom **lstValeurs** dans une application web appelée **liste** située dans le dossier *E:\data\serge\Servlets\lstValeurs* :




Nous plaçons le fichier *gener3.class* dans le dossier *WEB-INF\classes* ci-dessus :




Nous configurons l'application web **liste** en ajoutant dans le fichier **server.xml** les lignes suivantes au-dessus de celles qui définissent l'application web *manager* :

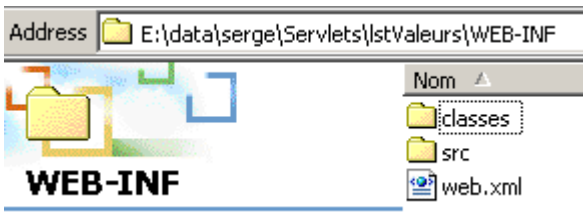
```
        <!-- Perso : lstValeurs -->
        <Context path="/liste" docBase="e:/data/serge/servlets/lstValeurs" />

        <!-- Tomcat Manager Context -->
        <Context path="/manager" docBase="manager" debug="0" privileged="true" />
        <!-- Tomcat Examples Context -->
        <Context path="/examples" docBase="examples" debug="0" reloadable="true" crossContext="true">
........
```

La ligne qui définit l'application **liste** indique qu'elle se trouve dans le dossier **e:/data/serge/servlets/lstValeurs**. Nous devons maintenant définir le fichier **web.xml** de cette application. Ce fichier définira l'unique servlet de l'application :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
```

```
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>lstValeurs</servlet-name>
    <servlet-class>gener3</servlet-class>
  </servlet>
</web-app>
```

Le fichier ci-dessus indique que la servlet nommée **lstValeurs** est associée au fichier de classe **gener3.class**. Ce fichier *web.xml* doit être créé et sauvegardé dans le dossier WEB-INF de l'application **liste** :



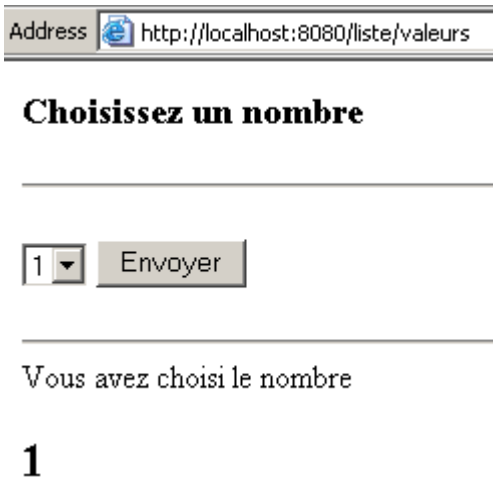

La copie d'écran ci-dessus montre un dossier *src* dans lequel on a placé le fichier source *gener3.java*. Ce dossier pourrait ne pas exister. Il ne sert à rien dans la démonstration présente. Nous sommes prêts à faire les tests :

- arrêtez et lancez Tomcat afin qu'il relise son fichier de configuration *server.xml*. Ici nous sommes sous Windows. Sous Unix, on peut forcer Tomcat à relire son fichier de configuration sans pour autant l'arrêter.
- avec un navigateur, demandez l'URL *http://localhost:8080/liste/servlet/lstValeurs*


On voit que l'URL précédente contient le mot clé *servlet* comme toutes les URL de servlets utilisées jusqu'à maintenant. On peut s'en passer en associant dans le fichier **web.xml** de l'application **liste** la servlet *lstValeurs* à un modèle d'URL (*url-pattern*) :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>lstValeurs</servlet-name>
    <servlet-class>gener3</servlet-class>
  </servlet>
   <servlet-mapping>
    <servlet-name>lstValeurs</servlet-name>
     <url-pattern>/valeurs</url-pattern>
   </servlet-mapping>
</web-app>
```

Dans la balise **<servlet-mapping>**, nous associons le chemin **/valeurs** à la servlet **lstValeurs** définie dans les lignes qui précèdent. Nous sauvegardons le nouveau fichier *web.xml* et demandons l'URL *http://localhost:8080/liste/valeurs* :




## 2.3.3 Déploiement des pages publiques d'une application web

Nous venons de voir le déploiement d'une application web formée d'une unique servlet. Une application web peut avoir de nombreuses composantes : des servlets, des pages JSP, des fichiers HTML, des applets Java, ... Où place-t-on ces éléments de l'application ? Si *<application>* est le dossier de l'application web défini par l'attribut *docBase* de l'application dans le fichier *server.xml* de configuration de Tomcat, nous avons vu que les servlets étaient placées dans *<application>\WEB-INF\classes*. Les autres éléments de l'application peuvent être placés n'importe où dans l'arborescence du dossier *<application>* sauf dans le dossier WEB-INF. Considérons l'application JSP *listvaleurs.jsp* déjà étudiée :


Cette page JSP avait été stockée dans le dossier *<tomcat>\webapps\ examples\jsp\perso\listvaleurs*. Cette page pourrait être une composante de l'application **liste** déployée précédemment. Plaçons le fichier *listvaleurs.jsp* directement dans le dossier de cette application :


Rappelons la configuration de l'application *liste* dans le fichier *server.xml* :

```
        <Context path="/liste" docBase="e:/data/serge/servlets/lstValeurs" />
```

Toute URL commençant par le chemin */liste* est considérée comme faisant partie de l'application *liste* et sera cherchée dans le dossier indiqué. Demandons l'URL *http://localhost:8080/liste/listvaleurs.jsp* avec un navigateur :


Nous avons bien obtenu la page JSP attendue.

## 2.3.4 Paramètres d'initialisation d'une servlet

Nous avons vu qu'une servlet était configurée par le fichier *<application>\WEB-INF\web.xml* où *<application>* est le dossier de l'application web à laquelle elle appartient. Il est possible d'inclure dans ce fichier des paramètres d'initialisation de la servlet. Revenons à notre servlet *lstValeurs* de l'application web *liste* et dont le fichier de configuration était le suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>lstValeurs</servlet-name>
    <servlet-class>gener3</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>lstValeurs</servlet-name>
    <url-pattern>/valeurs</url-pattern>
  </servlet-mapping>
</web-app>
```

La classe associée à la servlet est la classe *gener3*. On trouve dans le code source de celle-ci la définition de quelques constantes :

```
public class gener3 extends HttpServlet{
    // le titre de la page
    private final String title="Génération d'un formulaire";
    // la base de données des valeurs de liste
    private final String DSNValeurs="odbc-valeurs";
    private final String admDbValeurs="admDbValeurs";
    private final String mdpDbValeurs="mdpDbValeurs";
```

Rappelons la signification des quatre constantes définies ci-dessus :

| | |
|---|---|
| `title` | titre du document HTML généré par la servlet |
| `DSNValeurs` | nom DSN de la base ODBC où la servlet va chercher des données |
| `admDbValeurs` | nom d'un utilisateur ayant un droit de lecture sur la base précédente |
| `mdpDbvaleurs` | son mot de passe |

Si l'administrateur de la base *DSNValeurs* change le mot de passe de l'utilisateur *admDbValeurs*, le source de la servlet doit être modifié et recompilé. Ce n'est pas très pratique. Le fichier *web.xml* de configuration de la servlet nous offre une alternative en permettant la définition de paramètres d'initialisation de la servlet avec la balise *<init-param>* :

```
  <init-param>
      <param-name>...</param-name>
      <param-value>...</param-value>
  </init-param>
```

| | |
|---|---|
| `<param-name>` | permet de définir le nom du paramètre |
| `<param-value>` | définit la valeur associée au paramètre précédent |

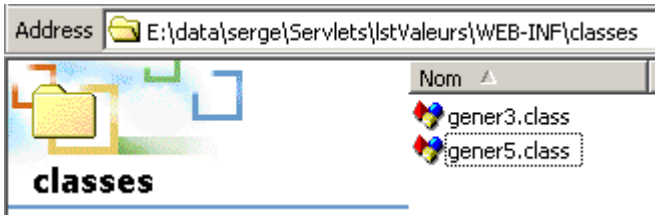
La servlet a accès à ses paramètres d'initialisation grâce aux méthodes suivantes :

| | |
|---|---|
| `[Servlet].getServletConfig()` | méthode de la classe *Servlet* dont dérive la classe *HttpServlet* utilisée pour la programmation web. Rend un objet *ServletConfig* donnant accès aux paramètres de configuration de la servlet. |
| `[ServletConfig].getInitParameter("paramètre")` | méthode de la classe *ServletConfig* qui donne la valeur du paramètre d'initialisation "paramètre" |

Nous configurons l'application *liste* avec le nouveau fichier *web.xml* suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>lstValeurs</servlet-name>
    <servlet-class>gener3</servlet-class>
  </servlet>
    <servlet>
    <servlet-name>lstValeurs2</servlet-name>
     <servlet-class>gener5</servlet-class>
     <init-param>
       <param-name>title</param-name>
        <param-value>Génération d'un formulaire</param-value>
     </init-param>
     <init-param>
       <param-name>DSNValeurs</param-name>
        <param-value>odbc-valeurs</param-value>
     </init-param>
     <init-param>
       <param-name>admDbValeurs</param-name>
        <param-value>admDbValeurs</param-value>
     </init-param>
     <init-param>
       <param-name>mdpDbValeurs</param-name>
        <param-value>mdpDbValeurs</param-value>
     </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>lstValeurs</servlet-name>
    <url-pattern>/valeurs</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>lstValeurs2</servlet-name>
    <url-pattern>/valeurs2</url-pattern>
  </servlet-mapping>
</web-app>
```

Dans l'application *liste*, nous définissons une seconde servlet appelée *lstValeurs2* liée au fichier de classe *gener5*. Ce dernier a été placé dans *<application>\WEB-INF\classes* :



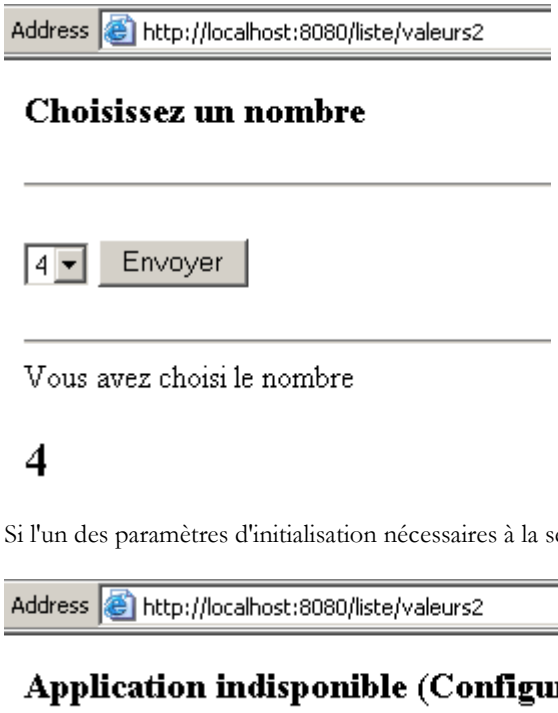

La servlet *lstValeurs2* a quatre paramètres d'initialisation : *title*, *DSNValeurs*, *admDbValeurs*, *mdpDbValeurs*. Par ailleurs, l'alias */valeurs2* a été défini pour la servlet à l'aide de la balise *<servlet-mapping>*. Aussi, la servlet *lstValeurs2* de l'application *liste* sera-t-elle accessible via l'URL *http://localhost:8080/liste/valeurs2*.

Le code source de la servlet a été modifié de la façon suivante pour récupérer les paramètres d'initialisation de la servlet :

```
public class gener5 extends HttpServlet{
    // le titre de la page
    private String title=null;
```

```
   // la base de données des valeurs de liste
   private String DSNValeurs=null;
   private String admDbValeurs=null;
   private String mdpDbValeurs=null;
...............
   // initialisation de la servlet
   public void init(){
     // on récupère les paramètres d'initialisation de la servlet
     ServletConfig config=getServletConfig();
     title=config.getInitParameter("title");
     DSNValeurs=config.getInitParameter("DSNValeurs");
     admDbValeurs=config.getInitParameter("admDbValeurs");
     mdpDbValeurs=config.getInitParameter("mdpDbValeurs");

     //a-t-on récupéré tous les paramètres ?
     if(title==null || DSNValeurs==null || admDbValeurs==null
        || mdpDbValeurs==null){
       msgErreur="Configuration incorrecte";
       return;
     }

     // on remplit le tableau des valeurs à partir d'une base de données ODBC
     // de nom DSN : DSNvaleurs
...............
```

Pour tester la servlet, il faut relancer Tomcat afin qu'il prenne en compte le nouveau fichier de configuration *web.xml* de l'application *liste*. Avec un navigateur, on demande l'URL de la servlet *http://localhost:8080/liste/valeurs2* :




Si l'un des paramètres d'initialisation nécessaires à la servlet est absent du fichier *web.xml*, on obtient la page suivante :




## 2.3.5 Paramètres d'initialisation d'une application web

Dans l'exemple précédent, seule la servlet *lstValeurs2* a accès aux paramètres *title*, *DSNValeurs*, *admDbValeurs*, *mdpDbValeurs*. On pourrait imaginer qu'une autre servlet de la même application *liste* ait besoin des données dans la même base de données qu'utilise la servlet *lstValeurs2*. Il faudrait alors redéfinir les paramètres *DSNValeurs*, *admDbValeurs*, *mdpDbValeurs* dans la partie configuration du fichier *web.xml* de la nouvelle servlet. Une autre solution est de définir les paramètres communs à plusieurs servlets au niveau de l'application et non plus au niveau des servlets. Le nouveau fichier *web.xml* de l'application devient le suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>
   <context-param>
    <param-name>DSNValeurs</param-name>
     <param-value>odbc-valeurs</param-value>
   </context-param>
   <context-param>
    <param-name>admDbValeurs</param-name>
     <param-value>admDbValeurs</param-value>
   </context-param>
   <context-param>
    <param-name>mdpDbValeurs</param-name>
     <param-value>mdpDbValeurs</param-value>
   </context-param>

  <servlet>
    <servlet-name>lstValeurs</servlet-name>
    <servlet-class>gener3</servlet-class>
  </servlet>
    <servlet>
    <servlet-name>lstValeurs3</servlet-name>
     <servlet-class>gener6</servlet-class>
     <init-param>
       <param-name>title</param-name>
        <param-value>Génération d'un formulaire</param-value>
     </init-param>
   </servlet>
   <servlet-mapping>
     <servlet-name>lstValeurs</servlet-name>
     <url-pattern>/valeurs</url-pattern>
   </servlet-mapping>
   <servlet-mapping>
    <servlet-name>lstValeurs3</servlet-name>
     <url-pattern>/valeurs3</url-pattern>
   </servlet-mapping>
</web-app>
```

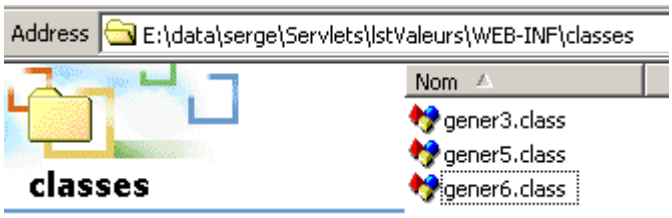
La nouvelle servlet s'appelle *lstValeurs3* et est liée au fichier de classe *gener6* et a été associée à l'alias */valeurs3* (servlet-mapping). Le paramètre *title* est le seul paramètre qui a été conservé dans la définition de la servlet. Les autres ont été placés dans la configuration de l'application dans des balises *<context-param>*. Cette balise sert à définir des informations propres à l'application et non à une servlet ou page JSP particulière. Comment la servlet Java a-t-elle accès à ces paramètres souvent appelés paramètres de contexte ? Les méthodes disponibles pour obtenir les informations de contexte sont très analogues à celles rencontrées pour obtenir les paramètres d'initialisation particuliers à une servlet :

| | |
|---|---|
| `[Servlet].getServletContext()` | méthode de la classe *Servlet* dont dérive la classe *HttpServlet* utilisée pour la programmation web. Rend un objet *ServletContext* donnant accès aux paramètres de configuration de l'application |
| `[ServletContext].getInitParameter("paramètre")` | méthode de la classe *ServletContext* qui donne la valeur du paramètre d'initialisation "paramètre" |

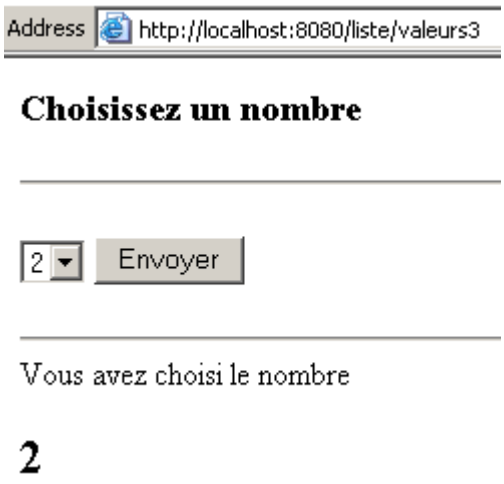
La classe *gener6.java* amène les seules modification suivantes au code Java de *gener5.java* utilisé précédemment :

```
     // on récupère les paramètres d'initialisation de la servlet
     ServletConfig config=getServletConfig();
     title=config.getInitParameter("title");

     ServletContext context=getServletContext();
     DSNValeurs=context.getInitParameter("DSNValeurs");
     admDbValeurs=context.getInitParameter("admDbValeurs");
     mdpDbValeurs=context.getInitParameter("mdpDbValeurs");

     //a-t-on récupéré tous les paramètres ?
     if(title==null || DSNValeurs==null || admDbValeurs==null
        || mdpDbValeurs==null){
       msgErreur="Configuration incorrecte";
       return;
     }

     // on remplit le tableau des valeurs à partir d'une base de données ODBC
     // de nom DSN : DSNvaleurs
.............
```

Le paramètre *title* propre à la servlet est obtenu via un objet *ServletConfig*. Les trois autres paramètres définis au niveau application sont eux obtenus via un objet *ServletContext*. Nous compilons cette classe et la mettons comme les autres dans *<application>\WEB-INF\classes* :




Nous relançons Tomcat pour qu'il prenne en compte le nouveau fichier *web.xml* de l'application et demandons l'URL *http://localhost:8080/liste/valeurs3* :




## 2.3.6 Paramètres d'initialisation d'une page JSP

Nous avons vu comment définir des paramètres d'initialisation pour une servlet ou une application web. Peut-on faire de même pour une page JSP ? Revenons sur le début du code de la page *listvaleurs.jsp* déjà étudiée :

```
<%@ page import="java.sql.*, java.util.*" %>

<%!
    // variables globales de l'application
    // le titre de la page
    private final String title="Génération d'un formulaire";
    // la base de données des valeurs de liste
    private final String DSNValeurs="odbc-valeurs";
    private final String admDbValeurs="admDbValeurs";
    private final String mdpDbValeurs="mdpDbValeurs";
.........
```

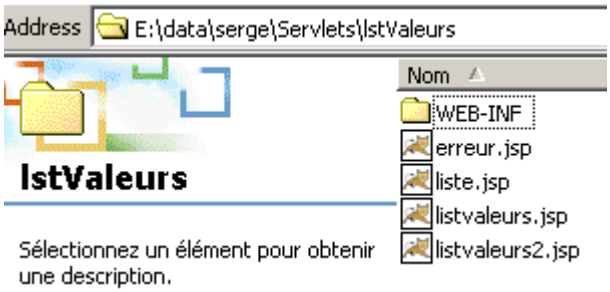
On retrouve les quatre constantes *title*, *DSNValeurs*, *admDbValeurs* et *mdpDbValeurs* définies dans le fichier *web.xml* de l'application. Les constantes *DSNValeurs*, *admDbValeurs* et *mdpDbValeurs* ont été maintenant définies au niveau application et on peut donc supposer qu'une page JSP faisant partie de cette application y aura accès. C'est le cas. Nous savons que la page JSP sera traduite en une servlet. Celle-ci aura accès au contexte via la méthode *getServletContext()*. Plus délicat est le cas de la constante *title*. En effet nous l'avons définie au niveau servlet et non au niveau application de la façon suivante :

```
  <servlet>
    <servlet-name>lstValeurs3</servlet-name>
     <servlet-class>gener6</servlet-class>
     <init-param>
       <param-name>title</param-name>
        <param-value>Génération d'un formulaire</param-value>
     </init-param>
  </servlet>
```

Pour la page JSP la syntaxe précédente ne convient plus car on n'a plus la notion de fichier de classe. La syntaxe de configuration d'une page JSP est cependant très proche de celle d'une servlet. C'est la suivante :

```
  <servlet>
    <servlet-name>JSPlstValeurs</servlet-name>
     <jsp-file>/listvaleurs2.jsp</jsp-file>
...
  </servlet>
```

En fait, une page JSP est assimilée à une servlet à laquelle on donne un nom (servlet-name). Au lieu d'associer un fichier de classe à cette servlet, on associe le fichier source de la page JSP à exécuter (jsp-file). Ainsi les quelques lignes précédentes définissent une servlet appelée *JSPlstvaleurs* associée à la page JSP */listvaleurs2.jsp*. Le chemin */listvaleurs2.jsp* est mesuré par rapport à la racine de l'application. Ainsi dans le cas de notre application *liste*, le fichier *listvaleurs2.jsp* se trouverait dans le dossier *docBase* (cf *server.xml*) de l'application *liste* :




La configuration de la page JSP sera la suivante dans le fichier *web.xml* de l'application :

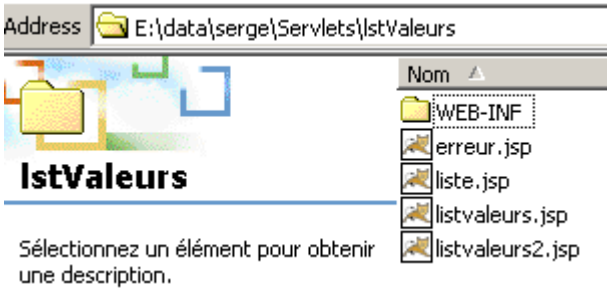
```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <context-param>
    <param-name>DSNValeurs</param-name>
    <param-value>odbc-valeurs</param-value>
  </context-param>
  <context-param>
    <param-name>admDbValeurs</param-name>
    <param-value>admDbValeurs</param-value>
  </context-param>
  <context-param>
    <param-name>mdpDbValeurs</param-name>
    <param-value>mdpDbValeurs</param-value>
  </context-param>
.......
   <servlet>
   <servlet-name>JSPlstValeurs</servlet-name>
    <jsp-file>/listvaleurs2.jsp</jsp-file>
    <init-param>
      <param-name>JSPtitle</param-name>
       <param-value>Génération d'un formulaire</param-value>
    </init-param>
  </servlet>
..........
  <servlet-mapping>
   <servlet-name>JSPlstValeurs</servlet-name>
    <url-pattern>/jspvaleurs</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
........
</web-app>
```

La page JSP *listvaleurs2.jsp* est placée dans la racine de l'application *liste* et associée au nom de servlet *JSPlstValeurs* (servlet-name) qui est lui même associé à l'alias */jspvaleurs* (servlet-mapping). Ainsi notre page JSP sera-t-elle accessible via l'URL *http://localhost:8080/liste/jspvaleurs*.

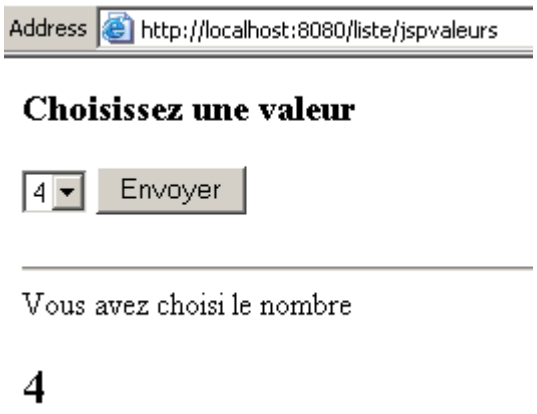
La page JSP initiale *listvaleurs.jsp* est modifiée en *listvaleurs2.jsp* et récupère ses quatre paramètres d'initialisation dans la méthode *jspInit()* :

```
<%!
    // variables globales de l'application
    // le titre de la page
    private String title=null;
    // la base de données des valeurs de liste
    private String DSNValeurs=null;
    private String admDbValeurs=null;
    private String mdpDbValeurs=null;
    // valeurs de liste
    private String[] valeurs=null;
    // msg d'erreur
    private String msgErreur=null;

    // initialisation de la page JSP - n'est exécutée qu'une seule fois
    public void jspInit(){

      // on récupère les paramètres d'initialisation de la servlet
       ServletConfig config=getServletConfig();
      title=config.getInitParameter("JSPtitle");
       ServletContext context=getServletContext();
      DSNValeurs=context.getInitParameter("DSNValeurs");
      admDbValeurs=context.getInitParameter("admDbValeurs");
      mdpDbValeurs=context.getInitParameter("mdpDbValeurs");

      //a-t-on récupéré tous les paramètres ?
      if(title==null || DSNValeurs==null || admDbValeurs==null
         || mdpDbValeurs==null){
        msgErreur="Configuration incorrecte";
        return;
      }

      // remplit le tableau des valeurs à partir d'une base de données ODBC
      // de nom DSN : DSNvaleurs
..............
```

La page JSP récupère ses paramètres d'initialisation de la même façon que les servlets. Le fichier précédent est sauvegardé dans la racine de l'application web *liste* :




Le serveur Tomcat est relancé pour le forcer à relire le nouveau fichier de configuration *web.xml* de l'application. On peut alors demander l'URL *http://localhost:8080/liste/jspvaleurs* :




## 2.3.7 Collaboration servlets/pages JSP au sein d'une application web

Lorsqu'un client fait une requête à un serveur Web, la réponse peut être élaborée par plusieurs servlets et pages JSP. Jusqu'à maintenant, la réponse avait été élaborée par une unique servlet ou page JSP. Nous avons vu que la page JSP amenait une meilleure lisibilité de la structure du document HTML généré. Cependant elle contient aussi en général beaucoup de code Java. On peut améliorer les choses en mettant

- dans une ou plusieurs servlets, le code Java qui ne génère pas le code HTML de la réponse
- dans des pages JSP, le code de génération des différents documents HTML envoyés en réponse au client

On peut ainsi espérer améliorer la séparation code Java/code HTML. Nous allons appliquer cette nouvelle structuration à notre application *liste* : une servlet Java *lstValeurs4* sera chargée de lire au démarrage les valeurs dans la base de données et ensuite d'analyser les requêtes des clients. Selon le résultat de celle-ci, la requête du client sera dirigée vers une page d'erreur *erreur.jsp* ou vers la page d'affichage de la liste de nombres *liste.jsp*. L'application *liste* sera donc formée d'une servlet et de deux pages JSP.

Comment une servlet peut-elle passer la requête qu'elle a reçue d'un client à une autre servlet ou à une page JSP ? Nous utiliserons les méthodes suivantes :

| | |
|---|---|
| `[ServletContext].getRequestDispatcher(String url)` | méthode de la classe *ServletContext* qui rend un objet *RequestDispatcher*. Le paramètre *url* est le nom de l'URL à qui on veut transmettre la requête du client. Ce passage de requête ne peut se faire qu'au sein d'une même application. Aussi le paramètre *url* est un chemin relatif à l'arborescence web de cette application. |
| `[RequestDispatcher].forward(ServletRequest request, ServletResponse response)` | méthode de l'interface *RequestDispatcher* qui transmet à l'URL précédente la requête *request* du client et l'objet *response* qui doit être utilisé pour élaborer la réponse. |
| `[ServletRequest].setAttribute(String nom, Object obj)` | lorsqu'une servlet ou page JSP passe une requête à une autre servlet ou page JSP, elle a en général besoin de passer à celle-ci d'autres informations que la seule requête du client, informations issues de son propre travail sur la requête. La méthode *setAttribute* de la classe *ServletRequest* permet d'ajouter des attributs à l'objet *request* du client sous une forme qui ressemble à un dictionnaire de couples *(attribut, valeur)* où *attribut* est le nom de l'attribut et *valeur* un objet quelconque représentant la valeur de celui-ci. |
| `[ServletRequest].getAttribute(String attribut)` | permet de récupérer les valeurs des attributs d'une requête. Cette méthode sera utilisée par la servlet ou la page JSP à qui on a relayé une requête pour obtenir les informations qui y ont été ajoutées. |

La servlet chargée de traiter le formulaire sera configurée de la façon suivante dans le fichier *web.xml* :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <context-param>
    <param-name>DSNValeurs</param-name>
    <param-value>odbc-valeurs</param-value>
  </context-param>
  <context-param>
    <param-name>admDbValeurs</param-name>
    <param-value>admDbValeurs</param-value>
  </context-param>
  <context-param>
    <param-name>mdpDbValeurs</param-name>
    <param-value>mdpDbValeurs</param-value>
  </context-param>

.............
   <servlet>
   <servlet-name>lstValeurs4</servlet-name>
    <servlet-class>gener7</servlet-class>
    <init-param>
      <param-name>title</param-name>
       <param-value>Génération d'un formulaire</param-value>
    </init-param>
    <init-param>
      <param-name>JSPerreur</param-name>
       <param-value>/erreur.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>JSPliste</param-name>
       <param-value>/liste.jsp</param-value>
    </init-param>
    <init-param>
```

```
      <param-name>URLservlet</param-name>
       <param-value>/liste/valeurs4</param-value>
     </init-param>
   </servlet>
...........
   <servlet-mapping>
    <servlet-name>lstValeurs4</servlet-name>
     <url-pattern>/valeurs4</url-pattern>
   </servlet-mapping>
.......
</web-app>
```

La servlet *lstValeurs4* aura quatre paramètres d'initialisation qui lui seront propres :

| | |
|---|---|
| `title` | le titre du document HTML à généréer |
| `JSPerreur` | l'URL de la page JSP d'erreur |
| `JSPliste` | l'URL de la page JSP présentant la liste de nombres |
| `URLservlet` | l'URL associée à l'attribut action du formulaire présenté par la page JSPliste. Cette URL sera celle de la servlet *lstValeurs4* |

La servlet aura l'alias */valeurs4* (servlet-mapping) et sera donc accessible via l'URL *http://localhost:8080/liste/valeurs4*. Elle est liée au fichier classe *gener7.java* dont le code source complet est le suivant :

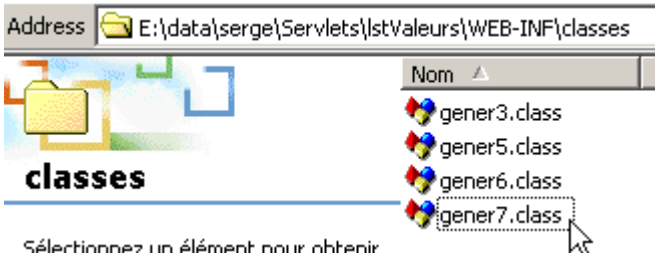
```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;
import java.util.*;

public class gener7 extends HttpServlet{
    // le titre de la page
    private String title=null;
    // la base de données des valeurs de liste
    private String DSNValeurs=null;
    private String admDbValeurs=null;
    private String mdpDbValeurs=null;
    // les pages JSP d'affichage
    private String JSPerreur=null;
    private String JSPliste=null;
    // l'URL de la servlet
    private String URLservlet=null;
    // valeurs de liste
    private String[] valeurs=null;
    // msg d'erreur
    private String msgErreur=null;

    // ------------------------------------------------------------------
    // GET
    public void doGet(HttpServletRequest request,HttpServletResponse response)
         throws IOException, ServletException{

      // on met msgErreur,title dans les attributs de la requête
      request.setAttribute("msgErreur",msgErreur);
      request.setAttribute("title",title);
      request.setAttribute("URLservlet",URLservlet);

      // y-a-t-il eu erreur au chargement de la servlet ?
      if(msgErreur!=null){
        // on passe la main à une page JSP d'erreur
        getServletContext().getRequestDispatcher(JSPerreur).forward(request,response);
        // fin
        return;
      }

      // il n'y a pas eu d'erreur
      // on met la liste des valeurs dans les attributs de la requête
      request.setAttribute("valeurs",valeurs);

      // on récupère l'éventuel choix de l'utilisateur
      String choix=request.getParameter("cmbValeurs");
      if(choix==null) choix="";
      request.setAttribute("choix",choix);

      // on passe la main à la page JSP de présentation de la liste
      getServletContext().getRequestDispatcher(JSPliste).forward(request,response);
```

```
      // fin
      return;
    }//GET

    // ------------------------------------------------------------------
    // POST
    public void doPost(HttpServletRequest request,HttpServletResponse response)
         throws IOException, ServletException{

      // on renvoie sur GET
      doGet(request,response);
    }//POST

    // ------------------------------------------------------------------
    // initialisation de la servlet
    public void init(){

      // on récupère les paramètres d'initialisation de la servlet
      ServletConfig config=getServletConfig();
      title=config.getInitParameter("title");
      JSPerreur=config.getInitParameter("JSPerreur");
      JSPliste=config.getInitParameter("JSPliste");
      URLservlet=config.getInitParameter("URLservlet");

      ServletContext context=getServletContext();
      DSNValeurs=context.getInitParameter("DSNValeurs");
      admDbValeurs=context.getInitParameter("admDbValeurs");
      mdpDbValeurs=context.getInitParameter("mdpDbValeurs");


      //a-t-on récupéré tous les paramètres ?
      if(title==null || DSNValeurs==null || admDbValeurs==null
         || mdpDbValeurs==null || JSPerreur==null || JSPliste==null || URLservlet==null){
        msgErreur="Configuration incorrecte";
        return;
      }

      // remplit le tableau des valeurs à partir d'une base de données ODBC
      // de nom DSN : DSNvaleurs
      Connection connexion=null;
      Statement st=null;
      ResultSet rs=null;
      try{
        // connexion à la base ODBC
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        connexion=DriverManager.getConnection("jdbc:odbc:"+DSNValeurs,admDbValeurs,mdpDbValeurs);
        // objet Statement
        st=connexion.createStatement();
        // exécution requête select pour récupérer les valeurs
        rs=st.executeQuery("select valeur from Tvaleurs");
        // les valeurs sont récupérées et mises dans un tableau dynamique
        ArrayList lstValeurs=new ArrayList();
        while(rs.next()){
          // on enregistre la valeur dans la liste
          lstValeurs.add(rs.getString("valeur"));
        }//while
        // transformation liste --> tableau
        valeurs=new String[lstValeurs.size()];
        for (int i=0;i<lstValeurs.size();i++){
          valeurs[i]=(String)lstValeurs.get(i);
        }
      }catch(Exception ex){
        // problème
        msgErreur=ex.getMessage();
      }finally{
        try{rs.close();}catch(Exception ex){}
        try{st.close();}catch(Exception ex){}
        try{connexion.close();}catch(Exception ex){}
      }//try
    }//init
  }//classe
```

La nouveauté dans cette classe est la transmission de la requête du client à la page *JSPerreur* en cas d'erreur et à la page *JSPliste* sinon. La classe n'élabore pas elle-même la réponse. Ce sont les pages JSP *JSPerreur* et *JSPliste* qui s'en chargent. Auparavant la servlet a ajouté des attributs (*setAttribute*) à la requête du client :

- un message d'erreur *msgErreur* en cas d'erreur pour la page JSPerreur
- les valeurs (*valeurs)* à afficher, la valeur choisie (*choix*) par l'utilisateur, le titre (*title*) du formulaire, l'URL (*URLservlet*) de l'attribut action du formulaire pour la page JSPliste

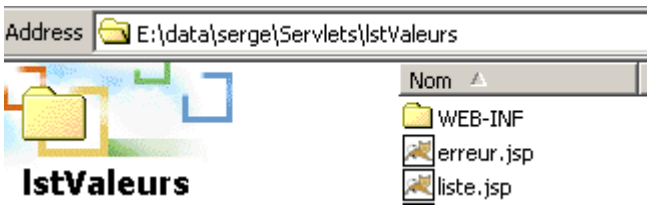
Cette classe est compilée et mise dans les classes de l'application :




La page JSP affichant un message d'erreur est configurée de la façon suivante :

```
  <servlet>
    <servlet-name>JSPerreur</servlet-name>
     <jsp-file>/erreur.jsp</jsp-file>
     <init-param>
       <param-name>mainServlet</param-name>
        <param-value>/valeurs4</param-value>
     </init-param>
  </servlet>
.........
<servlet-mapping>
    <servlet-name>JSPerreur</servlet-name>
     <url-pattern>/JSPerreur</url-pattern>
  </servlet-mapping>
```

Le fichier JSP associé à la page d'erreur s'appelle *erreur.jsp* et se trouve dans la racine de l'application :




Elle a pour alias */JSPerreur* ce qui la rend accessible via l'URL *http://localhost:8080/liste/JSPerreur*. Elle a un paramètre d'initialisation appelé *mainServlet* et dont la valeur est l'alias de la servlet principale décrite précédemment. On notera que cet alias est relatif à la racine de l'application *liste,* sinon on aurait */liste/valeurs4*. Le code de la page *erreur.jsp* est le suivant :

```
<%
  // code de _jspService
  // on récupère le paramètre d'initialisation mainServlet
  String servletListValeurs=config.getInitParameter("mainServlet");
  // on récupère l'attribut msgErreur
  String msgErreur=(String)request.getAttribute("msgErreur");
  // attribut valide ?
  if(msgErreur!=null){
%>
    <!-- code HTML -->
    <html>
      <head>
        <title>Erreur</title>
       </head>
       <body>
        <h3>Application indisponible (<%= msgErreur %>)</h3>
       </body>
    </html>
<%
  } else { // attribut msgErreur invalide - retour à la servlet principale
%>
  <jsp:forward page="<%= servletListValeurs %>" />
<%
  }
%>
```

Cette page doit normalement être appelée par la servlet précédente qui doit lui passer l'attribut *msgErreur*. Cependant rien n'empêche de l'appeler directement si on connaît son URL. Aussi si on découvre que l'attribut *msgErreur* est absent, on passe la requête à la servlet principale. Ici, on utilise une balise propre aux pages JSP et dont la syntaxe est :

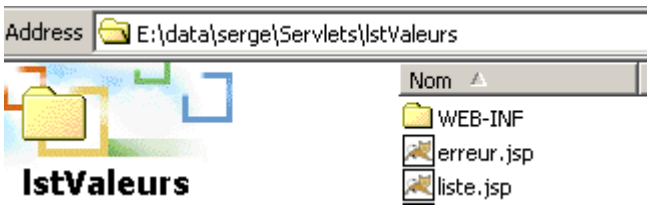
```
<jsp:forward page="URL" />
```

où URL est l'URL de la servlet à qui on transfère la requête du client. Si l'attribut *msgErreur* est présent, la page d'erreur est affichée.

La page JSP affichant la liste des nombres est configurée de la façon suivante :

```
  <servlet>
    <servlet-name>JSPliste</servlet-name>
     <jsp-file>/liste.jsp</jsp-file>
     <init-param>
       <param-name>mainServlet</param-name>
        <param-value>/valeurs4</param-value>
     </init-param>
.........
  <servlet-mapping>
    <servlet-name>JSPliste</servlet-name>
     <url-pattern>/JSPliste</url-pattern>
  </servlet-mapping>
```

Le fichier JSP associé à la page d'erreur s'appelle *liste.jsp* et se trouve dans la racine de l'application :




La servlet a pour alias */JSPliste* ce qui la rend accessible via l'URL *http://localhost:8080/liste/JSPliste*. Elle a un paramètre d'initialisation appelé *mainServlet* et dont la valeur est l'alias de la servlet principale. Le code de la page *liste.jsp* est le suivant :

```
  <%-- page d'affichage de la liste des valeurs --%>
  <%
    // code de jspService
    // on récupère le paramètre d'initialisation
    String servletListValeurs=config.getInitParameter("mainServlet");

    // on récupère les attributs de la requête venant de la servlet principale
    String title=(String) request.getAttribute("title");
    String[] valeurs=(String[]) request.getAttribute("valeurs");
    String choix=(String) request.getAttribute("choix");
    String URLservlet=(String) request.getAttribute("URLservlet");

    // attributs valides ?
    if(title==null || valeurs==null || choix==null){
      // il y a un attribut invalide - on passe la main à la servlet
   %>
    <jsp:forward page="<%= servletListValeurs %>" />
  <%
    }//if
  %>

  <%-- code HTML --%>
    <html>
      <head>
        <title><%= title %></title>
      </head>
      <body>
        <h3>Choisissez une valeur</h3>
        <form method="POST" action="<%= URLservlet %>">
          <select name="cmbValeurs">
            <%
              // affichage dynamique des valeurs
              String selected="";
              for (int i=0;i<valeurs.length;i++){
                if(valeurs[i].equals(choix)) selected="selected"; else selected="";
                out.println("<option "+selected+">"+valeurs[i]+"</option>");
              }//for
            %>
          </select>
          <input type="submit" value="Envoyer">
        </form>
        <%
          // y-avait-il une valeur choisie ?
          if(! choix.equals("")){
            // on affiche le choix de l'utilisateur
        %>
        <hr>Vous avez choisi le nombre<h2><%= choix %></h2>
        <%
          }//if
        %>
      </body>
    </html>
```

Cette page procède comme la page *erreur.jsp*. Elle doit normalement être appelée par la servlet */liste/valeurs4* et recevoir les attributs *title*, *valeurs* et *choix*. Si l'un de ces paramètres est manquant, on passe la main à la servlet *URLservlet* (*/liste/valeurs4*). Si les paramètres sont tous présents, la liste des nombres est affichée ainsi que le nombre choisi par l'utilisateur s'il en avait choisi un.

Si on demande l'URL de la servlet principale, on obtient le résultat suivant :


avec le code source suivant (*View/Source*) :

```
<html>
  <head>
    <title>Génération d'un formulaire</title>
  </head>
  <body>
    <h3>Choisissez une valeur</h3>
    <form method="POST" action="/liste/valeurs4">
      <select name="cmbValeurs">
        <option >0</option>
         <option >1</option>
         <option >2</option>
         <option >3</option>
         <option >4</option>
         <option >6</option>
         <option >5</option>
         <option >7</option>
         <option >8</option>
         <option >9</option>
       </select>
       <input type="submit" value="Envoyer">
    </form>

  </body>
</html>
```

Ce document HTMl a été généré par la page JSP *liste.jsp*. On voit que les attributs *title*, *valeurs*, *URLservlet* ont bien été récupérés.

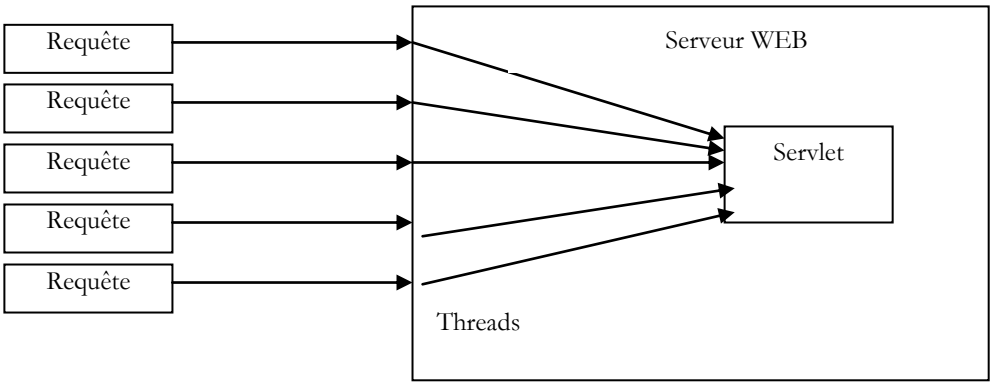
Pour conclure sur la collaboration servlets/pages JSP, nous observons que les pages JSP sont ici très courtes et dépourvues du code Java ne concourant pas à créer directement la réponse HTML. La structure des documents générés est ainsi plus visible.

## 2.4 Cycle de vie des servlets et pages JSP

### 2.4.1 Le cycle de vie

Nous nous intéressons ici au cycle de vie des servlets. Celui des pages JSP en découle. Considérons une servlet appelée pour la première fois. Une instance de classe est alors créée par le serveur Web et chargée en mémoire. Elle va alors servir la requête. Ceci fait, la servlet n'est pas déchargée de la mémoire. Il y reste afin de servir d'autres requêtes afin d'optimiser les temps de réponse du serveur. Elle sera déchargé lorsqu'un temps suffisamment long sera passé sans qu'elle ait servi de nouvelles requêtes. Ce temps est en général configurable au sein du serveur web.

Lorsqu'elle est en mémoire, la servlet peut servir plusieurs requêtes simultanément. Le serveur Web crée un thread par requête qui tous utilisent la même instance de servlet :




Tous les threads ci-dessus partagent les variables de l'instance de servlet. Il peut y avoir besoin de synchroniser les threads pour éviter la corruption des données de la servlet. Nous allons y revenir.

Au chargement d'une servlet, une méthode particulière de la servlet est exécutée :

```
public void init() throws ServletException{
}
```

Pour une page JSP, c'est la méthode

```
  public void jspInit(){
  }
```

qui est exécutée. Voici un exemple de page JSP utilisant la méthode **jspInit** :

```
<html>
  <head>
    <title>Compteur synchronisé</title>
  </head>
  <body>
    Compteur= <%= getCompteur() %>
  </body>
</html>

<%!
  // variables et méthodes globales de la page JSP

  // variable d'instance
  int compteur;

  // méthode pour incrémenter le compteur
  public int getCompteur(){
    // on incrémente le compteur
    int myCompteur=compteur;
    myCompteur++;
    compteur=myCompteur;
    // on le rend
    return compteur;
  }

  // la méthode exécutée au chargement initial de la page
  public void jspInit(){
    // init compteur
    compteur=100;
  }
%>
```

La page JSP précédente initialise à 100 un compteur dans *jspInit*. Toute requête ultérieure à la servlet incrémente puis affiche la valeur de ce compteur :

La première fois :


Compteur= 101

La seconde fois :


Compteur= 102

On voit bien ci-dessus, qu'entre les deux requêtes, la servlet n'a pas été déchargée sinon on aurait eu le compteur à 101 lors de la seconde requête. Lorsque la servlet est déchargée, la méthode

```
public void destroy(){
}
```

est exécutée si elle existe. Pour les pages JSP c'est la méthode

```
  public void jspDestroy(){
  }
```

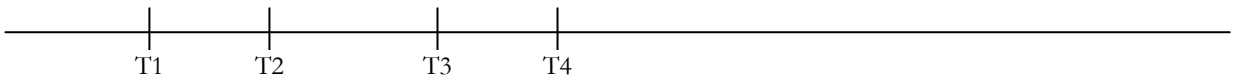
Dans ces méthodes, on pourra par exemple fermer des connexions à des bases de données, connexions qui auront été ouvertes dans les méthodes *init* correspondantes.

## 2.4.2 Synchronisation des requêtes à une servlet

Revenons à la page JSP précédente qui incrémente un compteur et renvoie celui-ci au client web. Supposons qu'il y ait 2 requêtes simultanées. Deux threads sont alors créés pour les exécuter, threads qui vont utiliser la même instance de servlet donc ici le même compteur. Rappelons le code qui incrémente le compteur :

```
  public int getCompteur(){
    // on incrémente le compteur
    int myCompteur=compteur;
    myCompteur++;
    compteur=myCompteur;
    // on le rend
    return compteur;
  }
```

L'incrémentation du compteur a été écrite volontairement de façon maladroite. Supposons que l'exécution des deux threads se passe de la façon suivante :




1) au temps T1, le thread TH1 est exécuté. Il lit le compteur (=145) dans *myCompteur* puis il est interrompu et perd le processeur. Il n'a dons pas eu le temps d'incrémenter *myCompteur* et de recopier la nouvelle valeur dans *compteur*.
2) au temps T2, le thread TH2 est exécuté. Il lit le compteur (=145) dans *myCompteur* puis il est interrompu et perd le processeur. On notera que les deux threads ont des variables *myCompteur* différentes. Ils ne partagent que les variables d'instance, celles qui sont globales aux méthodes.
3) au temps T3, le thread TH1 reprend la main et termine. Il renvoie donc 146 à son client.
4) au temps T4, le thread TH2 reprend la main et termine. Il renvoie lui aussi 146 à son client alors qu'il aurait du renvoyer 147.

On a là, un problème de synchronisation de threads. Lorsque TH1 veut incrémenter le compteur, il faudrait alors empêcher tout autre thread de le faire également. Pour mettre en évidence ce problème, nous réécrivons la page JSP de la façon suivante :

```
<html>
  <head>
    <title>Compteur synchronisé</title>
  </head>
  <body>
    Compteur= <%= getCompteur() %>
  </body>
</html>
```

```
<%!
  // variables et méthodes globales de la page JSP

  // variable d'instance
  int compteur;

  // méthode pour incrémenter le compteur
  public int getCompteur(){
    // on lit le compteur
    int myCompteur=compteur;
    // on s'arrête 10 secondes
    try{
      Thread.sleep(10000);
    }catch (Exception ignored){}
    // on incrémente le compteur
    compteur=myCompteur+1;
    // on le rend
    return compteur;
  }

  // la méthode exécutée au chargement initial de la page
  public void jspInit(){
    // init compteur
    compteur=100;
  }
%>
```

Ici, nous avons forcé le thread à s'arrêter 10 secondes après avoir lu le compteur. Il devrait donc perdre le processeur et un autre thread pouvoir lire à son tour un compteur qui n'a pas été incrémenté. Lorsque nous faisons des requêtes avec un navigateur, nous ne voyons pas de différence si ce n'est l'attente de 10 secondes avant d'avoir le résultat.


Compteur= 101

Maintenant si nous ouvrons deux fenêtres de navigateur et faisons deux requêtes suffisamment rapprochées dans le temps :


Compteur= 105


Compteur= 105

Nous obtenons la même valeur de compteur. Nous pouvons mieux mettre en évidence le problème avec un client programmé plutôt que manuel comme l'est le navigateur. Suit un client perl qui s'appelle de la façon suivante :

*programme URL N*

où

**URL** est l'URL de la servlet de comptage
**N** le nombre de requêtes à faire à cette servlet

Voici les résultats obtenus pour 5 requêtes qui montrent bien le problème de mauvaise synchronisation des threads : elles obtiennent toutes la même valeur du compteur.

```
DOS>java clientCompteurJSP http://localhost:8080/examples/jsp/perso/compteur/compteur2.jsp 5
Compteur=121
Compteur=121
Compteur=121
Compteur=121
Compteur=121
```

Le code du client Java est le suivant.

```
import java.net.*;
import java.util.regex.*;
import java.io.*;

public class clientCompteurJSP {

  public static void main(String[] params){

    // données
    String syntaxe="Syntaxe : pg URL nbAppels";

    // vérification des paramètres
    if(params.length!=2){
      System.err.println(syntaxe);
      System.exit(1);
    }//if
    // URL
    URL urlCompteur=null;
    try{
      urlCompteur=new URL(params[0]);
      String query=urlCompteur.getQuery();
      if(query!=null) throw new Exception();
    }catch (Exception ex){
      System.err.println(syntaxe);
      System.err.println("URL ["+params[0]+" incorrecte");
      System.exit(2);
    }//try-catch
    // nombre d'appels
    int nbAppels=0;
    try{
      nbAppels=Integer.parseInt(params[1]);
      if(nbAppels<=0) throw new Exception();
    }catch(Exception ex){
      System.err.println(syntaxe);
      System.err.println("Nombre d'appels ["+params[1]+" incorrect");
      System.exit(3);
    }//try-catch

    // les paramètres sont corrects - on peut faire les connexions à l'URL
    try{
      getCompteurs(urlCompteur,nbAppels);
    }catch(Exception ex){
      System.err.println(syntaxe);
      System.err.println("L'erreur suivante s'est produite : "+ex.getMessage());
      System.exit(4);
    }//try-catch
  }//main

   private static void getCompteurs (URL urlCompteur, int nbAppels)
     throws Exception {
    // fait nbAppels à l'URL urlCompteur
    // affiche à chaque fois la valeur du compteur renvoyée par le serveur web

    // on retire d'urlCompteur les infos nécessaire à la connexion au serveur d'impôts
    String path=urlCompteur.getPath();
    if(path.equals("")) path="/";
    String host=urlCompteur.getHost();
    int port=urlCompteur.getPort();
    if(port==-1) port=urlCompteur.getDefaultPort();

    // on fait les appels à l'URL
    Socket[] clients=new Socket[nbAppels];
    for(int i=0;i<nbAppels;i++){
      // on se connecte au serveur
      clients[i]=new Socket(host,port);
      // on cré un flux d'écriture vers le serveur
      PrintWriter OUT=new PrintWriter(clients[i].getOutputStream(),true);
      // on demande l'URL - envoi des entêtes HTTP
      OUT.println("GET " + path + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      OUT.println("Connection: close");
      OUT.println("");
    }//for

    // données locales
    String réponse=null;          // réponse du serveur
    // le modèle recherché dans la réponse HTML du serveur
```

```
    Pattern modèleCompteur=Pattern.compile("^\\s*Compteur= (\\d+)");
    // le modèle d'une réponse correcte
    Pattern réponseOK=Pattern.compile("^.*? 200 OK");
    // le résultat de la comparaison au modèle
    Matcher résultat=null;

    for(int i=0;i<nbAppels;i++){
      // chaque client lit la réponse que lui envoie le serveur

      // on crée les flux d'entrée-sortie du client TCP
      BufferedReader IN=new BufferedReader(new InputStreamReader(clients[i].getInputStream()));

      // on lit la 1ère ligne de la réponse
      réponse=IN.readLine();
      // on compare la ligne HTTP au modèle de la réponse correcte
      résultat=réponseOK.matcher(réponse);
      if(! résultat.find()){
        // on a un problème d'URL
        throw new Exception("Client n° " + i + " - Le serveur a répondu : URL ["+ urlCompteur + "]
inconnue");
      }//if

      // on lit la réponse jusqu'à la fin des entêtes
      while((réponse=IN.readLine())!=null && ! réponse.equals("")){
      }//while

      // c'est fini pour les entêtes HTTP - on passe au code HTML
      // pour récupérer la valeur du compteur
      boolean compteurTrouvé=false;
      while((réponse=IN.readLine())!=null){
        // on compare la ligne au modèle du compteur
        if(! compteurTrouvé){
          résultat=modèleCompteur.matcher(réponse);
          if(résultat.find()){
            // compteur trouvé
            System.out.println("Compteur="+résultat.group(1));
            compteurTrouvé=true;
          }//if
        }//if
      }//while

      // c'est fini
      clients[i].close();
    }//for
  }//getCompteurs

}//classe
```

Explicitons le code précédent :

- le programme admet deux paramètres :
  - l'URL de la page JSP du compteur
  - le nombre de clients à créer pour cette URL
- le programme commence donc par vérifier la validité des paramètres : qu'il y en a bien deux, que le premier ressemble syntaxiquement à une URL et que le second à un nombre entier >0. Pour vérifier que l'URL est syntaxiquement correcte, on utilise la classe URL et son constructeur **URL (String)** qui construit un objet URL à partir d'une chaîne de caractères telle que *http://istia.univ-angers.fr*. Une exception est lancée si la chaîne de caractères n'est pas une URL syntaxiquement valide. Cela nous permet de vérifier la validité du premier paramètre.
- une fois les paramètres vérifiés, la main est passée à la procédure *getCompteurs*. Celle-ci va créer *nbAppels* clients qui tous vont se connecter en même temps (ou quasiment) à l'URL *urlCompteur*.
- le port et la machine sur laquelle les clients doivent se connecter sont tirés de l'URL *urlCompteur* : **[URL].getHost()** permet d'avoir le nom de la machine et **[URL].getPort()** permet d'obtenir le port.
- une première boucle permet à chaque client :
  - de se connecter au serveur web
  - de lui demander l'URL *urlCompteur*

  Dans cette boucle, le client n'attend pas la réponse du serveur. En effet, on veut amener le serveur à recevoir des demandes quasi simultanées.
- une seconde boucle permet à chaque client de recevoir et traiter la réponse que lui envoie le serveur. Le traitement consiste à trouver dans la réponse la ligne qui contient la valeur du compteur et à afficher celle-ci.

Pour résoudre le problème mis en évidence précédemment (même compteur envoyé aux cinq clients) il nous faut synchroniser les threads du service de comptage sur un même objet avant d'entrer dans la section critique de lecture et mise à jour du compteur. La nouvelle page JSP est la suivante :

```
<html>
  <head>
    <title>Compteur synchronisé</title>
  </head>
  <body>
    Compteur= <%= getCompteur() %>
  </body>
</html>

<%!
  // variables et méthodes globales de la page JSP

  // variables d'instance
  int compteur;
  Object verrou=new Object();

  // méthode pour incrémenter le compteur
  public int getCompteur(){

    // on syncronise la section critique
    synchronized(verrou){
      // on lit le compteur
      int myCompteur=compteur;
      // on s'arrête 10 secondes
      try{
        Thread.sleep(10000);
      }catch (Exception ignored){}
      // on incrémente le compteur
      compteur=myCompteur+1;
    }//synchronized
    // on le rend
    return compteur;
  }//getCompteur

  // la méthode exécutée au chargement initial de la page
  public void jspInit(){
    // init compteur
    compteur=100;
  }
%>
```

A l'exécution, on obtient alors les résultats suivants :

```
dos>c:\perl\bin\perl.exe client2.pl http://localhost:8080/examples/jsp/perso/compteur/compteur3.jsp 5
    Compteur= 104
    Compteur= 106
    Compteur= 105
    Compteur= 107
    Compteur= 108
```

La documentation indique que le serveur Web peut parfois créer plusieurs instances d'une même servlet. Dans ce cas, la synchronisation précédente ne fonctionne plus car la variable *verrou* est locale à une instance et n'est donc pas connue des autres instances. Il en est de même pour la variable *compteur*. Pour les rendre globales à toutes les instances, on écrira :

```
// variable de classe
  static int compteur;
  static Object verrou=new Object();
```

Le reste du code ne change pas.

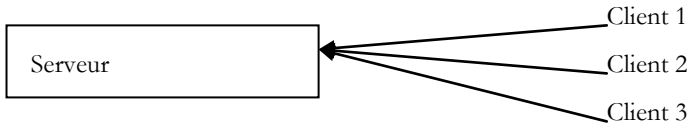
# 3. Suivi de session

## 3.1 Le problème

Une application web peut consister en plusieurs échanges de formulaires entre le serveur et le client. On a alors le fonctionnement suivant :

- **étape 1**
  1. le client C1 ouvre une connexion avec le serveur et fait sa requête initiale.
  2. le serveur envoie le formulaire F1 au client C1 et ferme la connexion ouverte en 1.
- **étape 2**
  3. le client C1 le remplit et le renvoie au serveur. Pour ce faire le navigateur ouvre une nouvelle connexion avec le serveur.
  4. celui-ci traite les données du formulaire 1, calcule des informations I1 à partir de celles-ci, envoie un formulaire F2 au client C1 et ferme la connexion ouverte en 3.
- **étape 3**
  5. le cycle des étapes 3 et 4 se répète dans des étapes 5 et 6. A l'issue de l'étape 6, le serveur aura reçu deux formulaires F1 et F2 et à partir de ceux-ci aura calculé des informations I1 et I2.

Le problème posé est : comment le serveur fait-il pour conserver les informations I1 et I2 liées au client C1 ? On appelle ce problème le suivi de la session du client C1. Pour comprendre sa racine, examinons le schéma d'une application serveur TCP-IP servant simultanément plusieurs clients :




Dans une application client-serveur TCP-IP classique :

- le client crée une connexion avec le serveur
- échange à travers celle-ci des données avec le serveur
- la connexion est fermée par l'un des deux partenaires

Les deux points importants de ce mécanisme sont :

1. une connexion unique est créée pour chacun des clients
2. cette connexion est utilisée pendant toute la durée du dialogue du serveur avec son client

Ce qui permet au serveur de savoir à un moment donné avec quel client il travaille, c'est la connexion ou dit autrement le "tuyau" qui le relie à son client. Ce tuyau étant dédié à un client donné, tout ce qui arrive de ce tuyau vient de ce client et tout ce qui est envoyé dans ce tuyau arrive au client.

Le mécanisme du client-serveur HTTP suit bien le schéma précédent avec cependant la particularité que le dialogue client-serveur est limité à un **unique échange** entre le client et le serveur :

- le client ouvre une connexion vers le serveur et fait sa demande
- le serveur fait sa réponse et ferme la connexion

Si au temps T1, un client C fait une demande au serveur, il obtient une connexion C1 qui va servir à l'échange unique demande-réponse. Si au temps T2, ce même client fait une seconde demande au serveur, il va obtenir une connexion C2 différente de la connexion C1. Pour le serveur, il n'y a alors aucune différence entre cette seconde demande de l'utilisateur C et sa demande intiale : dans les deux cas, le serveur considère le client comme un nouveau client. Pour qu'il y ait un lien entre les différentes connexions du client C au serveur, il faut que le client C se fasse "reconnaître" par le serveur comme un "habitué" et que le serveur récupère les informations qu'il a sur cet habitué.

Imaginons une administration qui fonctionnerait de la façon suivante :

- Il y a une unique file d'attente

- Il y a plusieurs guichets. Donc plusieurs clients peuvent être servis simultanément. Lorsqu'un guichet se libère, un client quitte la file d'attente pour être servi à ce guichet
- Si c'est la première fois que le client se présente, la personne au guichet lui donne un jeton avec un numéro. Le client ne peut poser qu'une question. Lorsqu'il a sa réponse il doit quitter le guichet et passer à la fin de la file d'attente. Le guichetier note les informations de ce client dans un dossier portant son numéro.
- Lorsqu'arrive à nouveau son tour, le client peut être servi par un autre guichetier que la fois précédente. Celui-ci lui demande son jeton et récupère le dossier ayant le numéro du jeton. De nouveau le client fait une demande, a une réponse et des informations sont rajoutées à son dossier.
- et ainsi de suite... Au fil du temps, le client aura la réponse à toutes ses requêtes. Le suivi entre les différentes requêtes est réalisé grâce au jeton et au dossier associé à celui-ci.

Le mécanisme de suivi de session dans une application client-serveur web est analogue au fonctionnement précédent :

- lors de sa première demande, un client se voit donner un jeton par le serveur web
- il va présenter ce jeton à chacune de ses demandes ultérieures pour s'identifier

Le jeton peut revêtir différentes formes :

- **celui d'un champ caché dans un formulaire**
  - le client fait sa première demande (le serveur le reconnaît au fait que le client n'a pas de jeton)
  - le serveur fait sa réponse (un formulaire) et met le jeton dans un champ caché de celui-ci. A ce moment, la connexion est fermée (le client quitte le guichet avec son jeton). Le serveur a pris soin éventuellement d'associer des informations à ce jeton.
  - le client fait sa seconde demande en renvoyant le formulaire. Le serveur récupère dans celui-ci le jeton. Il peut alors traiter la seconde demande du client en ayant accès, grâce au jeton, aux informations calculées lors de la première demande. De nouvelles informations sont ajoutées au dossier lié au jeton, une seconde réponse est faite au client et la connexion est fermée pour la seconde fois. Le jeton a été mis de nouveau dans le formulaire de la réponse afin que l'utilisateur puisse le présenter lors de sa demande suivante.
  - et ainsi de suite...

  L'inconvénient principal de cette technique est que le jeton doit être mis dans un formulaire. Si la réponse du serveur n'est pas un formulaire, la méthode du champ caché n'est plus utilisable.

- **celui du cookie**
  - le client fait sa première demande (le serveur le reconnaît au fait que le client n'a pas de jeton)
  - le serveur fait sa réponse en ajoutant un cookie dans les entêtes HTTP de celle-ci. Cela se fait à l'aide de la commande HTTP **Set-Cookie** :

    **Set-Cookie: param1=valeur1;param2=valeur2;....**

    où *parami* sont des noms de paramètres et *valeursi* leurs valeurs. Parmi les paramètres, il y aura le jeton. Bien souvent, il n'y a que le jeton dans le cookie, les autres informations étant consignées par le serveur dans le dossier lié au jeton. Le navigateur qui reçoit le cookie va le stocker dans un fichier sur le disque. Après la réponse du serveur, la connexion est fermée (le client quitte le guichet avec son jeton).
  - le client fait sa seconde demande au serveur. A chaque fois qu'une demande à un serveur est faite, le navigateur regarde parmi tous les cookies qu'il a, s'il en a un provenant du serveur demandé. Si oui, il l'envoie au serveur toujours sous la forme d'une commande HTTP, la commande **Cookie** qui a une syntaxe analogue à celle de la commande *Set-Cookie* utilisée par le serveur :

    **Cookie: param1=valeur1;param2=valeur2;....**

    Parmi les paramètres envoyés par le navigateur, le serveur retrouvera le jeton lui permettant de reconnaître le client et de retrouver les informations qui lui sont liées.

  C'est la forme la plus utilisée de jeton. Elle présente un inconvénient : un utilisateur peut configurer son navigateur afin qu'il n'accepte pas les cookies. Ce type d'utilisateur n'a alors pas accès aux applications web avec cookie.

- **réécriture d'URL**
  - le client fait sa première demande (le serveur le reconnaît au fait que le client n'a pas de jeton)
  - le serveur envoie sa réponse. Celle-ci contient des liens que l'utilisateur doit utiliser pour continuer l'application. Dans l'URL de chacun de ces liens, le serveur rajoute le jeton sous la forme *URL;jeton=valeur*.

- lorsque l'utilisateur clique sur l'un des liens pour continuer l'application, le navigateur fait sa demande au serveur web en lui envoyant dans les entêtes HTTP l'URL *URL;jeton=valeur* demandée. Le serveur est alors capable de récupérer le jeton.

## 3.2 L'API Java pour le suivi de session

Nous présentons maintenant les principales méthodes utiles au suivi de session :

| | |
|---|---|
| `HttpSession [HttpServletRequest].getSession()` | obtient l'objet *Session* auquel appartient la requête en cours. Si celle-ci ne faisait pas encore partie d'une session, cette dernière est créée. |
| `String [HttpSession].getId()` | idntifiant de la session en cours |
| `long [HttpSession].getCreationTime()` | date de création de la session en cours (nombre de millisecondes écoulées depuis le 1er janvier 1970, 0h). |
| `long [HttpSession].getLastAccessedTime()` | date du dernier accès à la session par le client |
| `long [HttpSession].getMaxInactiveInterval()` | durée maximale en secondes d'inactivité d'une session. Au bout de ce laps de temps, la session est invalidée. |
| `[HttpSession].setMaxInactiveInterval(int durée)` | fixe en secondes la durée maximale d'inactivité d'une session. Au bout de ce laps de temps, la session est invalidée. |
| `boolean [HttpSession].isNew()` | vrai si la session vient d'être créée |
| `[HttpSession].setAttribute(String paramètre, Object valeur)` | associe une valeur à un paramètre dans une session donnée. C'est ce mécanisme qui permet de mémoriser des informations qui resteront disponibles tout au long de la session. |
| `[HttpSession].removeAttribute(String paramètre)` | enlève *parametre* des données de la session. |
| `Object [HttpSession].getAttribute(String paramètre)` | valeur associée au paramètre *paramètre* de la session. Rend *null* si ce dernier n'existe pas. |
| `Enumeration [HttpSession].getAttributeNames()` | liste sous forme d'énumération de tous les attributs de la session en cours |
| `[HttpSession].invalidate()` | clôt la session en cours. Toutes les informations associées à celle-ci sont détruites. |

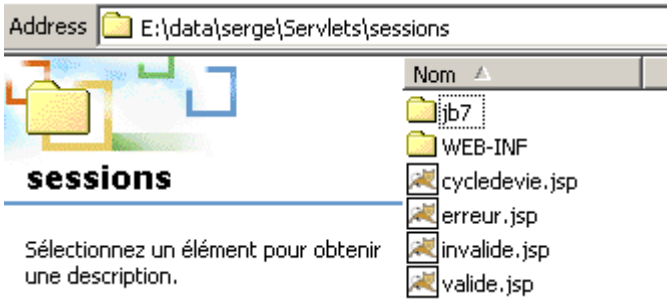
## 3.3 Exemple 1

Nous présentons un exemple tiré de l'excellent livre "Programmation avec J2EE" aux éditions Wrox et distribué par Eyrolles. Ce livre est une mine d'informations de haut niveau pour les développeurs de solutions Web en Java. L'application présentée dans ce livre sous la forme d'une unique servlet Java a été ici reprise ici sous la forme d'une servlet principale faisant appel à des pages JSP pour l'affichage des diverses réponses possibles au client.

L'application s'appelle **sessions** et est configurée de la façon suivante dans le fichier *<tomcat>\conf\server.xml* :

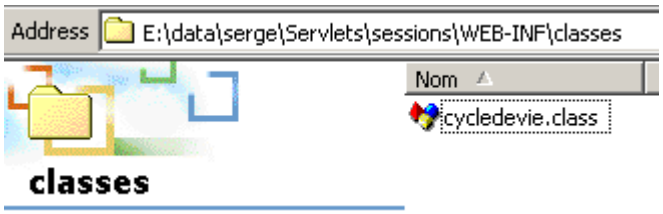
```
        <Context path="/sessions" docBase="e:/data/serge/servlets/sessions" />
```

Dans le dossier *docBase* ci-dessus, on trouve les éléments suivants :




Les fichiers **erreur.jsp**, **invalide.jsp**, **valide.jsp** sont tous trois associés à l'application **sessions**. Dans le dossier *WEB-INF* ci-dessus on trouve :


On voit ci-dessus le fichier *web.xml* de configuration de l'application **sessions**. Dans le dossier *classes* on trouve le fichier classe de la servlet :




Le fichier *web.xml* de l'application est le suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>cycledevie</servlet-name>
    <servlet-class>cycledevie</servlet-class>
    <init-param>
      <param-name>urlSessionValide</param-name>
      <param-value>/valide.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlSessionInvalide</param-name>
      <param-value>/invalide.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreur</param-name>
      <param-value>/erreur.jsp</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>cycledevie</servlet-name>
    <url-pattern>/cycledevie</url-pattern>
  </servlet-mapping>
</web-app>
```

La servlet principale s'appelle **cycledevie** (*servlet-name*) et est associée au fichier classe **cycledevie.class** (*servlet-class*). Elle a un alias **/cycledevie** (*servlet-mapping*) qui permet de l'appeler via l'URL *http://localhost:8080/sessions/cycledevie*. Elle a trois paramètres d'initialisation :

| | |
|---|---|
| `urlSessionValide` | url de la page qui présente les caractéristiques de la session en cours |
| `urlSessionInvalide` | url de la page présentée après une invalidation de la session en cours |
| `urlErreur` | url de la page présentée en cas d'erreur d'initialisation de la servlet principale **cycledevie** |

Les composantes de l'application **sessions** sont les suivantes :

`cycledevie` servlet principale - analyse la requête du client :

- si celle-ci fait partie d'une session, passe la main à la page **valide.jsp** qui va afficher les caractéristiques de cette session. A partir de cette page, l'utilisateur peut :
  - la recharger
  - l'invalider
- si la requête demande à invalider la session en cours, la servlet passe la main à la page **invalide.jsp** qui va proposer à l'utilisateur de recréer une nouvelle session
- si lors de son initialisation, la servlet rencontre des erreurs, elle passe la main à la page **erreur.jsp** qui affichera un message d'erreur.

`valide.jsp`
- affiche les caractéristiques de la session en cours et propose deux liens :
  - un pour recharger la page et voir ainsi évoluer le paramètre du dernier accès à la session courante
  - l'autre pour invalider la session en cours

`invalide.jsp` affichée lorsque l'utilisateur a invalidé la session en cours. Propose alors d'en recréer une nouvelle.

`erreur.jsp` affichée lorsque la servlet principale rencontre des erreurs lors de son initialisation.

La servlet principale *cycledevie* est la suivante :

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

public class cycledevie extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
  String urlSessionInvalide=null;
  String urlSessionValide=null;
  String urlErreur=null;

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

    // l'initialisation s'est-elle bien passée ?
    if(msgErreur!=null){
      // on passe la main à la page d'erreur
      getServletContext().getRequestDispatcher(urlErreur).forward(request,response);
    }

    // on récupère la session en cours
    HttpSession session=request.getSession();

    // on analyse l'action à faire
    String action=request.getParameter("action");
    // invalider la session courante
    if(action!=null && action.equals("invalider")){
      // on invalide la session courante
      session.invalidate();
      // on passe la main à l'url urlSessionInvalide
      getServletContext().getRequestDispatcher(urlSessionInvalide).forward(request,response);
    }
    // autres cas
    // on passe la main à l'url urlSessionInvalide
    getServletContext().getRequestDispatcher(urlSessionValide).forward(request,response);
  }

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

  //-------- INIT
  public void init(){
    // on récupère les paramètres d'initialisation
    ServletConfig config=getServletConfig();
    urlSessionInvalide=config.getInitParameter("urlSessionInvalide");
    urlSessionValide=config.getInitParameter("urlSessionValide");
    urlErreur=config.getInitParameter("urlErreur");

    // paramètres ok ?
    if(urlSessionValide==null || urlSessionInvalide==null){
      msgErreur="Configuration incorrecte";
    }
  }
}
```

On notera les points suivants :

- dans sa méthode d'initialisation, la servlet récupère ses trois paramètres
- dans le traitement (doGet) d'une requête, la servlet :
  - vérifie tout d'abord qu'il n'y a pas eu d'erreur lors de l'initialisation. S'il y en a eu, elle passe la main à la page *erreur.jsp*.
  - vérifie la valeur du paramètre *action*. Si ce dernier a la valeur "*invalider*", la servlet passe la main à la page *invalide.jsp* sinon à la page *valide.jsp*.

La page JSP **valide.jsp** d'affichage des caractéristiques de la session courante :

```
<%@ page import="java.util.*" %>

<%
   // jspService
  // ici on est dans le cas où on doit décrire la session en cours
  String etat= session.isNew() ? "Nouvelle session" : "Ancienne session";
%>
<!-- début de la page HTML -->
  <html>
    <meta http-equiv="pragma" content="no-cache">
    <head>
      <title>Cycle de vie d'une session</title>
    </head>
    <body>
      <h3>Cycle de vie d'une session</h3>
      <hr>
      <br>Etat session : <%= etat %>
       <br>ID session : <%= session.getId() %>
       <br>Heure de création : <%= new Date(session.getCreationTime()) %>
       <br>Heure du dernier accès : <%= new Date(session.getLastAccessedTime()) %>
       <br>Intervalle maximum d'inactivité : <%= session.getMaxInactiveInterval() %>
       <br><a href="/sessions/cycledevie?action=invalider">Invalider la session</a>
       <br><a href="/sessions/cycledevie">Recharger la page</a>
    <body>
  </html>
```

On notera que dans la ligne

```
  String etat= session.isNew() ? "Nouvelle session" : "Ancienne session";
```

on utilise un objet **session** venant de nulle part. En fait cet objet fait partie des objets implicites mis à la disposition des pages JSP comme le sont les objets *request*, *response*, *out*, *config* (ServletConfig), *context* (ServletContext) déjà rencontrés. Les deux liens de la page référencent la servlet *cycledevie* présentée précédemment :

```
       <br><a href="/sessions/cycledevie?action=invalider">Invalider la session</a>
       <br><a href="/sessions/cycledevie">Recharger la page</a>
```

Le lien pour invalider la session comprend le paramètre *action=invalider* qui permettra à la servlet *cycledevie* de reconnaître le fait que l'utilisateur veut invalider la session courante. L'autre lien permet de recharger la page. Pour que le navigateur n'aille pas chercher celle-ci dans un cache, la directive HTML :

```
    <meta http-equiv="pragma" content="no-cache">
```

a été utilisée. Elle indique au navigateur de ne pas utiliser de cache pour la page qu'il reçoit.

La page **invalide.jsp** est la suivante :

```
<!-- début de la page HTML -->
<html>
  <head>
    <title>Cycle de vie d'une session</title>
  </head>
  <body>
    <h3>Cycle de vie d'une session</h3>
    <hr>
    Votre session a été invalidée
     <a href="/sessions/cycledevie">Créer une nouvelle session</a>
  </body>
</html>
```

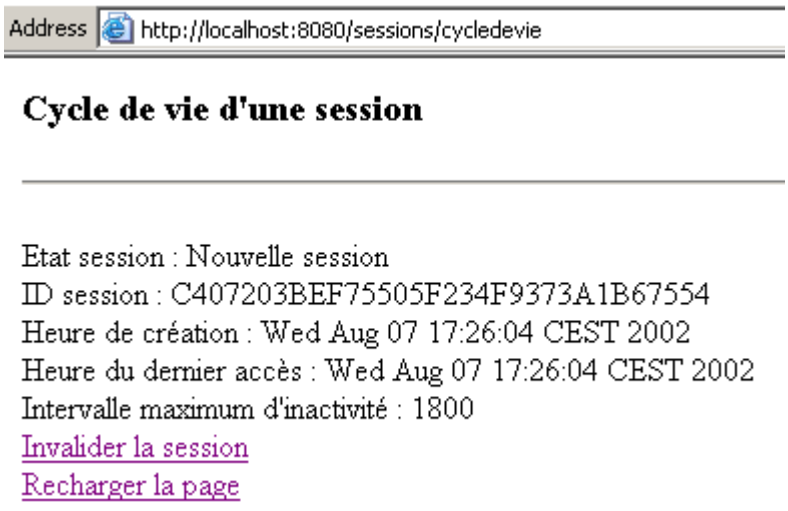
Elle offre un lien pointant sur la servlet *cycledevie* sans le paramètre *action*. Ce lien amènera la servlet *cycledevie* à créer une nouvelle session.

La page **erreur.jsp** est la suivante :

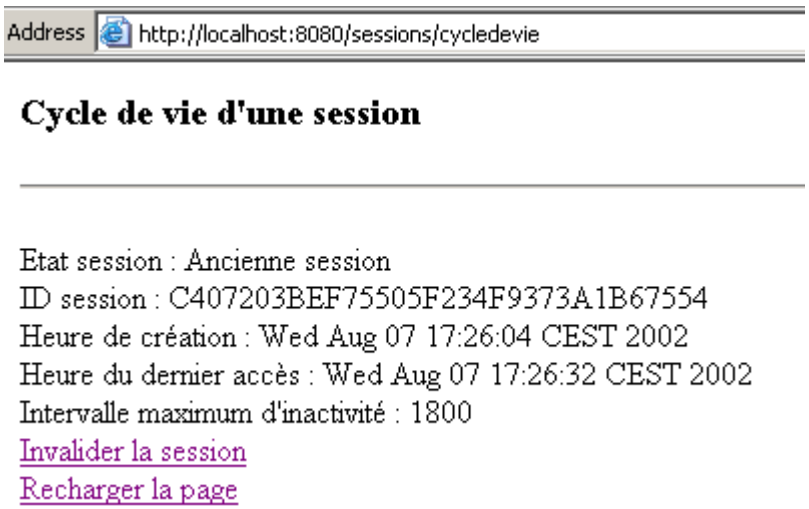
```
<%
   // jspService
  // ici on est dans le cas où on doit décrire la session en cours
  String msgErreur= request.getAttribute("msgErreur");
  if(msgErreur==null) msgErreur="Erreur non identifiée)";
```

```
%>
<!-- début de la page HTML -->
<html>
  <head>
    <title>Cycle de vie d'une session</title>
  </head>
  <body>
    <h3>Cycle de vie d'une session</h3>
    <hr>
    Application indisponible(<%= msgErreur %>)
  </body>
</html>
```

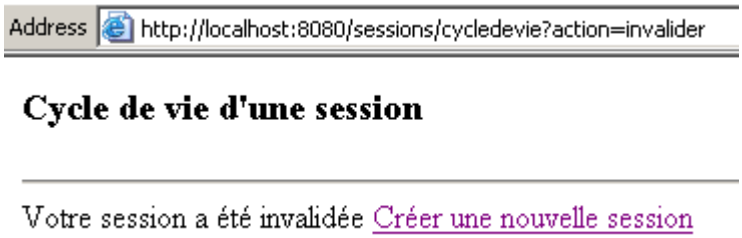
Elle a pour rôle d'afficher le message d'erreur que lui a transmis la servlet *cycledevie*. Voyons maintenant des exemples d'exécution. La servlet est demandée une première fois :




La page ci-dessus indique qu'on est dans une nouvelle session. On utilise le lien "*Recharger la page*" :

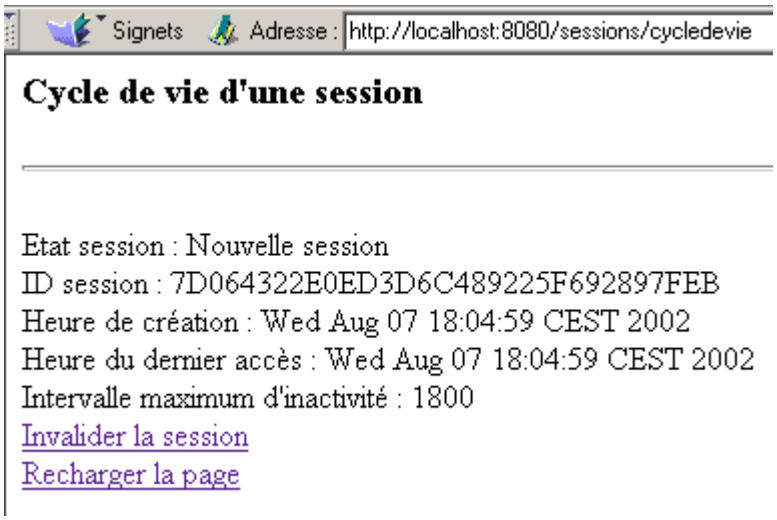



Le résultat précédent indique qu'on est toujours dans la même session que dans la page précédente (même ID). On remarquera que l'heure du dernier accès à cette session a changé. Maintenant utilisons le lien "*Invalider la session*" :

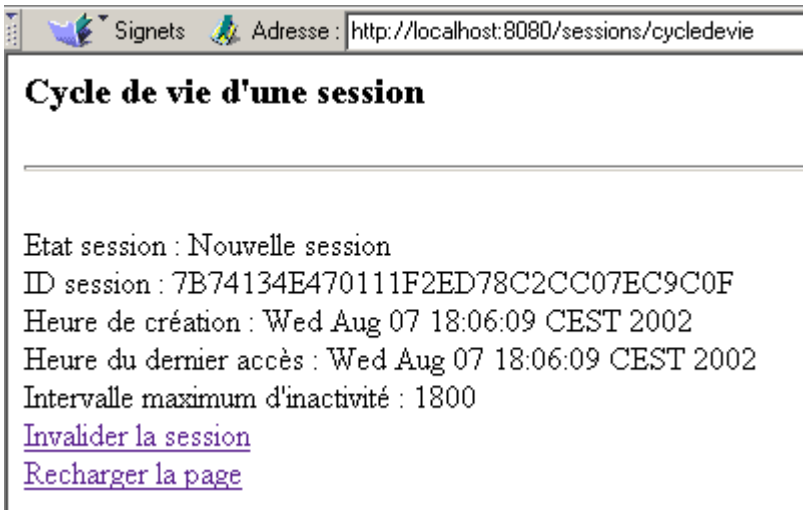



On remarquera l'URL de cette nouvelle page avec le paramètre *action=invalider*. Utilisons le lien "*Créer une nouvelle session*" pour créer une nouvelle session :


On remarque qu'une nouvelle session a démarré. Dans les exemples précédents, la session s'appuie sur le mécanisme des cookies. Inhibons maintenant l'utilisation des cookies sur notre navigateur et refaisons les tests. Les exemples suivants ont été réalisés avec Netscape Communicator. Pour une raison inexpliquée les tests réalisés avec IE6 donnaient des résultats inattendus comme si IE6 continuait à utiliser des cookies alors que ceux-ci avaient été désactivés. La servlet **cycledevie** est demandée une première fois :




Nous utilisons maintenant le lien "*Recharger la page*" :




On peut voir deux choses :

- l'ID de la session a changé
- la servlet détecte la session comme une nouvelle session

Le serveur Tomcat apporte une solution au problème des utilisateurs qui inhibent l'utilisation des cookies sur leur navigateur. Il utilise deux mécanismes pour implémenter le jeton dont on a parlé au début de ce paragraphe : les cookies et la réécriture d'URL. Si le cookie de session n'est pas disponible, il essaiera d'obtenir le jeton à partir de l'URL demandée par le client. Pour cela, il faut que celle-ci contienne le jeton. De façon générale, il faut que tous les liens générés dans un document HTML vers l'application web contiennent le jeton de celle-ci. Cela peut se faire avec la méthode *encodeURL* :

`String [HttpResponse].encodeURL(String URL)` ajoute le jeton de la session encours à l'URL passée en paramètre sous la forme *URL;jsessionid=xxxx*

Nous modifions notre application de la façon suivante :

- dans la servlet **cycledevie.java** les URL sont encodées :

```
     // on passe la main à la page d'erreur

  getServletContext().getRequestDispatcher(response.encodeURL(urlErreur)).forward(request,response);
....
     // on passe la main à l'url urlSessionInvalide

  getServletContext().getRequestDispatcher(response.encodeURL(urlSessionInvalide)).forward(request,r
esponse);
....
    // on passe la main à l'url urlSessionInvalide

  getServletContext().getRequestDispatcher(response.encodeURL(urlSessionValide)).forward(request,res
ponse);
```

- dans la page **valide.jsp** les URL sont encodées :

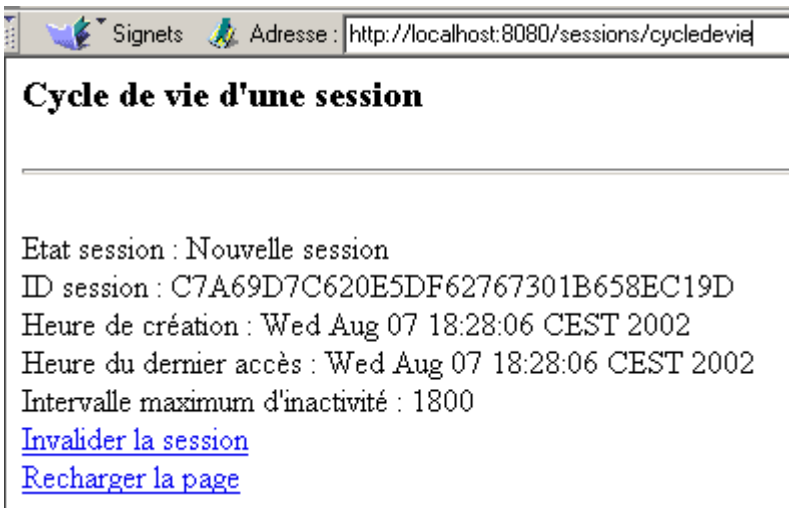
```
<%
  // jspService
  // ici on est dans le cas où on doit décrire la session en cours
  String etat= session.isNew() ? "Nouvelle session" : "Ancienne session";
  // encodage URL cycledevie
  String URLcycledevie=response.encodeURL("/sessions/cycledevie");
%>
............
      <br><a href="<%= URLcycledevie %>?action=invalider">Invalider la session</a>
      <br><a href="<%= URLcycledevie %>">Recharger la page</a>
```

- dans la page in**valide.jsp** les URL sont encodées :

```
<%
  // jspservice - on invalide la session en cours
  session.invalidate();
  // encodage URL cycledevie
  String URLcycledevie=response.encodeURL("/sessions/cycledevie");
%>
```

```
.....
    <a href="<%= URLcycledevie %>">Créer une nouvelle session</a>
```

Maintenant nous sommes prêts pour les tests. Nous utilisons Netscape 4.5 et les coookies ont été inhibés. Nous demandons une première fois la servlet **cycledevie** :




et nous rechargeons la page avec le lien "*Recharger la page*" :


Nous pouvons voir que :

- la session n'a pas changé (même ID)
- l'URL de la servlet **cycledevie** contient bien le jeton comme le montre le champ *Adresse* ci-dessus
- le serveur Tomcat récupère donc le jeton de session dans l'URL demandée (si le développeur a pris soin d'encoder celle-ci).

## 3.4 Exemple 2

Nous présentons maintenant un exemple montrant comment stocker des informations dans la session d'un client. Ici l'unique information sera un compteur qui sera incrémenté à chaque fois que l'utilisateur appellera l'URL de la servlet. Lorsque celle-ci est appelée la première fois, on a la page suivante :


Si on clique sur le lien "*Recharger la page*" ci-dessus, on obtient la nouvelle page suivante :


L'application a trois composantes :

- une servlet qui traite la requête du client
- une page jsp qui affiche la valeur du compteur
- une page jsp qui affiche une éventuelle erreur

Ces trois composantes sont installées dans l'application web **sessions** déjà utilisée. Le fichier *web.xml* de celle-ci a été modifié pour configurer les nouvelles servlets :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
...
   <servlet>
    <servlet-name>compteur</servlet-name>
     <servlet-class>compteur</servlet-class>
     <init-param>
    <param-name>urlAffichageCompteur</param-name>
      <param-value>/compteur.jsp</param-value>
     </init-param>
     <init-param>
    <param-name>urlErreur</param-name>
      <param-value>/erreurcompteur.jsp</param-value>
     </init-param>
   </servlet>
...
   <servlet-mapping>
    <servlet-name>compteur</servlet-name>
     <url-pattern>/compteur</url-pattern>
   </servlet-mapping>
</web-app>
```

- la servlet s'appelle **compteur** (*servlet-name*) et est liée au fichier classe **compteur.class** (*servlet-class*)
- elle a deux paramètres d'initialisation :
  - *urlAffichageCompteur* : URL de la page JSP d'affichage du compteur
  - *urlErreur* : URL de la page JSP d'affichage d'une éventuelle erreur
- et un alias **/compteur** qui fait qu'on l'appellera via l'URL *http://localhost:8080/sessions/compteur*

La servlet *compteur.java* est la suivante :

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

public class compteur extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
  String urlAffichageCompteur=null;
  String urlErreur=null;

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

    // l'initialisation s'est-elle bien passée ?
    if(msgErreur!=null){
      // on passe la main à la page d'erreur
      getServletContext().getRequestDispatcher(urlErreur).forward(request,response);
    }

    // on récupère la session en cours
    HttpSession session=request.getSession();
    // et le compteur
    String compteur=(String)session.getAttribute("compteur");
    if(compteur==null) compteur="0";
    // incrémentation du compteur
    try{
      compteur=""+(Integer.parseInt(compteur)+1);
    }catch(Exception ex){}
    // mémorisation compteur dans la session
    session.setAttribute("compteur",compteur);
    // et dans la requête
    request.setAttribute("compteur",compteur);

    // on passe la main à l'url d'affichage du compteur
    getServletContext().getRequestDispatcher(urlAffichageCompteur).forward(request,response);
  }

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

  //-------- INIT
  public void init(){
    // on récupère les paramètres d'initialisation
    ServletConfig config=getServletConfig();
    urlAffichageCompteur=config.getInitParameter("urlAffichageCompteur");
    urlErreur=config.getInitParameter("urlErreur");

    // paramètres ok ?
    if(urlAffichageCompteur==null){
      msgErreur="Configuration incorrecte";
    }
  }
}
```

Cette servlet a la structure des servlets déjà rencontrées. On notera simplement la gestion du compteur :

- la session est récupérée via *request.getSession()*
- le compteur est récupéré dans cette session via *session.getAttribute("compteur")*
- si on récupère une valeur *null*, c'est que la session vient de commencer. Le compteur est alors mis à 0.
- le compteur est incrémenté, remis dans la session (*session.setAttribute("compteur",compteur)*) et placé dans la requête qui va être passée à la servlet d'affichage (*request.setAttribute("compteur",compteur)*).

La page d'affichage *compteur.jsp* est la suivante :

```
<%
   // jspService
  // on récupère le compteur
  String compteur= (String) request.getAttribute("compteur");
  if(compteur==null) compteur="inconnu";
%>
<!-- début de la page HTML -->
<html>
  <head>
```

```
    <title>Comptage au fil d'une session</title>
  </head>
  <body>
    <h3>Comptage au fil d'une session (nécessite l'activation des cookies)</h3>
    <hr>
    compteur = (<%= compteur %>)
    <br><a href="/sessions/compteur">Recharger la page</a>
  </body>
</html>
```

La page ci-dessus se contente de récupérer l'attribut *compteur* (*request.getAttribute("compteur")*) que lui a passé la servlet principale et l'affiche.

La page d'erreur *erreurcompteur.jsp* est la suivante :

```
<%
  // jspService
  // une erreur s'est produite
  String msgErreur= request.getAttribute("msgErreur");
  if(msgErreur==null) msgErreur="Erreur non identifiée";
%>
<!-- début de la page HTML -->
<html>
  <head>
    <title>Comptage au fil d'une session</title>
  </head>
  <body>
    <h3>Comptage au fil d'une session (nécessite l'activation des cookies)</h3>
    <hr>
    Application indisponible(<%= msgErreur %>)
  </body>
</html>
```

## 3.5 Exemple 3

Nous nous proposons d'écrire une application java qui serait cliente de l'application *compteur* précédente. Elle l'appellerait N fois de suite où N serait passé en paramètre. Notre but est de montrer un client web programmé et la façon de gérer les cookies. Notre point de départ sera un client web générique présenté dans le polycopié Java du même auteur. Il est appellé de la façon suivante :

**clientweb URL GET/HEAD**

- URL : url demandée
- GET/HEAD : GET pour demander le code HTML de la page, HEAD pour se limiter aux seuls entêtes HTTP

Voici un exemple avec l'URL *http://localhost:8080/sessions/compteur* :

```
E:\data\serge\JAVA\SOCKETS\client web>java clientweb http://localhost:8080/sessions/compteur GET

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 14:21:18 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=B8A9076E552945009215C34A97A0EC5D;Path=/sessions


<!-- début de la page HTML -->
<html>
  <head>
        <title>Comptage au fil d'une session</title>
  </head>
  <body>
        <h3>Comptage au fil d'une session (nécessite l'activation des cookies)</h3>
        <hr>
    compteur = (1)
    <br><a href="/sessions/compteur">Recharger la page</a>
  </body>
</html>
```

Le programme *clientweb* affiche tout ce qu'il reçoit du serveur. On voit ci-dessus la commande HTTP Set-cookie avec laquelle le serveur envoie un cookie à son client. Ici le cookie contient deux informations :

- **JSESSIONID** qui est le jeton de la session

- **Path** qui définit l'URL à laquelle appartient le cookie. *Path=/sessions* indique au navigateur qu'il devra renvoyer le cookie au serveur à chaque fois qu'il demandera une URL commençant par */sessions*. Dans l'application *sessions*, nous avons utilisé différentes servlets dont les servlets */sessions/cycledevie* et */sessions/compteur*. Si on appelle la servlet */sessions/cycledevie* le navigateur va recevoir un jeton J. Si avec ce même navigateur, on appelle ensuite la servlet */sessions/compteur*, le navigateur va renvoyer au serveur le jeton J car celui-ci concerne toutes les URL commençant par */sessions*. Dans notre exemple, les servlets *cycledevie* et *compteur* n'ont pas à partager le même jeton de session. Elles n'auraient donc pas du être mises dans la même application web. C'est un point à retenir : toutes les servlets d'une même application partagent le même jeton de session.
- un cookie peut définir également **une durée de validité**. Ici cette information est absente. Le cookie sera donc détruit à la fermeture du navigateur. Un cookie peut avoir une durée de validité de N jours par exemple. Tant qu'il est valide, le navigateur le renverra à chaque fois que l'une des URL de son domaine (*Path*) sera consultée. Prenons un site de vente en ligne de CD. Celui-ci peut suivre le cheminement de son client dans son catalogue et déterminer peu à peu ses préférences : la musique classique par exemple. Ces préférences peuvent être rangées dans un cookie ayant une durée de vie de 3 mois. Si ce même client revient au bout d'un mois sur le site, le navigateur renverra le cookie à l'application serveur. Celle-ci d'après les informations renfermées dans le cookie pourra alors adapter les pages générées aux préférences de son client.

Le code du client web suit. Il sera ultérieurement le point de départ d'un autre client.

```
// paquetages importés
import java.io.*;
import java.net.*;

public class clientweb{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void main(String[] args){
    // syntaxe
    final String syntaxe="pg URI GET/HEAD";

    // nombre d'arguments
    if(args.length != 2)
      erreur(syntaxe,1);

    // on note l'URI demandée
    String URLString=args[0];
    String commande=args[1].toUpperCase();

    // vérification validité de l'URI
    URL url=null;
    try{
      url=new URL(URLString);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),2);
    }//catch
    // vérification de la commande
    if(! commande.equals("GET") && ! commande.equals("HEAD")){
      // commande incorrecte
      erreur("Le second paramètre doit être GET ou HEAD",3);
    }

    // on extrait les infos utiles de l'URL
     String path=url.getPath();
     if(path.equals("")) path="/";
     String query=url.getQuery();
     if(query!=null) query="?"+query; else query="";
     String host=url.getHost();
     int port=url.getPort();
     if(port==-1) port=url.getDefaultPort();

    // on peut travailler
    Socket  client=null;          // le client
    BufferedReader IN=null;         // le flux de lecture du client
    PrintWriter OUT=null;         // le flux d'écriture du client
    String réponse=null;          // réponse du serveur
    try{
      // on se connecte au serveur
      client=new Socket(host,port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      OUT=new PrintWriter(client.getOutputStream(),true);

      // on demande l'URL - envoi des entêtes HTTP
      OUT.println(commande + " " + path + query + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      OUT.println("Connection: close");
      OUT.println();
```

```
      // on lit la réponse
      while((réponse=IN.readLine())!=null){
        // on traite la réponse
        System.out.println(réponse);
      }//while
      // c'est fini
      client.close();
    } catch(Exception e){
      // on gère l'exception
      erreur(e.getMessage(),4);
    }//catch
  }//main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe
```

Nous créons maintenant le programme **clientCompteur** appelé de la façon suivante :

**clientCompteur URL N [JSESSIONID]**

- URL : url de la servlet compteur
- N : nombre d'appels à faire à cette servlet
- JSESSIONID : paramètre facultatif - jeton d'une session

Le but du programme est d'appelet N fois la servlet compteur en gérant le cookie de session et en affichant à chaque fois la valeur du compteur renvoyée par le serveur. A la fin des N appels, la valeur de celui-ci doit être N. Voici un premier exemple d'exécution :

```
E:\data\serge\Servlets\sessions\jb7>java.bat clientCompteur http://localhost:8080/sessions/compteur 3
--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:00 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A;Path=/sessions
cookie trouvÚ : 92DB3808CE8FCB47D47D997C8B52294A

compteur : 1

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:00 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 2

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:00 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 3
```

Le programme affiche :

- les entêtes HTTP qu'il envoie au serveur sous la forme -->
- les entêtes HTTP qu'il reçoit
- la valeur du compteur après chaque appel

On voit que lors du premier appel :

- le client n'envoie pas de cookie
- le serveur en envoie un

Pour les appels suivants :

- le client renvoie systématiquement le cookie qu'il a reçu du serveur lors du 1er appel. C'est ce qui va permettre au serveur de le reconnaître et d'incrémenter son compteur.
- le serveur lui ne renvoie plus de cookie

Nous relançons le programme précédent en passant le jeton ci-dessus comme troisième paramètre :

```
E:\data\serge\Servlets\sessions\jb7>java.bat clientCompteur http://localhost:8080/sessions/compteur 3
92DB3808CE8FCB47D47D997C8B52294A

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:25 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 4

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:25 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 5

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Cookie: JSESSIONID=92DB3808CE8FCB47D47D997C8B52294A
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:25:25 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 6
```

On voit ici que dès le 1er appel du client, le serveur reçoit un cookie de session valide. Il faut savoir que pour Tomcat la durée d'inactivité maximale d'une session est par défaut de 20 mn (c'est en fait configurable). Si le second appel du programme envoie assez vite le cookie reçu lors du premier appel, pour le serveur il s'agit alors de la même session. On pointe ici un trou de sécurité potentiel. Si je suis capable d'intercepter sur le réseau un jeton de session, je suis alors capable de me faire passer pour celui qui avait initié celle-ci. Dans notre exemple, le premier appel représente celui qui initie la session (peut-être avec un login et mot de passe qui vont lui donner le droit d'avoir un jeton) et le second appel représente celui qui a "piraté" le jeton de session du premier appel. Si l'opération en cours est une opération bancaire cela peut devenir très ennuyeux...

Le code du client est le suivant :

```
// paquetages importés
import java.io.*;
import java.net.*;
import java.util.regex.*;

public class clientCompteur{

  // demande une URL
  // affiche le contenu de celle-ci à l'écran

  public static void main(String[] args){
    // syntaxe
    final String syntaxe="pg URL-COMPTEUR N [JSESSIONID]";

    // nombre d'arguments
    if(args.length !=2 && args.length != 3)
      erreur(syntaxe,1);

    // on note l'URL demandée
    String URLString=args[0];

    // vérification validité de l'URL
    URL url=null;
    try{
      url=new URL(URLString);
    }catch (Exception ex){
      // URI incorrecte
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),2);
    }//catch
    // vérification du nombre d'appels N
    int N=0;
    try{
      N=Integer.parseInt(args[1]);
      if(N<=0) throw new Exception();
    }catch(Exception ex){
      // argument N incorrect
      erreur("Le nombre d'appels N doit être un entier >0",3);
    }
    // le jeton JSESSIONID a-t-il été passé en paramètre ?
    String JSESSIONID="";
    if (args.length==3) JSESSIONID=args[2];

    // on extrait les infos utiles de l'URL
    String path=url.getPath();
    if(path.equals("")) path="/";
    String query=url.getQuery();
    if(query!=null) query="?"+query; else query="";
    String host=url.getHost();
    int port=url.getPort();
    if(port==-1) port=url.getDefaultPort();

    // on peut travailler
    Socket  client=null;          // le client
    BufferedReader IN=null;         // le flux de lecture du client
    PrintWriter OUT=null;         // le flux d'écriture du client
    String réponse=null;          // réponse du serveur
    // le modèle recherché dans les entêtes HTTP
    Pattern modèleCookie=Pattern.compile("^Set-Cookie: JSESSIONID=(.*?);");
    // le modèle recherché dans le code HTML
    Pattern modèleCompteur=Pattern.compile("compteur = .*?(\\d+)");
    // le résultat de la comparaison au modèle
    Matcher résultat=null;
    // un booléen donnant le résultat de la recherche du compteur
    boolean compteurTrouvé;

    try{
      // on fait les N appels au serveur
      for(int i=0;i<N;i++){
        // on se connecte au serveur
        client=new Socket(host,port);

        // on crée les flux d'entrée-sortie du client TCP
        IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
        OUT=new PrintWriter(client.getOutputStream(),true);

        // on demande l'URL - envoi des entêtes HTTP
        envoie(OUT,"GET " + path + query + " HTTP/1.1");
        envoie(OUT,"Host: " + host + ":" + port);
        if(! JSESSIONID.equals("")){
          envoie(OUT,"Cookie: JSESSIONID="+JSESSIONID);
        }
        envoie(OUT,"Connection: close");
```

```
        envoie(OUT,"");
```

```
        // on lit la réponse jusqu'à la fin des entêtes en cherchant l'éventuel cookie
        while((réponse=IN.readLine())!=null){
          // suivi réponse
          System.out.println(réponse);
          // ligne vide ?
          if(réponse.equals("")) break;
          // ligne HTTP non vide
          // si on n'a pas le jeton de la session on le cherche
          if (JSESSIONID.equals("")){
            // on compare la ligne HTTP au modèle du cookie
            résultat=modèleCookie.matcher(réponse);
            if(résultat.find()){
              // on a trouvé le cookie
              JSESSIONID=résultat.group(1);
            }
          }
        }//while
```

```
        // c'est fini pour les entêtes HTTP - on passe au code HTML
        compteurTrouvé=false;
        while((réponse=IN.readLine())!=null){
          // la ligne courante contient-elle le compteur ?
          if (! compteurTrouvé){
            résultat=modèleCompteur.matcher(réponse);
            if(résultat.find()){
              // on a trouvé le compteur - on l'affiche
              System.out.println("compteur : " + résultat.group(1));
              compteurTrouvé=true;
            }
          }
        }//while
```

```
        // c'est fini
        client.close();
      }//for
    } catch(Exception e){
      // on gère l'exception
      erreur(e.getMessage(),4);
    }//catch
  }//main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur

  // suivi échanges client-serveur
  public static void envoie(PrintWriter OUT,String msg){
    // envoie message au serveur
    OUT.println(msg);
    // suivi écran
    System.out.println("--> "+msg);
  }//erreur
}//classe
```

Décortiquons les points importants de ce programme :

- on doit faire N échanges client-serveur. C'est pourquoi ceux-ci sont dans une boucle

```
      for(int i=0;i<N;i++){
```

- à chaque échange, le client ouvre une connexion TCP-IP avec le serveur. Une fois celle-ci obtenue, il envoie au serveur les entêtes HTTP de sa requête :

```
        // on demande l'URL - envoi des entêtes HTTP
        envoie(OUT,"GET " + path + query + " HTTP/1.1");
        envoie(OUT,"Host: " + host + ":" + port);
        if(! JSESSIONID.equals("")){
          envoie(OUT,"Cookie: JSESSIONID="+JSESSIONID);
        }
        envoie(OUT,"Connection: close");
        envoie(OUT,"");
```

Si le jeton JSESSIONID est disponible, il est envoyé sous la forme d'un cookie, sinon il ne l'est pas.

- Une fois sa requête envoyée, le client attend la réponse du serveur. Il commence par exploiter les entêtes HTTP de cette réponse à la recherche d'un éventuel cookie. Pour le trouver, il compare les lignes qu'il reçoit à l'expression régulière du cookie :

```
   // le modèle recherché dans les entêtes HTTP
   Pattern modèleCookie=Pattern.compile("^Set-Cookie: JSESSIONID=(.*?);");
.........................
       // on lit la réponse jusqu'à la fin des entêtes en cherchant l'éventuel cookie
       while((réponse=IN.readLine())!=null){
         // suivi réponse
         System.out.println(réponse);
         // ligne vide ?
         if(réponse.equals("")) break;
         // ligne HTTP non vide
         // si on n'a pas le jeton de la session on le cherche
         if (JSESSIONID.equals("")){
           // on compare la ligne HTTP au modèle du cookie
           résultat=modèleCookie.matcher(réponse);
           if(résultat.find()){
             // on a trouvé le cookie
             JSESSIONID=résultat.group(1);
           }
         }
       }//while
```

- lorsque le jeton aura été trouvé une première fois, il ne sera plus cherché lors des appels suivants au serveur. Lorsque les entêtes HTTP de la réponse ont été traités, on passe au code HTML de cette même réponse. Dans celle-ci, on cherche la ligne qui donne la valeur du compteur. Cette recherche est faite là aussi avec une expression régulière :

```
   // le modèle du compteur recherché dans le code HTML
   Pattern modèleCompteur=Pattern.compile("compteur = .*?(\\d+)");
.................................
       // c'est fini pour les entêtes HTTP - on passe au code HTML
       compteurTrouvé=false;
       while((réponse=IN.readLine())!=null){
         // la ligne courante contient-elle le compteur ?
         if (! compteurTrouvé){
           résultat=modèleCompteur.matcher(réponse);
           if(résultat.find()){
             // on a trouvé le compteur - on l'affiche
             System.out.println("compteur : " + résultat.group(1));
             compteurTrouvé=true;
           }
         }
       }//while
```

## 3.6 Exemple 4

Dans l'exemple précédent, le client web renvoie le jeton sous la forme d'un cookie. Nous avons vu qu'il pouvait aussi le renvoyer au sein même de l'URL demandée sous la forme *URL;jsessionid=xxx*. Vérifions-le. Le programme *clientCompteur.java* est transformé en *clientCompteur2.java* et modifié de la façon suivante :

```
....
       // on demande l'URL - envoi des entêtes HTTP
       if(JSESSIONID.equals(""))
         envoie(OUT,"GET " + path + query + " HTTP/1.1");
       else envoie(OUT,"GET " + path + query + ";jsessionid=" + JSESSIONID + " HTTP/1.1");
       envoie(OUT,"Host: " + host + ":" + port);
       envoie(OUT,"Connection: close");
       envoie(OUT,"");
....
```

Le client demande donc l'URL du compteur par **GET URL;jsessionid=xx HTTP/1.1** et n'envoie plus de cookie. C'est la seule modification. Voici les résultats d'un premier appel :

```
E:\data\serge\Servlets\sessions\jb7>java.bat clientCompteur2 http://localhost:8080/sessions/compteur 2

--> GET /sessions/compteur HTTP/1.1
--> Host: localhost:8080
--> Connection: close
-->
```

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:49:30 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=48A6DBA8357D808EC012AAF3A2AFDA63;Path=/sessions
cookie trouvÚ : 48A6DBA8357D808EC012AAF3A2AFDA63

compteur : 1

--> GET /sessions/compteur;jsessionid=48A6DBA8357D808EC012AAF3A2AFDA63 HTTP/1.1
--> Host: localhost:8080
--> Connection: close
-->

HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Thu, 08 Aug 2002 18:49:30 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)

compteur : 2
```

Lors du premier appel le client demande l'URL sans jeton de session. Le serveur lui répond en lui envoyant le jeton. Le client réinterroge alors la même URL en adjoignant le jeton reçu à celle-ci. On voit que le compteur est bien incrémenté preuve que le serveur a bien reconnu qu'il s'agissait de la même session.

## 3.7 Exemple 5

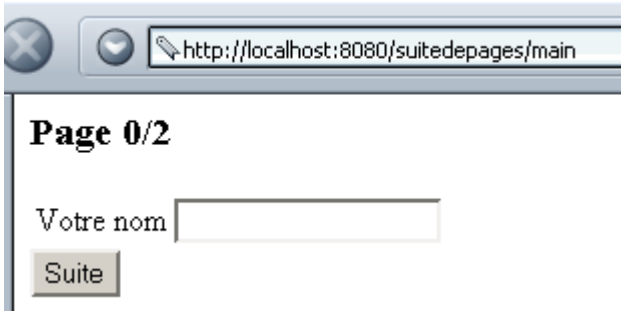
Cet exemple montre une application composée de trois pages qu'on appellera *page0*, *page1* et *page2*. L'utilisateur doit les obtenir dans cet ordre :

- **page0** est un formulaire demandant une information : un nom
- **page1** est un formulaire obtenu en réponse à l'envoi du formulaire de **page0**. Il demande une seconde information : un age
- **page2** est un document HTML qui affiche le nom obtenu par **page0** et l'âge obtenu par **page1**.

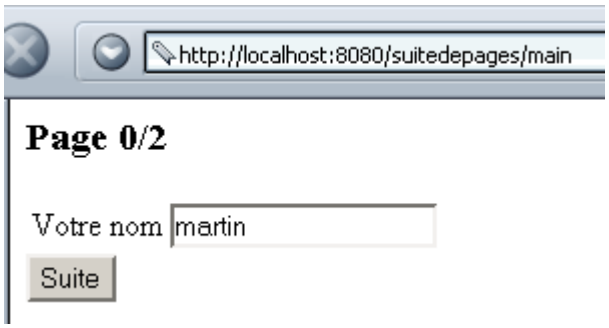
Il y a là trois échanges client-serveur :

- au premier échange le formulaire **page0** est demandé par le client et envoyé par le serveur
- au second échange le formulaire **page1** est demandé par le client et envoyé par le serveur. Le client envoie le nom au serveur.
- au troisième échange le document **page3** est demandé par le client et envoyé par le serveur. Le client envoie l'âge au serveur. Le document *page3* doit afficher le nom et l'âge. Le nom a été obtenu par le serveur au second échange et "oublié" depuis. On utilise une session pour enregistrer le nom à l'échange 2 afin qu'il soit disponible lors de l'échange 3.

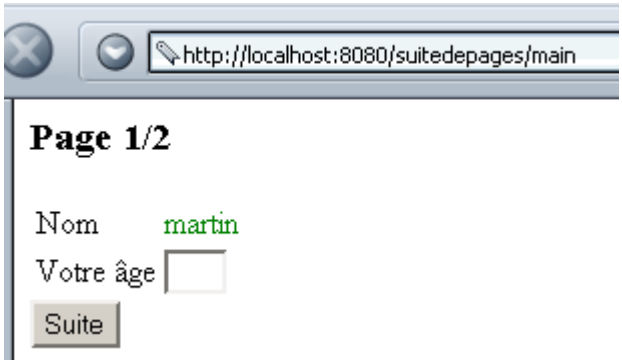
La page **page0** obtenue au premier échange est la suivante :



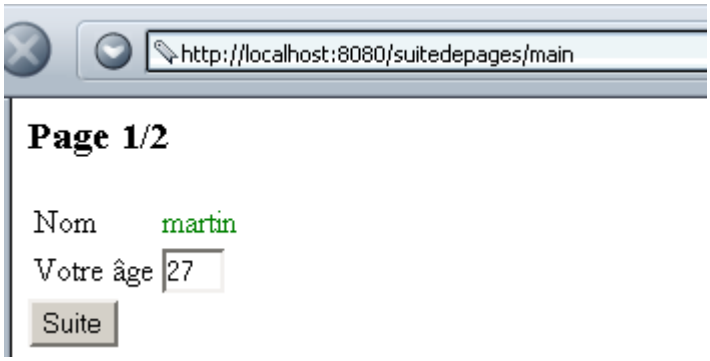

On remplit le champ du nom :



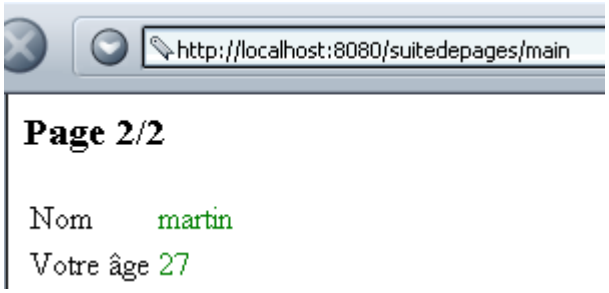

On utilise le bouton *Suite* et on obtient alors la page **page1** suivante :



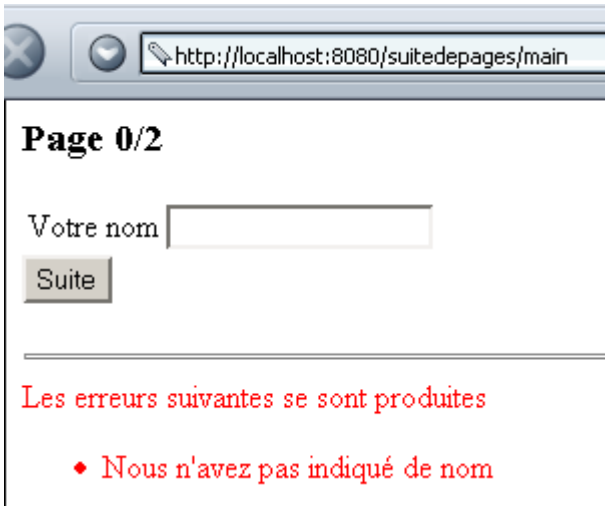

On remplit le champ de l'âge :



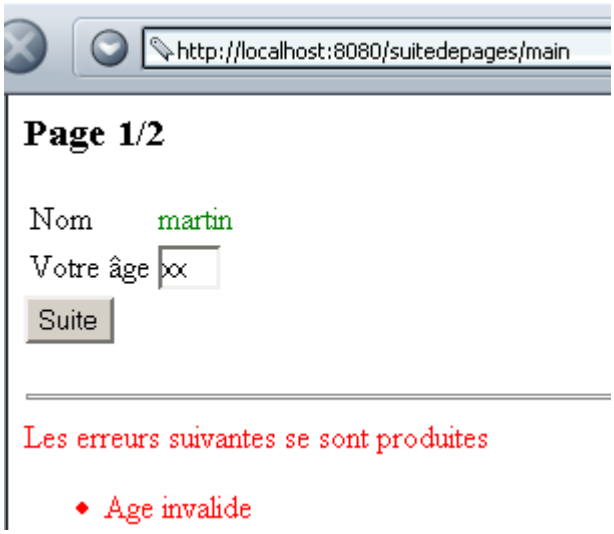

On utilise le bouton *Suite* et on obtient alors la page **page2** suivante :




Lorsqu'on soumet la page **page0** au serveur, celui-ci peut la renvoyer avec un code d'erreur si le nom est vide :




Lorsqu'on soumet la page **page1** au serveur, celui-ci peut la renvoyer avec un code d'erreur si l'âge est invalide :




L'application est composée d'une servlet et de quatre pages JSP :

| | |
|---|---|
| `page0.jsp` | affiche page0 |
| `page1.jsp` | affiche page1 |
| `page2.jsp` | affiche page2 |
| `erreur.jsp` | affiche une page d'erreur |

L'application web s'appelle **suitedepages** et est configurée comme suit dans le fichier *server.xml* de Tomcat :

```
        <Context path="/suitedepages" docBase="e:/data/serge/servlets/suitedepages" />
```

Le fichier de configuration *web.xml* de l'application suitedepages est le suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>main</servlet-class>
    <init-param>
      <param-name>urlPage0</param-name>
      <param-value>/page0.jsp</param-value>
```

```
    </init-param>
    <init-param>
      <param-name>urlPage1</param-name>
      <param-value>/page1.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlPage2</param-name>
      <param-value>/page2.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreur</param-name>
      <param-value>/erreur.jsp</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main</url-pattern>
  </servlet-mapping>
</web-app>
```

La servlet principale s'appelle **main** et grâce à son alias (servlet-mapping) est accessible via l'URL *http://localhost:8080/suitedepages/main*. Elle a quatre paramètres d'initialisation qui sont les URL des quatre pages JSP utilisées pour les différents affichages. Le code de la servlet **main** est le suivant :

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

public class main extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
  String urlPage0=null;
  String urlPage1=null;
  String urlPage2=null;
  String urlErreur=null;

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

    // l'initialisation s'est-elle bien passée ?
    if(msgErreur!=null){
      // on passe la main à la page d'erreur
      getServletContext().getRequestDispatcher(urlErreur).forward(request,response);
    }
    // on récupère le paramètre étape
    String étape=request.getParameter("etape");
    // on récupère la session en cours
    HttpSession session=request.getSession();
    // on traite l'étape en cours
    if(étape==null) étape0(request,response,session);
    if(étape.equals("1")) étape1(request,response,session);
    if(étape.equals("2")) étape2(request,response,session);
    // autres cas sont invalides
    étape0(request,response,session);
  }

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

  //-------- INIT
  public void init(){
    // on récupère les paramètres d'initialisation
    ServletConfig config=getServletConfig();
    urlPage0=config.getInitParameter("urlPage0");
    urlPage1=config.getInitParameter("urlPage1");
    urlPage2=config.getInitParameter("urlPage2");
    urlErreur=config.getInitParameter("urlErreur");

    // paramètres ok ?
    if(urlPage0==null || urlPage1==null || urlPage2==null){
      msgErreur="Configuration incorrecte";
    }
  }

  //-------- étape0
```

```
  public void étape0(HttpServletRequest request, HttpServletResponse response, HttpSession session)
      throws IOException, ServletException{
    // on fixe quelques attributs
    request.setAttribute("nom","");
    // on présente la page 0
    request.getRequestDispatcher(urlPage0).forward(request,response);
  }

  //-------- étape1
  public void étape1(HttpServletRequest request, HttpServletResponse response, HttpSession session)
      throws IOException, ServletException{
    // on récupère le nom dans la requête
    String nom=request.getParameter("nom");
    // nom positionné ?
    if(nom==null) étape0(request,response,session);
    // on enlève les éventuels espaces du nom
    nom=nom.trim();
    // on le met dans un attribut de la requête
    request.setAttribute("nom",nom);
    // nom vide ?
    if(nom.equals("")){
      // c'est une erreur
      ArrayList erreurs=new ArrayList();
      erreurs.add("Nous n'avez pas indiqué de nom");
      // on met les erreurs dans la requête
      request.setAttribute("erreurs",erreurs);
      // retour à la page 0
      étape0(request,response,session);
    }
    // nom valide - on le mémorise dans la session en cours
    session.setAttribute("nom",nom);
    // on fixe l'attribut age dans la requête
    request.setAttribute("age","");
    // on présente la page 1
    request.getRequestDispatcher(urlPage1).forward(request,response);
  }

  //-------- étape2
  public void étape2(HttpServletRequest request, HttpServletResponse response, HttpSession session)
      throws IOException, ServletException{
    // on récupère le nom dans la session
    String nom=(String)session.getAttribute("nom");
    // nom positionné ?
    if(nom==null) étape0(request,response,session);
    // on le met dans un attribut de la requête
    request.setAttribute("nom",nom);
    // on récupère l'âge dans la requête
    String age=request.getParameter("age");
    // age positionné ?
    if(age==null){
      // retour à la page 1
      request.setAttribute("age","");
      request.getRequestDispatcher(urlPage1).forward(request,response);
    }
    // on mémorise l'âge dans la requête
    age=age.trim();
    request.setAttribute("age",age);
    // age valide ?
    if(! Pattern.matches("^\\s*\\d+\\s*$",age)){
      // c'est une erreur
      ArrayList erreurs=new ArrayList();
      erreurs.add("Age invalide");
      // on met les erreurs dans la requête
      request.setAttribute("erreurs",erreurs);
      // retour à la page 1
      request.getRequestDispatcher(urlPage1).forward(request,response);
    }
    // age valide - on présente la page 2
    request.getRequestDispatcher(urlPage2).forward(request,response);
  }
}
```

- la méthode *init* récupère les quatre paramètres d'initialisation et positionne un message d'erreur si l'un d'eux est manquant
- nous avons vu que la requête comprenait trois échanges. Pour savoir où on en est dans ceux-ci, les formulaires *page0* et *page1* ont une variable cachée *etape* qui a la valeur 1 (*page0*) ou 2 (*page1*). On pourrait ici voir ce numéro comme le numéro

de page suivante à afficher. Dans la méthode *doGet*, ce paramètre est récupéré dans la requête et selon sa valeur le traitement est délégué à trois autres méthodes :
    - *étape0* traite la requête initiale et envoie *page0*
    - *étape1* traite le formulaire de *page0* et envoie *page1* ou de nouveau *page0* s'il y a eu erreur
    - étape2 traite le formulaire de *page1* et envoie *page2* ou de nouveau *page1* s'il y a eu erreur
- **étape0**
    - affiche *page0* avec un nom vide
- **étape1**
    - récupère le paramètre *nom* du formulaire de *page0*.
    - vérifie que le nom existe (pas null). Si ce n'est pas le cas on affiche de nouveau *page0* comme si c'était le premier appel.
    - vérifie que le nom est non vide. Si ce n'est pas le cas on affiche de nouveau *page0* avec un message d'erreur.
    - mémorise le nom dans la session courante et affiche *page1* si le nom est valide.
- **étape2**
    - récupère le paramètre *nom* dans la session courante.
    - vérifie que le nom existe (pas null). Si ce n'est pas le cas on affiche de nouveau *page0* comme si c'était le premier appel.
    - récupère le paramètre *age* dans la requête courante envoyée par *page1*.
    - vérifie que l'âge est valide. Si ce n'est pas le cas on affiche de nouveau *page1* avec un message d'erreur.
    - mémorise le nom et l'âge comme attributs de requête et affiche *page2* si le nom et l'âge sont valides.

La page *page0.jsp* est la suivante :

```
<%@ page import="java.util.*" %>
<% // page0.jsp
   // on récupère les attributs de la requête
  String nom=(String)request.getAttribute("nom");
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
  // attributs valides ?
  if(nom==null){
   // retour à la servlet principale
    request.getRequestDispatcher("/main").forward(request,response);
  }
%>
<html>
  <head>
    <title>page 0</title>
  </head>
  <body>
    <h3>Page 0/2</h3>
    <form name="frmNom" method="POST" action="/suitedepages/main">
     <input type="hidden" name="etape" value="1">
      <table>
        <tr>
          <td>Votre nom</td>
          <td><input type="text" name="nom" value="<%= nom %>"></td>
        </tr>
      </table>
      <input type="submit" value="Suite">
    </form>
    <% // erreurs ?
      if (erreurs!=null){
    %>
      <hr>
      <font color="red">
        Les erreurs suivantes se sont produites
        <ul>
        <% for(int i=0;i<erreurs.size();i++){ %>
            <li><%= erreurs.get(i) %>
        <% }//for %>
        </ul>
     <% }//if %>
  </body>
</html>
```

- la page *page0.jsp* peut être appelée par la servlet principale dans deux cas :
    - lors de la requête initiale
    - après traitement du formulaire de *page0* lorsqu'il y a une erreur
- le paramètre *nom* à afficher lui est donné par la servlet principale ainsi que l'éventuelle liste d'erreurs. La servlet *page0.jsp* commence donc par récupérer ces deux informations.

- le formulaire est "posté" à la servlet principale avec le champ caché (hidden) *etape* qui indique à quelle étape de l'application on se trouve.

La page *page1.jsp* est la suivante :

```
<%@ page import="java.util.*" %>
```

```
<% // page1.jsp
   // on récupère les attributs de la requête
  String nom=(String)request.getAttribute("nom");
  String age=(String)request.getAttribute("age");
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
  // attributs valides ?
  if(nom==null || age==null){
   // retour à la servlet principale
    request.getRequestDispatcher("/main").forward(request,response);
  }
%>
```

```
<html>
  <head>
    <title>page 1</title>
  </head>
  <body>
    <h3>Page 1/2</h3>
    <form name="frmAge" method="POST" action="/suitedepages/main">
     <input type="hidden" name="etape" value="2">
      <table>
        <tr>
          <td>Nom</td>
          <td><font color="green"><%= nom %></font></td>
        </tr>
        <tr>
          <td>Votre âge</td>
          <td><input type="text" name="age" size="3" value="<%= age %>"></td>
        </tr>
      </table>
      <input type="submit" value="Suite">
    </form>
    <% // erreurs ?
      if (erreurs!=null){
    %>
      <hr>
      <font color="red">
        Les erreurs suivantes se sont produites
        <ul>
        <% for(int i=0;i<erreurs.size();i++){ %>
            <li><%= erreurs.get(i) %>
        <% }//for %>
        </ul>
     <% }//if %>
  </body>
</html>
```

La page *page1.jsp* a une structure analogue à celle de la page *page0.jsp* au détail près qu'elle reçoit maintenant deux attributs de la servlet principale : *nom* et *age*. Enfin la page *page2.jsp* est la suivante :

```
<%
   // page2.jsp
   // on récupère les attributs de la requête
  String nom=(String)request.getAttribute("nom");
  String age=(String)request.getAttribute("age");
  // attributs valides ?
  if(nom==null || age==null){
   // retour à la servlet principale
    request.getRequestDispatcher("/main").forward(request,response);
  }
%>
```

```
<html>
  <head>
    <title>page 2</title>
  </head>
  <body>
    <h3>Page 2/2</h3>
      <table>
        <tr>
```

```
          <td>Nom</td>
          <td><font color="green"><%= nom %></font></td>
        </tr>
        <tr>
          <td>Votre âge</td>
          <td><font color="green"><%= age %></font></td>
        </tr>
      </table>
  </body>
</html>
```

La page *page2.jsp* reçoit elle aussi les attributs *nom* et *age* de la servlet principale. Elle se contente de les afficher. Pour terminer la page *erreur.jsp* chargée d'afficher une erreur en cas d'initialisation incorrecte de la servlet est la suivante :

```
<%
  // jspService
  // une erreur s'est produite
  String msgErreur= request.getAttribute("msgErreur");
  if(msgErreur==null) msgErreur="Erreur non identifiée";
%>
<!-- début de la page HTML -->
<html>
  <head>
    <title>Suite de pages</title>
  </head>
  <body>
    <h3>Suite de pages</h3>
    <hr>
    Application indisponible(<%= msgErreur %>)
  </body>
</html>
```

Elle affiche l'attribut *msgErreur* que lui a passé la servlet principale.

En conclusion, on pourra remarquer qu'au cours des trois étapes de l'application, c'est toujours la servlet principale qui est interrogée en premier par le navigateur. Mais ce n'est pas elle qui génère la réponse à afficher mais l'une des quatre pages JSP. L'utilisateur ne voit pas ce point, le navigateur continuant à afficher dans son champ "Adresse" l'URL initialement demandée donc celle de la servlet principale.

# 4. L'application IMPOTS

## 4.1 Introduction

Nous reprenons ici l'application IMPOTS qui est utilisée à de nombreuses reprises dans le polycopié Java du même auteur. Rappelons-en la problématique. Il s'agit d'écrire une application permettant de calculer l'impôt d'un contribuable. On se place dans le cas simplifié d'un contribuable n'ayant que son seul salaire à déclarer :

- on calcule le nombre de parts du salarié **nbParts=nbEnfants/2 +1** s'il n'est pas marié, **nbEnfants/2+2** s'il est marié, où *nbEnfants* est son nombre d'enfants.
- s'il a au moins trois enfants, il a une demi-part de plus
- on calcule son revenu imposable **R=0.72*S** où S est son salaire annuel
- on calcule son coefficient familial **QF=R/nbParts**
- on calcule son impôt **I**. Considérons le tableau suivant :

| | | |
|---|---|---|
| 12620.0 | 0 | 0 |
| 13190 | 0.05 | 631 |
| 15640 | 0.1 | 1290.5 |
| 24740 | 0.15 | 2072.5 |
| 31810 | 0.2 | 3309.5 |
| 39970 | 0.25 | 4900 |
| 48360 | 0.3 | 6898.5 |
| 55790 | 0.35 | 9316.5 |
| 92970 | 0.4 | 12106 |
| 127860 | 0.45 | 16754.5 |
| 151250 | 0.50 | 23147.5 |
| 172040 | 0.55 | 30710 |
| 195000 | 0.60 | 39312 |
| 0 | 0.65 | 49062 |

Chaque ligne a 3 champs. Pour calculer l'impôt I, on recherche la première ligne où *QF<=champ1*. Par exemple, si QF=23000 on trouvera la ligne

**24740 0.15 2072.5**

L'impôt I est alors égal à **0.15*R - 2072.5*nbParts**. Si QF est tel que la relation *QF<=champ1* n'est jamais vérifiée, alors ce sont les coefficients de la dernière ligne qui sont utilisés. Ici :

0 0.65 49062

ce qui donne l'impôt I=0.65*R - 49062*nbParts.

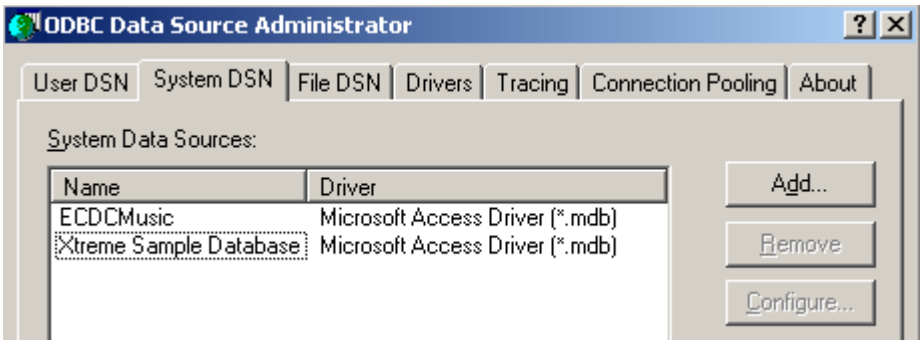
Les données définissant les différentes tranches d'impôt sont dans une base de données ODBC-MySQL. MySQL est un SGBD du domaine public utilisable sur différentes plate-formes dont Windows et Linux. Avec ce SGBD, une base de données **dbimpots** a été créée avec dedans une unique table appelée **impots**. L'accès à la base est contrôlé par un login/motdepasse ici **admimpots/mdpimpots**. La copie d'écran suivante montre comment utiliser la base **dbimpots** avec MySQL :

```
C:\Program Files\EasyPHP\mysql\bin>mysql -u admimpots -p
Enter password: *********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 18 to server version: 3.23.49-max-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use dbimpots;
Database changed

mysql> show tables;
+--------------------+
| Tables_in_dbimpots |
+--------------------+
| impots             |
+--------------------+
1 row in set (0.00 sec)

mysql> describe impots;
+---------+--------+------+-----+---------+-------+
| Field   | Type   | Null | Key | Default | Extra |
```

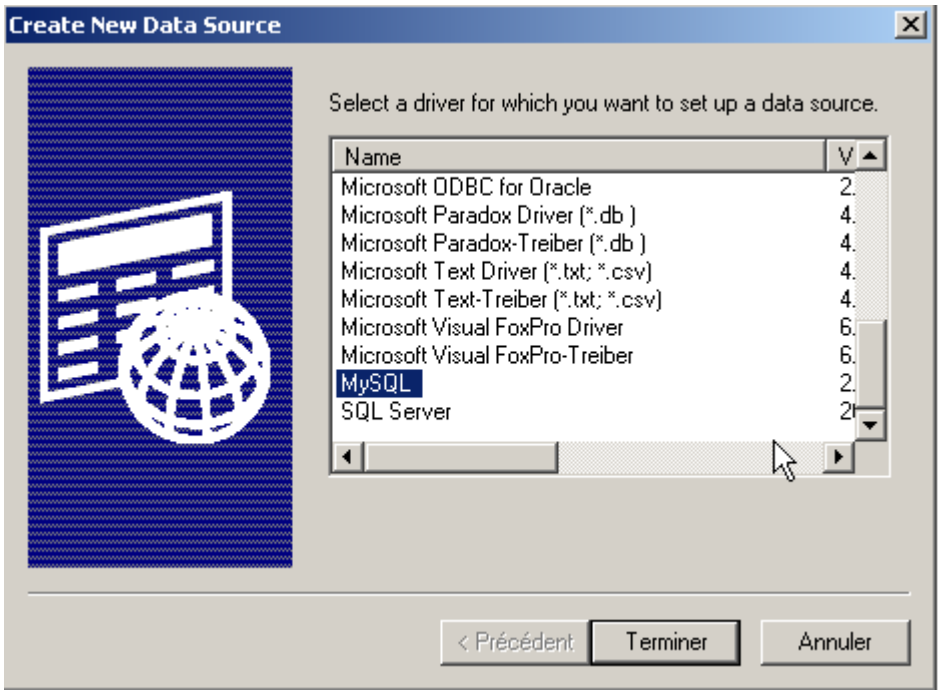
```
+---------+--------+------+-----+---------+-------+
| limites | double | YES  |     | NULL    |       |
| coeffR  | double | YES  |     | NULL    |       |
| coeffN  | double | YES  |     | NULL    |       |
+---------+--------+------+-----+---------+-------+
3 rows in set (0.02 sec)

mysql> select * from impots;
+---------+--------+---------+
| limites | coeffR | coeffN  |
+---------+--------+---------+
|   12620 |      0 |       0 |
|   13190 |   0.05 |     631 |
|   15640 |    0.1 |  1290.5 |
|   24740 |   0.15 |  2072.5 |
|   31810 |    0.2 |  3309.5 |
|   39970 |   0.25 |    4900 |
|   48360 |    0.3 |    6898 |
|   55790 |   0.35 |  9316.5 |
|   92970 |    0.4 |   12106 |
|  127860 |   0.45 |   16754 |
|  151250 |    0.5 | 23147.5 |
|  172040 |   0.55 |   30710 |
|  195000 |    0.6 |   39312 |
|       0 |   0.65 |   49062 |
+---------+--------+---------+
14 rows in set (0.00 sec)

mysql>quit
```

La base de données *dbimpots* est transformée en source de données ODBC de la façon suivante :

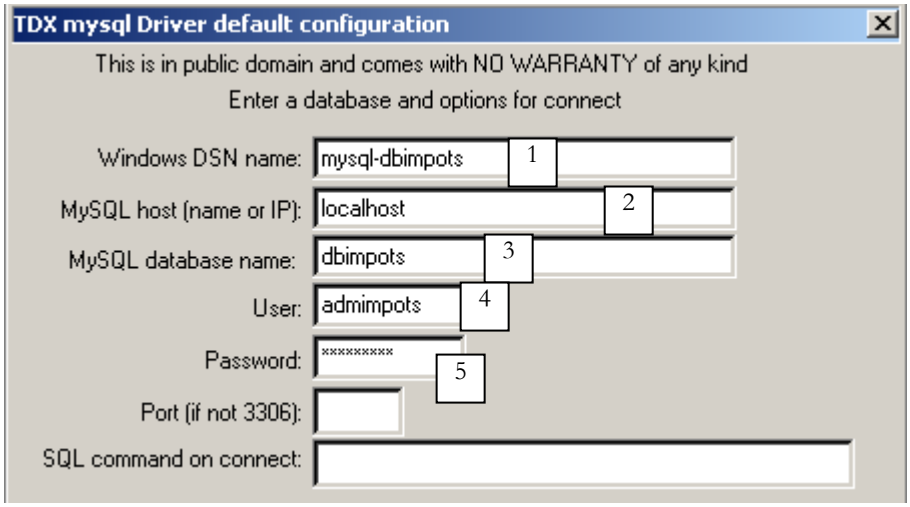
- on lance le gestionnaire des sources de données ODBC 32 bits




- on utilise le bouton [Add] pour ajouter une nouvelle source de données ODBC




- on désigne le pilote MySQL et on fait [Terminer]




- le pilote MySQL demande un certain nombre de renseignements :

**1** le nom DSN à donner à la source de données ODBC - peut-être quelconque
**2** la machine sur laquelle s'exécute le SGBD MySQL - ici *localhost*. Il est intéressant de noter que la base de données pourrait être une base de données distante. Les applications locales utilisant la source de données ODBC ne s'en apercevraient pas. Ce serait le cas notamment de notre application Java.
**3** la base de données MySQL à utiliser. MySQL est un SGBD qui gère des bases de données relationnelles qui sont des ensembles de tables reliées entre-elles par des relations. Ici, on donne le nom de la base gérée.
**4** le nom d'un utilisateur ayant un droit d'accès à cette base
**5** son mot de passe

Deux classes ont été définies pour calculer l'impôt : **impots** et **impotsJDBC**. Une instance de la classe **impots** est construite avec les données des tranches d'impôts passées en paramètres dans des tableaux :

```
// création d'une classe impots

public class impots{

  // les données nécessaires au calcul de l'impôt
  // proviennent d'une source extérieure
```

```
  protected double[] limites=null;
  protected double[] coeffR=null;
  protected double[] coeffN=null;

  // constructeur vide
  protected impots(){}

  // constructeur
  public impots(double[] LIMITES, double[] COEFFR, double[] COEFFN) throws Exception{
    // on vérifie que les 3 tableaux ont la même taille
    boolean OK=LIMITES.length==COEFFR.length && LIMITES.length==COEFFN.length;
    if (! OK) throw new Exception ("Les 3 tableaux fournis n'ont pas la même taille("+
      LIMITES.length+","+COEFFR.length+","+COEFFN.length+")");
    // c'est bon
    this.limites=LIMITES;
    this.coeffR=COEFFR;
    this.coeffN=COEFFN;
  }//constructeur

  // calcul de l'impôt
  public long calculer(boolean marié, int nbEnfants, int salaire){
    // calcul du nombre de parts
    double nbParts;
    if (marié) nbParts=(double)nbEnfants/2+2;
    else nbParts=(double)nbEnfants/2+1;
    if (nbEnfants>=3) nbParts+=0.5;
   // calcul revenu imposable & Quotient familial
   double revenu=0.72*salaire;
   double QF=revenu/nbParts;
   // calcul de l'impôt
   limites[limites.length-1]=QF+1;
   int i=0;
   while(QF>limites[i]) i++;
   // retour résultat
   return (long)(revenu*coeffR[i]-nbParts*coeffN[i]);
  }//calculer
}//classe
```

La classe **impotsJDBC** dérive de la classe **impots** précédente. Une instance de la classe **impotsJDBC** est construite avec les données des tranches d'impôts stockées dans une base de données. Les informations nécessaires pour accéder à cette base de données sont passées en paramètres au constructeur :

```
// paquetages importés
import java.sql.*;
import java.util.*;

public class impotsJDBC extends impots{
  // rajout d'un constructeur permettant de construire
  // les tableaux limites, coeffr, coeffn à partir de la table
  // impots d'une base de données
  public impotsJDBC(String dsnIMPOTS, String userIMPOTS, String mdpIMPOTS)
   throws SQLException,ClassNotFoundException{

    // dsnIMPOTS : nom DSN de la base de données
    // userIMPOTS, mdpIMPOTS : login/mot de passe d'accès à la base

    // les tableaux de données
    ArrayList aLimites=new ArrayList();
    ArrayList aCoeffR=new ArrayList();
    ArrayList aCoeffN=new ArrayList();

    // connexion à la base
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection connect=DriverManager.getConnection("jdbc:odbc:"+dsnIMPOTS,userIMPOTS,mdpIMPOTS);
    // création d'un objet Statement
    Statement S=connect.createStatement();
    // requête select
    String select="select limites, coeffr, coeffn from impots";
    // exécution de la requête
    ResultSet RS=S.executeQuery(select);
    while(RS.next()){
      // exploitation de la ligne courante
      aLimites.add(RS.getString("limites"));
      aCoeffR.add(RS.getString("coeffr"));
      aCoeffN.add(RS.getString("coeffn"));
    }// ligne suivante
    // fermeture ressources
    RS.close();
    S.close();
    connect.close();
    // transfert des données dans des tableaux bornés
    int n=aLimites.size();
```

```
    limites=new double[n];
    coeffR=new double[n];
    coeffN=new double[n];
    for(int i=0;i<n;i++){
      limites[i]=Double.parseDouble((String)aLimites.get(i));
      coeffR[i]=Double.parseDouble((String)aCoeffR.get(i));
      coeffN[i]=Double.parseDouble((String)aCoeffN.get(i));
    }//for
  }//constructeur
}//classe
```

Une fois qu'une instance de la classe **impotsJDBC** a été construite, on peut appeler de façon répétée sa méthode *calculer* afin de calculer l'impôt :

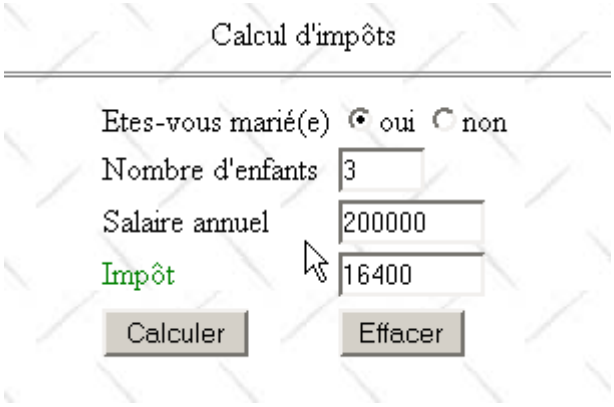
```
  public long calculer(boolean marié, int nbEnfants, int salaire){
```

L'acquisition des trois données nécessaires peut se faire de multiple façons. L'intérêt de la classe *impotsJDBC* est qu'on a juste à se préoccuper de cette acquisition. Une fois les trois information (statut marital, nombre d'enfants, salaire annuel) obtenues, l'appel à la méthode *calculer* de la classe *impotsJDBC* nous donne l'impôt à payer.

## 4.2 Version 1

On se place dans le contexte d'une application web qui présenterait une interface HTML à un utilisateur afin d'obtenir les trois paramètres nécessaires au calcul de l'impôt :

- l'état marital (marié ou non)
- le nombre d'enfants
- le salaire annuel




L'affichage du formulaire est réalisé par la page JSP suivante :

```
<%@ page import="java.util.*" %>
```

```
<%
   // on récupère les attributs passés par la servlet principale
  String chkoui=(String)request.getAttribute("chkoui");
  String chknon=(String)request.getAttribute("chknon");
  String txtEnfants=(String)request.getAttribute("txtEnfants");
  String txtSalaire=(String)request.getAttribute("txtSalaire");
  String txtImpots=(String)request.getAttribute("txtImpots");
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
%>
```

```
<html>

  <head>
    <title>impots</title>
```

```
    <script language="JavaScript" type="text/javascript">
     function effacer(){
       // raz du formulaire
        with(document.frmImpots){
         optMarie[0].checked=false;
          optMarie[1].checked=true;
          txtEnfants.value="";
          txtSalaire.value="";
          txtImpots.value="";
```

```
        }//with
      }//effacer
    </script>
  </head>

  <body background="/impots/images/standard.jpg">
    <center>
      Calcul d'impôts
      <hr>
      <form name="frmImpots" action="/impots/main" method="POST">
        <table>
          <tr>
            <td>Etes-vous marié(e)</td>
              <td>
                <input type="radio" name="optMarie" value="oui" <%= chkoui %>>oui
                <input type="radio" name="optMarie" value="non" <%= chknon %>>non
              </td>
            </tr>
            <tr>
             <td>Nombre d'enfants</td>
              <td><input type="text" size="5" name="txtEnfants" value="<%= txtEnfants %>"></td>
            </tr>
            <tr>
             <td>Salaire annuel</td>
              <td><input type="text" size="10" name="txtSalaire" value="<%= txtSalaire %>"></td>
            </tr>
            <tr>
             <td><font color="green">Impôt</font></td>
              <td><input type="text" size="10" name="txtImpots" value="<%= txtImpots %>" readonly></td>
            </tr>
            <tr></tr>
            <tr>
             <td><input type="submit" value="Calculer"></td>
              <td><input type="button" value="Effacer" onclick="effacer()"></td>
            </tr>
          </table>
        </form>
      </center>
      <%
        // y-a-t-il des erreurs
        if(erreurs!=null){
          // affichage des erreurs
          out.println("<hr>");
          out.println("<font color=\"red\">");
          out.println("Les erreurs suivantes se sont produites<br>");
          out.println("<ul>");
          for(int i=0;i<erreurs.size();i++){
           out.println("<li>"+(String)erreurs.get(i));
          }
          out.println("</ul>");
          out.println("</font>");
        }
      %>
  </body>
</html>
```

La page JSP se contente d'afficher des informations qui lui sont passées par la servlet principale de l'application :

```
  // on récupère les attributs passés par la servlet principale
  String chkoui=(String)request.getAttribute("chkoui");
  String chknon=(String)request.getAttribute("chknon");
  String txtEnfants=(String)request.getAttribute("txtEnfants");
  String txtSalaire=(String)request.getAttribute("txtSalaire");
  String txtImpots=(String)request.getAttribute("txtImpots");
  ArrayList erreurs=(ArrayList)request.getAttribute("erreurs");
```

| | |
|---|---|
| `chkoui, chknon` | attributs des boutons radio oui, non - ont pour valeurs possibles *"checked"* ou *""* afin d'allumer ou non le bouton radio correspondant |
| `txtEnfants` | le nombre d'enfants du contribuable |
| `txtSalaire` | son salaire annuel |
| `txtImpots` | le montant de l'impôt à payer |
| `erreurs` | une liste éventuelle d'erreurs si erreurs!=null |

La page envoyée au client contient un script javascript contenant une fonction *effacer* associée au bouton "*Effacer*" dont le rôle est de remettre le formulaire dans son état initial : bouton non coché, champs de saisie vides. On notera que ce résultat ne pouvait pas être

obtenu avec un bouton HTML de type "*reset*". En effet, lorsque ce type de bouton est utilisé, le navigateur remet le formulaire dans l'état où il l'a reçu. Or dans notre application, le navigateur reçoit des formulaires qui peuvent être non vides.

La servlet principale de l'application s'appelle *main.java* et son code est le suivant :

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

public class main extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
  String urlAffichageImpots=null;
  String urlErreur=null;
  String DSNimpots=null;
  String admimpots=null;
  String mdpimpots=null;
  impotsJDBC impots=null;

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

    // l'initialisation s'est-elle bien passée ?
    if(msgErreur!=null){
      // on passe la main à la page d'erreur
      request.setAttribute("msgErreur",msgErreur);
      getServletContext().getRequestDispatcher(urlErreur).forward(request,response);
    }

    // des attributs de la requête
    String chkoui=null;
    String chknon=null;
    String txtImpots=null;

    // on récupère les paramètres de la requête
    String optMarie=request.getParameter("optMarie");              // statut marital
    String txtEnfants=request.getParameter("txtEnfants");       // nbre d'enfants
    if(txtEnfants==null) txtEnfants="";
    String txtSalaire=request.getParameter("txtSalaire");       // salaire annuel
    if(txtSalaire==null) txtSalaire="";

    // a-t-on tous les paramètres attendus
    if(optMarie==null || txtEnfants==null || txtSalaire==null){
      // il manque des paramètres
      request.setAttribute("chkoui","");
      request.setAttribute("chknon","checked");
      request.setAttribute("txtEnfants","");
      request.setAttribute("txtSalaire","");
      request.setAttribute("txtImpots","");
      // on passe la main à l'url d'affichage de l'impôt
      getServletContext().getRequestDispatcher(urlAffichageImpots).forward(request,response);
    }

    // on a tous les paramètres - on les vérifie
    ArrayList erreurs=new ArrayList();
    // état marital
    if( ! optMarie.equals("oui") && ! optMarie.equals("non")){
      // erreur
      erreurs.add("Etat marital incorrect");
      optMarie="non";
    }
    // nombre d'enfants
    txtEnfants=txtEnfants.trim();
    if(! Pattern.matches("^\\d+$",txtEnfants)){
      // erreur
      erreurs.add("Nombre d'enfants incorrect");
    }
    // salaire
    txtSalaire=txtSalaire.trim();
    if(! Pattern.matches("^\\d+$",txtSalaire)){
      // erreur
      erreurs.add("Salaire incorrect");
    }
```

```
    // s'il y a des erreurs, on les passe en attribut de la requête
    if(erreurs.size()!=0){
      request.setAttribute("erreurs",erreurs);
      txtImpots="";
    }else{
      // on peut calculer l'impôt à payer
      try{
        int nbEnfants=Integer.parseInt(txtEnfants);
        int salaire=Integer.parseInt(txtSalaire);
        txtImpots=""+impots.calculer(optMarie.equals("oui"),nbEnfants,salaire);
      }catch(Exception ex){}
    }

    // les autres attributs de la requête
    if(optMarie.equals("oui")){
      request.setAttribute("chkoui","checked");
      request.setAttribute("chknon","");
    }else{
      request.setAttribute("chknon","checked");
      request.setAttribute("chkoui","");
    }
    request.setAttribute("txtEnfants",txtEnfants);
    request.setAttribute("txtSalaire",txtSalaire);
    request.setAttribute("txtImpots",txtImpots);

    // on passe la main à l'url d'affichage de l'impôt
    getServletContext().getRequestDispatcher(urlAffichageImpots).forward(request,response);
  }

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

  //-------- INIT
  public void init(){
    // on récupère les paramètres d'initialisation
    ServletConfig config=getServletConfig();
    urlAffichageImpots=config.getInitParameter("urlAffichageImpots");
    urlErreur=config.getInitParameter("urlErreur");
    DSNimpots=config.getInitParameter("DSNimpots");
    admimpots=config.getInitParameter("admimpots");
    mdpimpots=config.getInitParameter("mdpimpots");

    // paramètres ok ?
    if(urlAffichageImpots==null || DSNimpots==null || admimpots==null || mdpimpots==null){
      msgErreur="Configuration incorrecte";
      return;
    }

    // on crée une instance d'impotsJDBC
    try{
      impots=new impotsJDBC(DSNimpots,admimpots,mdpimpots);
    }catch(Exception ex){
      msgErreur=ex.getMessage();
    }
  }
}
```

- la méthode **init** de la servlet fait deux choses :
  - elle récupère ses paramètres d'initialisation. Ceux-ci lui permettent de se connecter à la base de données ODBC contenant les données des différentes transches d'impôts (DSNimpots, admimpots, mdpimpots) et les URL des pages associées à l'application : *urlAffichageImpots* pour le formulaire, *urlErreur* pour la page d'erreur.
  - elle crée une instance de la classe *impotsJDBC*

  Dans les deux cas, les erreurs possibles sont gérées et le message d'erreur placé dans la variable *msgErreur*.
- la méthode **doGET**
  - vérifie tout d'abord que la servlet s'est initialisée correctement. Si ce n'est pas le cas, la page d'erreur est affichée
  - récupère les paramètres attendus du formulaire d'impôts : *optMarie*, *txtEnfants*, *txtSalaire*. Si l'un d'eux manque (==*null*) un formulaire d'impôts vide est envoyé. On pourrait se dire que la vérification de la validité du paramètre *optMarie* est inutile. Celui-ci est la valeur d'un bouton radio et ne peut avoir ici que l'une des valeurs "*oui*" et "*non*". C'est oublier que rien n'empêche un programme d'interroger directement la servlet en lui envoyant les paramètres qu'il veut. On ne peut jamais être assuré d'être réellement connecté à un navigateur. Oublier ce point peut amener à des trous de sécurité dans l'application et il est d'ailleurs fréquent d'en trouver même dans des applications commerciales.

- la validité de chacun des trois paramètres récupérés est vérifié. Chaque erreur trouvée est ajoutée à une liste d'erreurs (*ArrayList erreurs*). S'il n'y a pas d'erreurs, le montant de l'impôt est calculé sinon il ne l'est pas.
- les informations nécessaires à l'affichage de la page sont mis en attributs de la requête et le formulaire d'impôts est ensuite affiché

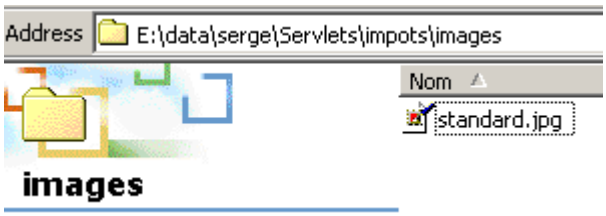
La page JSP d'erreur est la suivante :

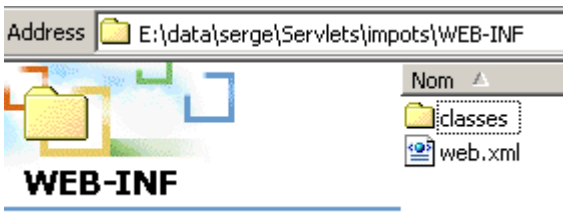
```
<%
  // jspService
  // une erreur s'est produite
  String msgErreur= (String)request.getAttribute("msgErreur");
  if(msgErreur==null) msgErreur="Erreur non identifiée";
%>
<!-- début de la page HTML -->
<html>
  <head>
    <title>impots</title>
  </head>
  <body>
    <h3>calcul d'impots</h3>
    <hr>
    Application indisponible(<%= msgErreur %>)
  </body>
</html>
```

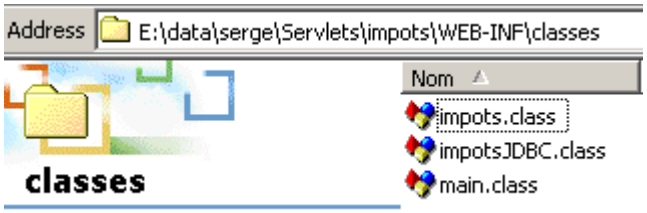
L'aplication web s'appelle **impots** et est configuré dans le fichier *server.xml* de Tomcat de la façon suivante :

```
        <Context path="/impots" docBase="e:/data/serge/servlets/impots" />
```

Le dossier de l'application contient les dossiers et fichiers suivants :

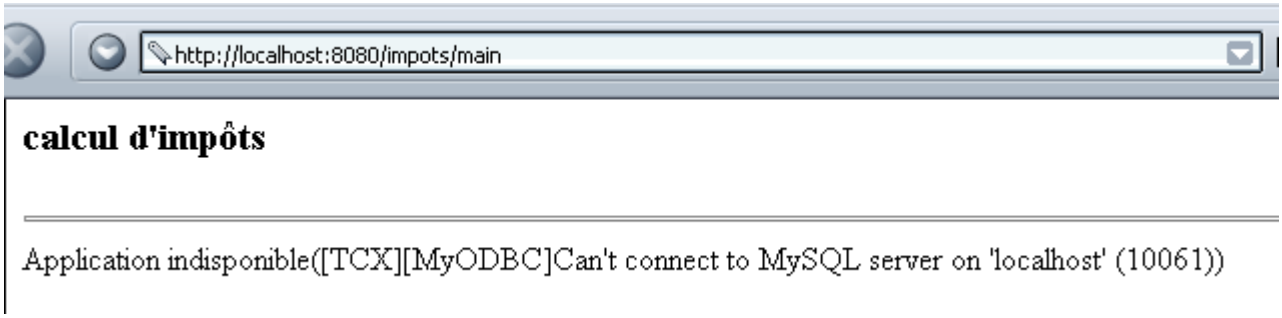
Le fichier *web.xml* de configuration de l'application **impots** est le suivant :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>main</servlet-name>
    <servlet-class>main</servlet-class>
    <init-param>
      <param-name>urlAffichageImpots</param-name>
      <param-value>/impots.jsp</param-value>
    </init-param>
    <init-param>
      <param-name>DSNimpots</param-name>
      <param-value>mysql-dbimpots</param-value>
    </init-param>
    <init-param>
      <param-name>admimpots</param-name>
      <param-value>admimpots</param-value>
    </init-param>
    <init-param>
      <param-name>mdpimpots</param-name>
      <param-value>mdpimpots</param-value>
    </init-param>
    <init-param>
      <param-name>urlErreur</param-name>
      <param-value>/erreur.jsp</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>main</servlet-name>
    <url-pattern>/main</url-pattern>
  </servlet-mapping>
</web-app>
```

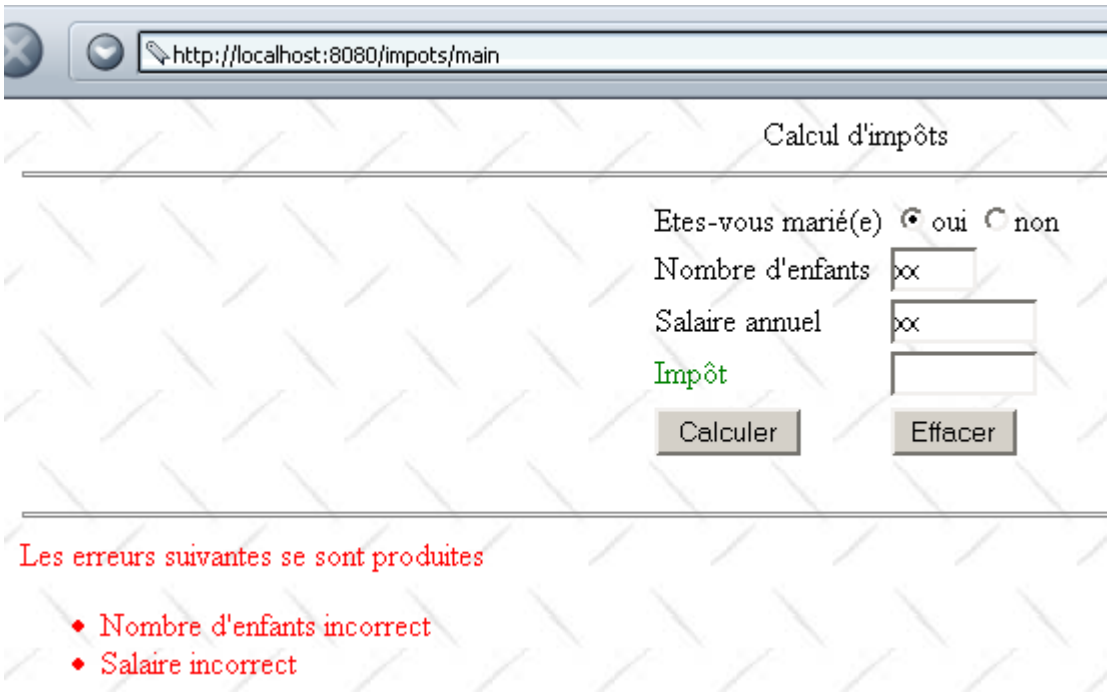
La servlet principale s'appelle *main* et a l'alias */main*. Elle est donc accessible via l'URL *http://localhost:8080/impots/main*.

Voici quelques exemples d'application :

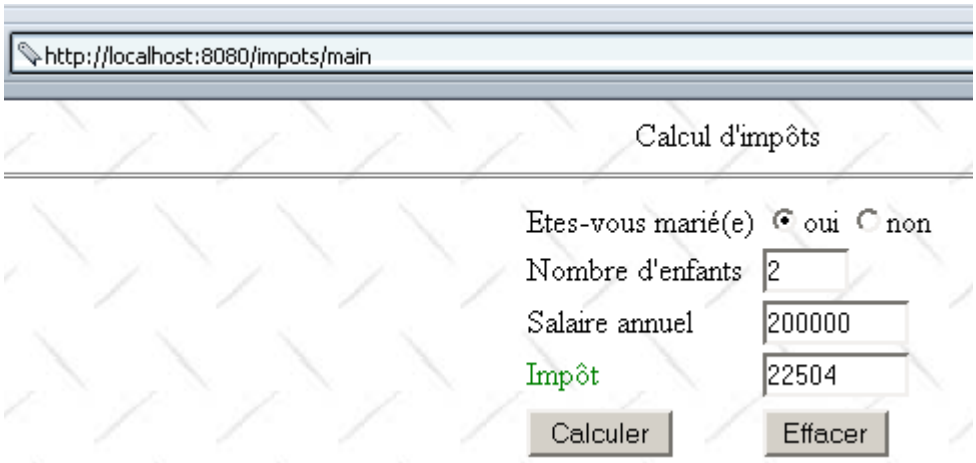
Pour s'initialiser correctement, la servlet doit avoir accès à la base de données *mysql-dbimpots*. Voici la page obtenue si par exemple le serveur MySQL n'est pas lancé et la base *mysql-dbimpots* en conséquence inaccessible :




La page obtenue en cas de saisie incorrecte est la suivante :




Si les saisie sont correctes, l'impôt est calculé :




## 4.3 Version 2

Dans l'exemple précédent, la validité des paramètres *txtEnfants*, *txtSalaire* du formulaire est vérifiée par le serveur. On se propose ici de la vérifier par un script Javascript inclus dans la page du formulaire. C'est alors le navigateur qui fait la vérification des paramètres. Le serveur n'est alors sollicité que si ceux-ci sont valides. On économise ainsi de la "bande passante". La page JSP d'affichage devient la suivante :

```
<%@ page import="java.util.*" %>
............
<html>

  <head>
    <title>impots</title>
     <script language="JavaScript" type="text/javascript">
      function effacer(){
.......
      }//effacer
```

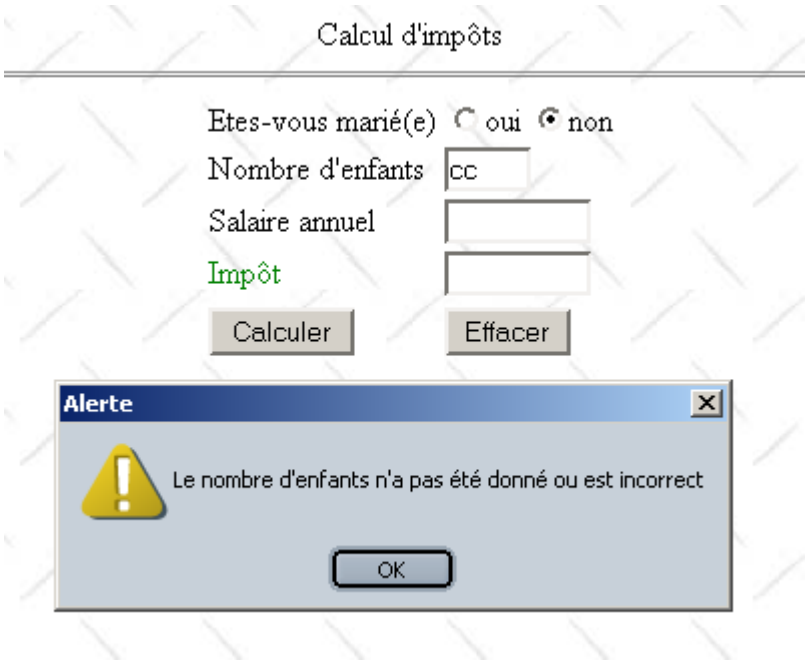
```
      function calculer(){
       // vérification des paramètres avant de les envoyer au serveur
```

```
        with(document.frmImpots){
          //nbre d'enfants
          champs=/^\s*(\d+)\s*$/.exec(txtEnfants.value);
          if(champs==null){
            // le modéle n'est pas vérifié
            alert("Le nombre d'enfants n'a pas été donné ou est incorrect");
            nbEnfants.focus();
            return;
          }//if
          //salaire
          champs=/^\s*(\d+)\s*$/.exec(txtSalaire.value);
          if(champs==null){
            // le modéle n'est pas vérifié
            alert("Le salaire n'a pas été donné ou est incorrect");
            salaire.focus();
            return;
          }//if
          // c'est bon - on envoie le formulaire au serveur
          submit();
        }//with
      }//calculer
    </script>
  </head>

  <body background="/impots/images/standard.jpg">
........
          <td><input type="button" value="Calculer" onclick="calculer()"></td>
          <td><input type="button" value="Effacer" onclick="effacer()"></td>
.....
 </body>
</html>
```

On notera les changements suivants :

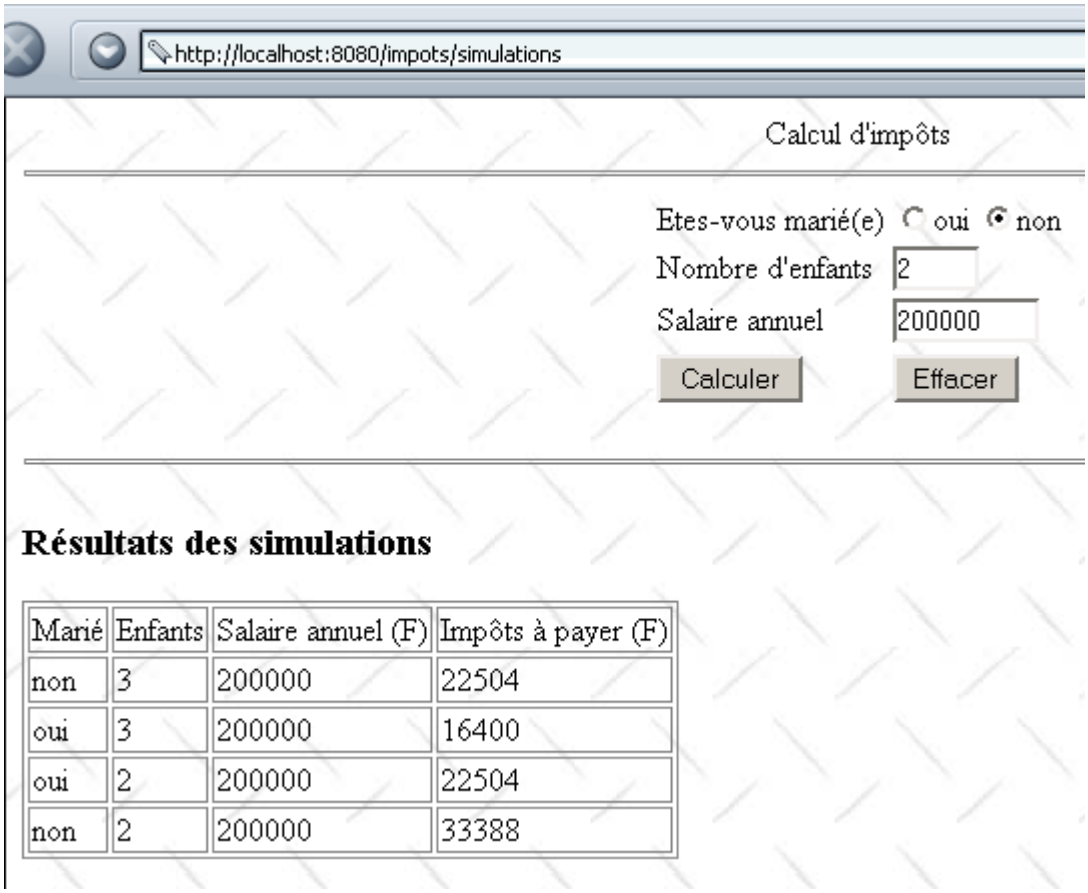
- le bouton *Calculer* n'est plus de type *submit* mais de type *button* associé à une fonction appelée *calculer*. C'est celle-ci qui fera l'analyse des champs *txtEnfants* et *txTsalaire*. S'ils sont corrects, les valeurs du formulaire seront envoyées au serveur (submit) sinon un message d'erreur sera affiché.

Voici un exemple d'affichage en cas d'erreur :




## 4.4 Version 3

Nous modifions légèrement l'application pour y introduire la notion de session. Nous considérons maintenant que l'application est une application de simulation de calcul d'impôts. Un utilisateur peut alors simuler différentes "configurations" de contribuable et voir quelle serait pour chacune d'elles l'impôt à payer. La page WEb ci-dessous donne un exemple de ce qui pourrait être obtenu :




La servlet principale est modifiée et s'appelle désormais **simulations**. Elle est configurée comme suit au sein de l'application **impots** :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  ....
</servlet>
  <servlet>
    <servlet-name>simulations</servlet-name>
     <servlet-class>simulations</servlet-class>
     <init-param>
     <param-name>urlSimulationImpots</param-name>
       <param-value>/simulationsImpots.jsp</param-value>
     </init-param>
     <init-param>
       <param-name>DSNimpots</param-name>
       <param-value>mysql-dbimpots</param-value>
     </init-param>
     <init-param>
       <param-name>admimpots</param-name>
       <param-value>admimpots</param-value>
     </init-param>
     <init-param>
       <param-name>mdpimpots</param-name>
       <param-value>mdpimpots</param-value>
     </init-param>
     <init-param>
       <param-name>urlErreur</param-name>
       <param-value>/erreur.jsp</param-value>
     </init-param>
  </servlet>
```

```
......
  <servlet-mapping>
    <servlet-name>simulations</servlet-name>
     <url-pattern>/simulations</url-pattern>
  </servlet-mapping>
</web-app>
```

La servlet principale s'appelle **simulations** et s'appuie sur le fichier classe **simulations.class**. Elle a l'alias **/simulations** qui la rend accessible via l'URL *http://localhost:8080/impots/simulations*. Elle a les mêmes paramètres d'initialisation que la servlet **main** étudiée précédemment pour ce qui est de l'accès à la base de données. Un nouveau paramètre apparaît, **urlSimulationsImpots** qui est l'URL de la page JSP de simulation (celle qui vient d'être présentée un peu plus haut).

La servlet **simulations.java** est proche de la servlet **main.java**. Elle en diffère par les points principaux suivants :

- la servlet **main** calcule une valeur *txtImpots* à partir des paramètres *optmarie*, *txtEnfants* et *txtSalaire* et passe cette valeur à la page JSP d'affichage
- la servlet **simulations** calcule de la même façon la valeur *txtImpots*, enregistre les paramètres (*optMarie*, *txtEnfants*, *txtsalaire*, *txtImpots*) dans une liste appelée **simulations**. C'est cette liste qui est passée en paramètre à la page JSP d'affichage. Afin que cette liste contienne toutes les simulations effectuées par l'utilisateur, elle est enregistrée comme attribut de la session courante.

La servlet **simulations** est la suivante ( n'ont été conservées que les lignes de code qui diffèrent de celles de l'application précédente) :

```
import java.io.*;
.......

public class simulations extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
    String urlSimulationImpots=null;
  String urlErreur=null;
...........

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

...........

    // on récupère les simulations précédentes de la session
    HttpSession session=request.getSession();
    ArrayList simulations=(ArrayList)session.getAttribute("simulations");
    if(simulations==null) simulations=new ArrayList();
    // on met les simulations dans la requête courante
    request.setAttribute("simulations",simulations);

    // d'autres attributs de la requête
...........

    // a-t-on tous les paramètres attendus
    if(optMarie==null || txtEnfants==null || txtSalaire==null){
........
      // on passe la main à l'url d'affichage des simulations de calculs d'impôts
      getServletContext().getRequestDispatcher(urlSimulationImpots).forward(request,response);
    }

    // on a tous les paramètres - on les vérifie
...........

    // s'il y a des erreurs, on les passe en attributs de la requête
    if(erreurs.size()!=0){
      request.setAttribute("erreurs",erreurs);
    }else{
      try{
        // on peut calculer l'impôt à payer
        int nbEnfants=Integer.parseInt(txtEnfants);
        int salaire=Integer.parseInt(txtSalaire);
        txtImpots=""+impots.calculer(optMarie.equals("oui"),nbEnfants,salaire);
        // on ajoute le résultat courant aux simulations précédentes
        String[] simulation={optMarie.equals("oui") ? "oui" : "non",txtEnfants, txtSalaire, txtImpots};
        simulations.add(simulation);
        // la nouvelle valeur de simulations est remise dans la session
        session.setAttribute("simulations",simulations);
```

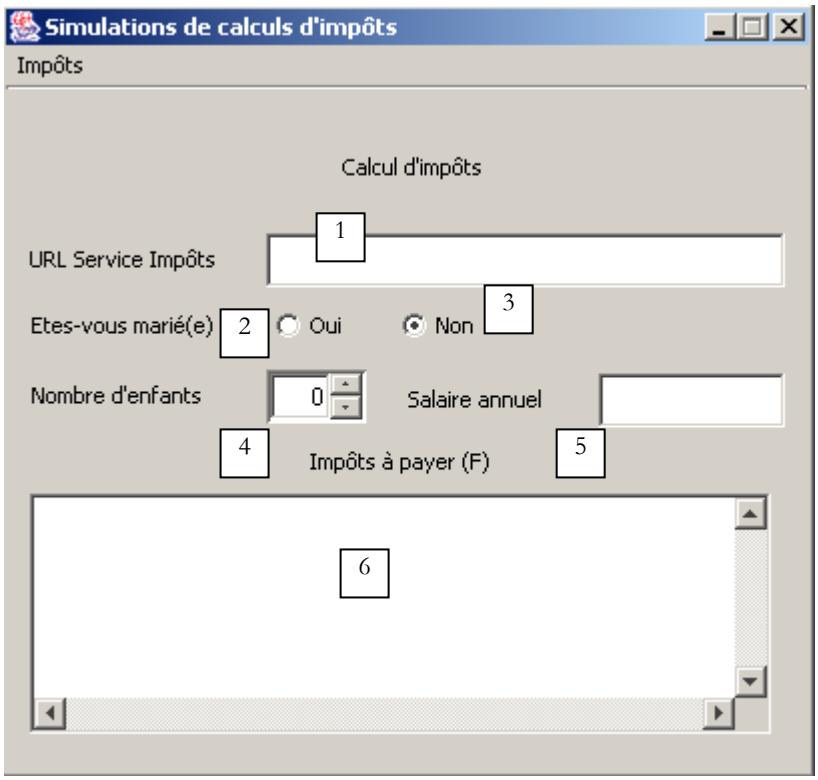
```
      }catch(Exception ex){}
    }

    // autres attributs de la requête
..........

    // on passe la main à l'url d'affichage des simulations
    getServletContext().getRequestDispatcher(urlSimulationImpots).forward(request,response);
  }

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

  //-------- INIT
  public void init(){
    // on récupère les paramètres d'initialisation
    ServletConfig config=getServletConfig();
    urlSimulationImpots=config.getInitParameter("urlSimulationImpots");
    urlErreur=config.getInitParameter("urlErreur");
    DSNimpots=config.getInitParameter("DSNimpots");
    admimpots=config.getInitParameter("admimpots");
    mdpimpots=config.getInitParameter("mdpimpots");

    // paramètres ok ?
.........................
  }
}
```

La page JSP d'affichage **simulationsImpots.jsp** est devenue la suivante (n'a été conservé que le code qui diffère de la page JSP d'affichage de l'application précédente).

```
<%@ page import="java.util.*" %>

<%
   // on récupère les attributs passés par la servlet principale
............
  ArrayList simulations=(ArrayList)request.getAttribute("simulations");
%>

<html>

  <head>
    <title>impots</title>
    <script language="JavaScript" type="text/javascript">
........
    </script>
  </head>

  <body background="/impots/images/standard.jpg">
    <center>
      Calcul d'impôts
      <hr>
      <form name="frmImpots" action="/impots/simulations" method="POST">
.......................
      </form>
    </center>
    <hr>
    <%
      // y-a-t-il des erreurs ?
      if(erreurs!=null){
..................
      }else if(simulations.size()!=0){
       // résultats des simulations
        out.println("<h3>Résultats des simulations<h3>");
        out.println("<table \"border=\"1\">");
        out.println("<tr><td>Marié</td><td>Enfants</td><td>Salaire annuel (F)</td><td>Impôts à payer
(F)</td></tr>");
        for(int i=0;i<simulations.size();i++){
         String[] simulation=(String[])simulations.get(i);

   out.println("<tr><td>"+simulation[0]+"</td><td>"+simulation[1]+"</td><td>"+simulation[2]+"</td><td>"+si
mulation[3]+"</td></tr>");
        }
        out.println("</table>");
      }
    %>
  </body>
</html>
```

## 4.5 Version 4

Nous allons maintenant créer une application autonome qui sera un client web de l'application */impots/simulations* précédente. Cette application aura l'interface graphique suivante :



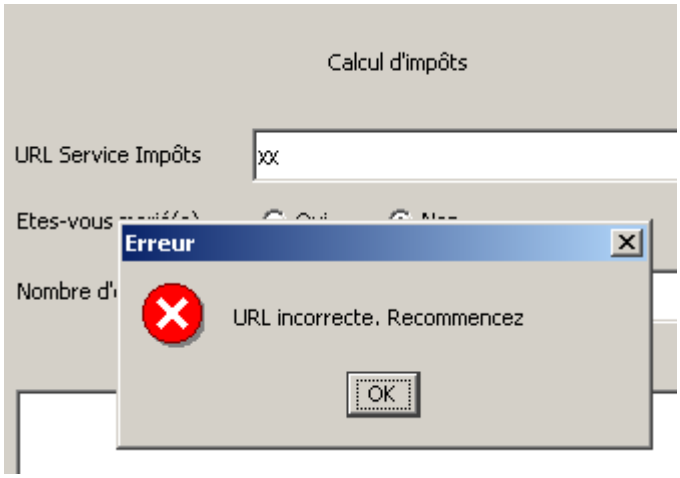

| n° | type | nom | rôle |
|---|---|---|---|
| 1 | JTextField | txtUrlServiceImpots | URL du service de simulation du calcul d'impôts |
| 2 | JRadioButton | rdOui | coché si marié |
| 3 | JRadioButton | rdNon | coché si non marié |
| 4 | JSpinner | spinEnfants | nombre d'enfants du contribuable (minimum=0, maximum=20, increment=1) |
| 5 | JTextField | txtSalaire | salaire annuel en F du contribuable |
| 6 | JList dans JScrollPane | lstSimulations | liste des simulations |

Le menu **Impôts** est composé des options suivantes :

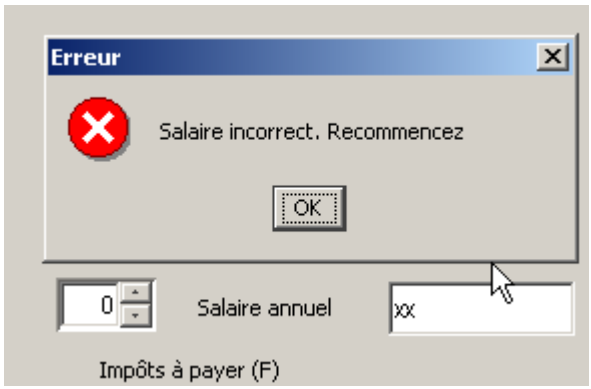
| option principale | option secondaire | nom | rôle |
|---|---|---|---|
| **Impôts** | | | |
| | **Calculer** | mnuCalculer | calcule l'impôt à payer lorsque toutes les données nécessaires au calcul sont présentes et correctes |
| | **Effacer** | mnuEffacer | remet le formulaire dans son état initial |
| | **Quitter** | mnuQuitter | termine l'application |

**Règles de fonctionnement**

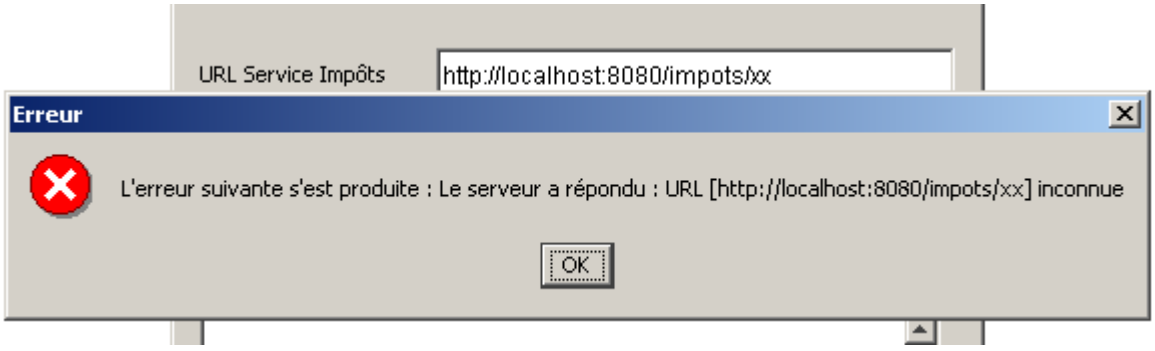
- l'option de menu *Calculer* reste éteinte si l'un des champs 1 ou 5 est vide
- une URL syntaxiquement incorrecte en 1 est détectée

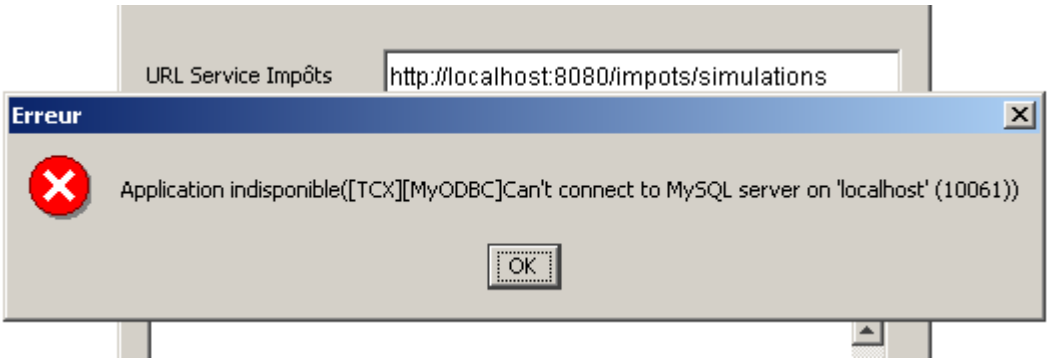



- un salaire incorrect est détecté

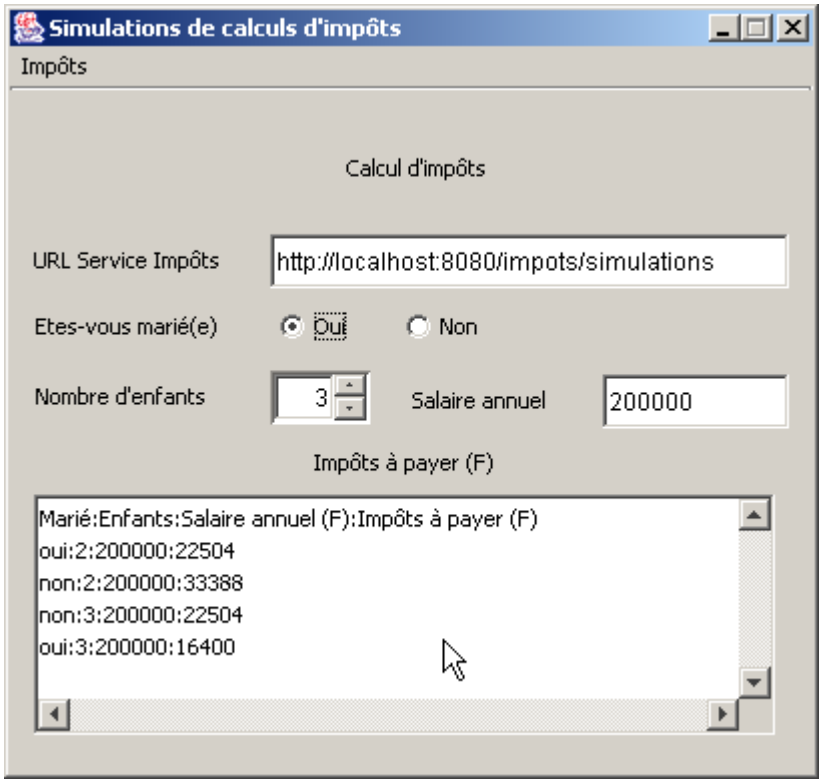



- toute erreur de connexion au serveur est signalée (dans l'exemple 1 ci-dessous, le port est incorrect - dans l'exemple 2, l'URL demandée n'existe pas - dans l'exemple 3 la base de données MySQL n'était pas lancé)




- Si tout est correct, les simulations sont affichées




Lorsqu'on écrit un client web programmé, il est nécessaire de savoir exactement ce qu'envoie le serveur en réponse aux différentes demandes possibles d'un client. Le serveur envoie un ensemble de lignes HTML comprenant des informations utiles et d'autres qui ne sont là que pour la mise en page HTML. Les expressions régulières de Java peuvent nous aider à trouver les informations utiles dans le flot des lignes envoyées par le serveur. Pour cela, nous avons besoin de connaître le format exact des différentes réponses du serveur. Ici nous allons utiliser le client web déjà rencontré qui permet d'afficher à l'écran la réponse d'un serveur à la demande d'une URL. L'URL demandée sera celle de notre service de simulations de calculs d'impôts *http://localhost:8080/impots/simulations* à laquelle nous pourrons passer des paramètres sous la forme *http://localhost:8080/impots/simulations?param1=vam1&param2=val2&...*

Demandons l'URL alors que la base MySQL permettant de créer on objet de type *impots* n'est pas lancée :

```
Dos>java clientweb http://localhost:8080/impots/simulations GET
```

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Fri, 16 Aug 2002 16:31:04 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=9DEC8B27966A1FBE3D4968A7B9DF3331;Path=/impots
```

```
<!-- dÚbut de la page HTML -->
<html>
  <head>
```

```
        <title>impots</title>
  </head>
  <body>
        <h3>calcul d'impôts</h3>
        <hr>
    Application indisponible([TCX][MyODBC]Can't connect to MySQL server on 'localhost' (10061))
  </body>
</html>
```

Pour récupérer l'erreur, le client web devra chercher dans la réponse du serveur web la ligne ayant le texte "*Application indisponible*". Maintenant lançons la base MySQL et demandons la même URL en lui passant les valeurs qu'elle attend (elle les attend indifféremment sous la forme d'un GET ou d'un POST - cf code java du serveur) :

```
Dos>java clientweb "http://localhost:8080/impots/simulations?ptMarie=oui&txtEnfants=2&txtSalaire=200000"
GET
```

```
HTTP/1.1 200 OK
Content-Type: text/html;charset=ISO-8859-1
Date: Fri, 16 Aug 2002 16:42:36 GMT
Connection: close
Server: Apache Tomcat/4.0.3 (HTTP/1.1 Connector)
Set-Cookie: JSESSIONID=C2A707600E98A37A343611D80DD5C8A2;Path=/impots


<html>

        <head>
        <title>impots</title>
    <script language="JavaScript" type="text/javascript">
        function effacer(){
.................................................
      }//effacer

      function calculer(){
.................................................
      }//calculer
                </script>
  </head>

  <body background="/impots/images/standard.jpg">
        <center>
        Calcul d'imp¶ts
        <hr>
      <form name="frmImpots" action="/impots/simulations" method="POST">
.................................................
      </form>
    </center>
    <hr>
    <h3>Résultats des simulations<h3>
<table "border="1">
<tr><td>Marié</td><td>Enfants</td><td>Salaire annuel (F)</td><td>Impôts à payer (F)</td></tr>
<tr><td>oui</td><td>2</td><td>200000</td><td>22504</td></tr>
</table>

 </body>
</html>
```

On retrouve ici la totalité du document HTML envoyé par le serveur. On trouve le résultat des différentes simulations dans l'unique table présente dans ce document. L'expression régulière nous permettant de récupérer les informations pertinentes du document pourra être la suivante :

```
"<tr>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*</tr>"
```

où les quatre expressions parenthésées représentent les quatre informations à récupérer.

Nous avons maintenant les lignes directrices de ce qu'il faut faire lorsque l'utilisateur demande le calcul de l'impôt à partir de l'interface graphique précédente :

- vérifier que toutes les données de l'interface sont valides et éventuellement signaler les erreurs.
- se connecter à l'URL indiquée dans le champ 1. Pour cela on suivra le modèle du client web générique déjà présenté et étudié
- dans le flux de la réponse du serveur, utiliser des expressions régulières pour soit :
    - trouver le message d'erreur s'il y en a un
    - trouver les résultats des simulations s'il n'y a pas d'erreur

Le code lié au menu *calculer* est le suivant :

```
  void mnuCalculer_actionPerformed(ActionEvent e) {
    // calcul de l'impôt
    // vérification URL service
    URL urlImpots=null;
    try{
      urlImpots=new URL(txtURLServiceImpots.getText().trim());
      String query=urlImpots.getQuery();
      if(query!=null) throw new Exception();
    }catch (Exception ex){
      // msg d'erreur
      JOptionPane.showMessageDialog(this,"URL incorrecte.
Recommencez","Erreur",JOptionPane.ERROR_MESSAGE);
      // focus sur champ erroné
      txtURLServiceImpots.requestFocus();
      // retour à l'interface
      return;
    }
    // vérification salaire
    int salaire=0;
    try{
      salaire=Integer.parseInt(txtSalaire.getText().trim());
      if(salaire<0) throw new Exception();
    }catch (Exception ex){
      // msg d'erreur
      JOptionPane.showMessageDialog(this,"Salaire incorrect.
Recommencez","Erreur",JOptionPane.ERROR_MESSAGE);
      // focus sur champ erroné
      txtSalaire.requestFocus();
      // retour à l'interface
      return;
    }
    // nbre d'enfants
    Integer nbEnfants=(Integer)spinEnfants.getValue();

    try{
      // on calcule l'impôt
      calculerImpots(urlImpots,rdOui.isSelected(),nbEnfants.intValue(),salaire);
    }catch (Exception ex){
      // on affiche l'erreur
      JOptionPane.showMessageDialog(this,"L'erreur suivante s'est produite : " +
ex.getMessage(),"Erreur",JOptionPane.ERROR_MESSAGE);
    }
  }//mnuCalculer


   public void calculerImpots(URL urlImpots,boolean marié, int nbEnfants, int salaire)
         throws Exception{
    // calcul de l'impôt
    // urlImpots : URL du service des impôts
    // marié : true si marié, false sinon
    // nbEnfants : nombre d'enfants
    // salaire : salaire annuel

    // on retire d'urlImpots les infos nécessaire à la connexion au serveur d'impôts
    String path=urlImpots.getPath();
    if(path.equals("")) path="/";
    String query="?"+"optMarie="+(marié ? "oui":"non")+"&txtEnfants="+nbEnfants+"&txtSalaire="+salaire;
    String host=urlImpots.getHost();
    int port=urlImpots.getPort();
    if(port==-1) port=urlImpots.getDefaultPort();

    // données locales
    Socket  client=null;          // le client
    BufferedReader IN=null;         // le flux de lecture du client
    PrintWriter OUT=null;         // le flux d'écriture du client
    String réponse=null;          // réponse du serveur
    // le modèle recherché dans les entêtes HTTP
    Pattern modèleCookie=Pattern.compile("^Set-Cookie: JSESSIONID=(.*?);");
    // le modèle d'une réponse correcte
    Pattern réponseOK=Pattern.compile("^.*? 200 OK");
    // le résultat de la comparaison au modèle
    Matcher résultat=null;

    try{
      // on se connecte au serveur
      client=new Socket(host,port);
```

```
      // on crée les flux d'entrée-sortie du client TCP
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      OUT=new PrintWriter(client.getOutputStream(),true);

      // on demande l'URL - envoi des entêtes HTTP
      OUT.println("GET " + path + query + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      if(! JSESSIONID.equals("")){
        OUT.println("Cookie: JSESSIONID="+JSESSIONID);
      }
      OUT.println("Connection: close");
      OUT.println("");
```

```
      // on lit la 1ère ligne de la réponse
      réponse=IN.readLine();
      // on compare la ligne HTTP au modèle de la réponse correcte
      résultat=réponseOK.matcher(réponse);
      if(! résultat.find()){
        // on a un problème d'URL
        throw new Exception("Le serveur a répondu : URL ["+ txtURLServiceImpots.getText().trim() + "]
inconnue");
      }//if(résultat)

      // on lit la réponse jusqu'à la fin des entêtes en cherchant l'éventuel cookie
      while((réponse=IN.readLine())!=null){
        // ligne vide ?
        if(réponse.equals("")) break;
        // ligne HTTP non vide
        // si on n'a pas le jeton de la session on le cherche
        if (JSESSIONID.equals("")){
          // on compare la ligne HTTP au modèle du cookie
          résultat=modèleCookie.matcher(réponse);
          if(résultat.find()){
            // on a trouvé le cookie du jeton
            JSESSIONID=résultat.group(1);
          }//if(résultat)
        }//if(JSESSIONID)
      }//while
```

```
      // c'est fini pour les entêtes HTTP - on passe au code HTML
      // pour récupérer les simulations
      ArrayList listeSimulations=getSimulations(IN,OUT,simulations);
      simulations.clear();
      for (int i=0;i<listeSimulations.size();i++){
        simulations.addElement(listeSimulations.get(i));
      }

      // c'est fini
      client.close();
    }catch (Exception ex){
      throw new Exception(ex.getMessage());
    }
  }//calculerImpots
```

```
   private ArrayList getSimulations(BufferedReader IN, PrintWriter OUT, DefaultListModel simulations)
throws Exception{
```

```
    // le modèle d'une ligne du tableau des simulations
    Pattern
ptnSimulation=Pattern.compile("<tr>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td
>\\s*</tr>");
    // le modèle d'une ligne de la liste des erreurs
    Pattern ptnErreur=Pattern.compile("(Application indisponible.*?)\\s*$");
```

```
    // le résultat de la comparaison au modèle
    Matcher résultat=null;
    // les simulations
    ArrayList listeSimulations=new ArrayList();

    // on lit toutes les lignes jusqu'à la fin
    String ligne=null;
    boolean simulationRéussie=false;
    while((ligne=IN.readLine())!=null){
      // suivi
      // on compare la ligne au modèle d'erreur si la partie simulations n'a pas encore été rencontrée
      if(! simulationRéussie){
        résultat=ptnErreur.matcher(ligne);
        if(résultat.find()){
          // msg d'erreur
          JOptionPane.showMessageDialog(this,résultat.group(1),"Erreur",JOptionPane.ERROR_MESSAGE);
          // c'est fini
          return listeSimulations;
        }//if
```

```
      }//if
      // on compare la ligne au modèle de simulation
      résultat=ptnSimulation.matcher(ligne);
      if(résultat.find()){
        // on a trouvé une ligne de la table
        listeSimulations.add(résultat.group(1)+":"+résultat.group(2)+":"+résultat.group(3)+
                    ":"+résultat.group(4));
        // la simulation a été réussie
        simulationRéussie=true;
      }//if
    }//while
    // fin
    return listeSimulations;
  }
```

Explicitons un peu ce code :

- la procédure **mnuCalculer_actionPerformed** vérifie que les données de l'interface sont valides. Si elles ne le sont pas un message d'erreur est émis et la procédure terminée. Si elles le sont, la procédure **calculerImpots** est exécutée.
- la procédure **calculerImpots** commence par construire l'URL qu'elle doit demander

```
    // on retire d'urlImpots les infos nécessaire à la connexion au serveur d'impôts
    String path=urlImpots.getPath();
    if(path.equals("")) path="/";
    String query="?"+"optMarie="+(marié ? "oui":"non")+"&txtEnfants="+nbEnfants+"&txtSalaire="+salaire;
    String host=urlImpots.getHost();
    int port=urlImpots.getPort();
    if(port==-1) port=urlImpots.getDefaultPort();
.....................
```

- puis se connecte à cette URL en envoyant les entêtes HTTP adéquats :

```
      // on se connecte au serveur
      client=new Socket(host,port);

      // on crée les flux d'entrée-sortie du client TCP
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      OUT=new PrintWriter(client.getOutputStream(),true);

      // on demande l'URL - envoi des entêtes HTTP
      OUT.println("GET " + path + query + " HTTP/1.1");
      OUT.println("Host: " + host + ":" + port);
      if(! JSESSIONID.equals("")){
        OUT.println("Cookie: JSESSIONID="+JSESSIONID);
      }
      OUT.println("Connection: close");
      OUT.println("");
```

- une fois la requête faite, notre client web attend la réponse. Il obtient d'abord les entêtes HTTP de la réponse du serveur. Il explore ces entêtes pour y trouver le jeton de la session. En effet, il devra renvoyer celui-ci au serveur pour que ce dernier puisse garder trace des différentes simulations faites. On notera ici qu'il aurait été plus simple que le client mémorise lui-mêmes les différentes simulations opérées par l'utilisateur. Néanmoins, nous gardons l'idée de renvoyer le jeton pour fournir un nouvel exemple de gestion de session. La première ligne de la réponse est traitée à part. En effet elle doit être de la forme *HTTP/version 200 OK* pour dire que l'URL demandée existe bien. Si donc elle n'est pas de cette forme on en conclut que l'utilisateur a demandé une URL incorrecte et on le lui dit.

```
    // le modèle recherché dans les entêtes HTTP
    Pattern modèleCookie=Pattern.compile("^Set-Cookie: JSESSIONID=(.*?);");
    // le modèle d'une réponse correcte
    Pattern réponseOK=Pattern.compile("^.*? 200 OK");
..........

      // on lit la 1ère ligne de la réponse
      réponse=IN.readLine();
      // on compare la ligne HTTP au modèle de la réponse correcte
      résultat=réponseOK.matcher(réponse);
      if(! résultat.find()){
        // on a un problème d'URL
        throw new Exception("Le serveur a répondu : URL ["+ txtURLServiceImpots.getText().trim() + "]
inconnue");
      }//if(résultat)

      // on lit la réponse jusqu'à la fin des entêtes en cherchant l'éventuel cookie
      while((réponse=IN.readLine())!=null){
        // ligne vide ?
        if(réponse.equals("")) break;
```

```
      // ligne HTTP non vide
      // si on n'a pas le jeton de la session on le cherche
      if (JSESSIONID.equals("")){
        // on compare la ligne HTTP au modèle du cookie
        résultat=modèleCookie.matcher(réponse);
        if(résultat.find()){
          // on a trouvé le cookie du jeton
          JSESSIONID=résultat.group(1);
        }//if(résultat)
      }//if(JSESSIONID)
    }//while
```

- une fois les entêtes HTTP exploités, on passe à la partie HTML de la réponse

```
    // c'est fini pour les entêtes HTTP - on passe au code HTML
    // pour récupérer les simulations
    ArrayList listeSimulations=getSimulations(IN,OUT,simulations);
    simulations.clear();
    for (int i=0;i<listeSimulations.size();i++){
      simulations.addElement(listeSimulations.get(i));
    }
```

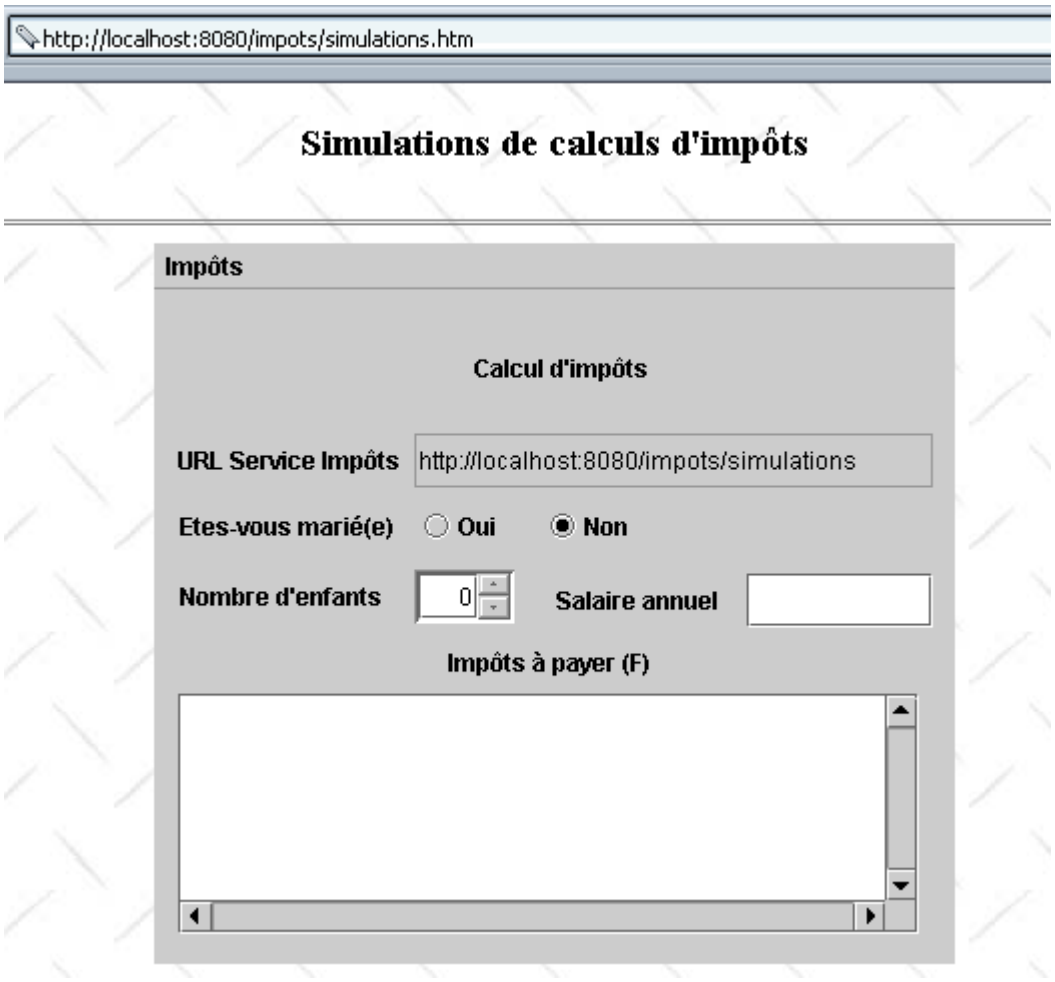
- la procédure **getSimulations** rend la liste des simulations si elles existent, liste qui sera vide si le serveur renvoie un message d'erreur. Dans ce cas, celui-ci est affiché dans une boîte de message. Si la liste est non vide, elle est affichée dans la liste déroulante de l'interface graphique.

- la procédure **getSimulations** va comparer chaque ligne de la réponse HTML à l'expression régulière représentant le message d'erreur (*Application indisponible...*) et à l'expression régulière représentant une simulation. Si le message d'erreur est rencontré, il est affiché et la procédure terminée. Si le résultat d'une simulation est trouvé, il est ajouté à la liste des simulations. A la fin de la procédure cette liste est rendue comme résultat.

```
  private ArrayList getSimulations(BufferedReader IN, PrintWriter OUT, DefaultListModel simulations)
throws Exception{

    // le modèle d'une ligne du tableau des simulations
    Pattern
ptnSimulation=Pattern.compile("<tr>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</t
d>\\s*</tr>");
    // le modèle d'une ligne de la liste des erreurs
    Pattern ptnErreur=Pattern.compile("(Application indisponible.*?)\\s*$");
```

## 4.6 Version 5

Ici, nous transformons l'application graphique autonome précédente en applet java. L'interface graphique est légèrement différente. Avec l'application autonome l'utilisateur donnait lui-même l'URL du service de simulation de calculs d'impôts et ensuite l'application se connectait à cette URL. Ici, l'application cliente est un navigateur et l'utilisateur demandera l'URL du document HTML contenant l'applet. Il faut se rappeler maintenant qu'une applet Java ne peut ouvrir une connexion réseau qu'avec le serveur d'où elle a été téléchargée. L'URL du service de simulation sera donc sur le même serveur que le document HTML contenant l'applet. Dans notre exemple, ce sera un paramètre d'initialisation de l'applet qui sera mis dans le champ *txtUrlServiceImpots*, champ qui sera non éditable par l'utilisateur. On aura ainsi le client suivant :




Le document HTML contenant l'applet s'appelle **simulations.htm** et est le suivant :

```
<html>
  <head>
    <title>Simulations de calculs d'impôts</title>
  </head>
  <body background="/impots/images/standard.jpg">
    <center>
      <h3>Simulations de calculs d'impôts</h3>
      <hr>
      <applet code="appletImpots.class" width="400" height="360">
        <param name="urlServiceImpots" value="simulations">
      </applet>
    </center>
  </body>
</html>
```

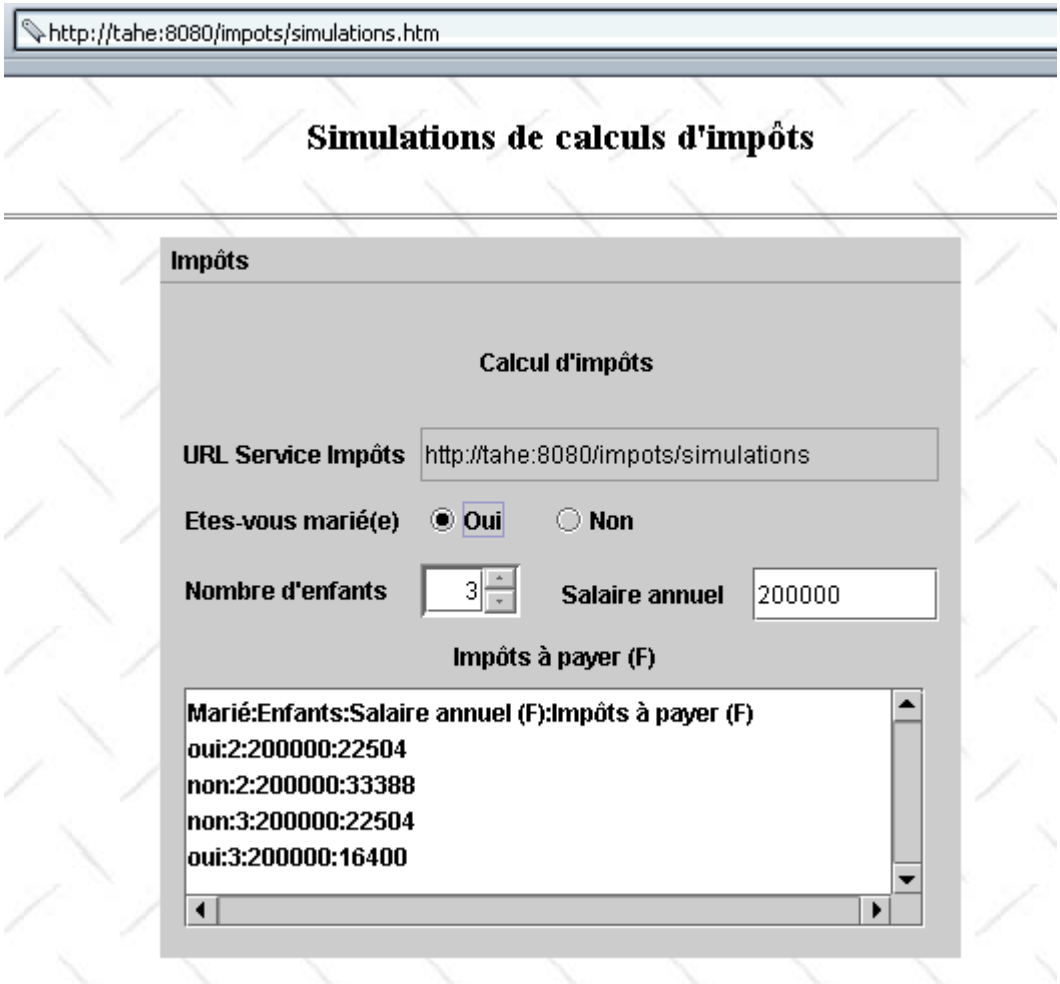
L'applet a un paramètre *urlServiceImpots* qui est l'URL du service de simulations du calcul d'impôts. Cette URL est relative et mesurée par rapport à l'URL du document HTML *simulations.htm*. Ainsi si un navigateur obtient ce document avec l'URL *http://localhost:8080/impots/simulations.htm*, l'URL du service de simulation sera *http://localhost:8080/impots/simulations*. Si cette URL avait été *http://stahe:8080/impots/simulations.htm* l'URL du service de simulation aurait été *http://stahe:8080/impots/simulations*.

L'applet **appletImpots.java** reprend intégralement le code de l'application graphique autonome précédente en respectant les règles de transformation d'une application graphique en applet.

```
public class appletImpots extends JApplet {
  // les composants de la fenêtre
  JPanel contentPane;
  JMenuBar jMenuBar1 = new JMenuBar();
  JMenu jMenu1 = new JMenu();
  JMenuItem mnuCalculer = new JMenuItem();
.............
```

```
  //Construire le cadre
  public void init() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    // autres initialisations
    moreInit();
  }

  // initialisation formulaire
  private void moreInit(){
    // on récupère le paramètre urlServiceImpots
    String urlServiceImpots=getParameter("urlServiceImpots");
    if(urlServiceImpots==null){
      // paramètre manquant
      JOptionPane.showMessageDialog(this,"Le paramètre urlServiceImpots de l'applet n'a pas été
défini","Erreur",JOptionPane.ERROR_MESSAGE);
      // fin
      return;
    }
    // on met l'URL dans son champ
    String codeBase=""+getCodeBase();
    if(codeBase.endsWith("/"))
       txtURLServiceImpots.setText(codeBase+urlServiceImpots);
     else txtURLServiceImpots.setText(codeBase+"/"+urlServiceImpots);
    // menu Calculer inhibé
    mnuCalculer.setEnabled(false);
    // spinner Enfants - entre 0 et 20 enfants
    spinEnfants=new JSpinner(new SpinnerNumberModel(0,0,20,1));
    spinEnfants.setBounds(new Rectangle(130,140,50,27));
    contentPane.add(spinEnfants);
  }//moreInit

  //Initialiser le composant
  private void jbInit() throws Exception  {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(null);
.............
  }
```

Lorsque un navigateur charge une applet, il exécute tout d'abord sa procédure **init**. Dans celle-ci, nous avons récupéré la valeur du paramètre *urlServiceImpots* de l'applet puis calculé le nom complet de l'URL du service de simulation et placé cette valeur dans le champ *txtURLServiceImpots* comme si c'était l'utilisateur qui l'avait tapé. Ceci fait, il n'y a plus de différence entre les deux applications. Notamment, le code associé au menu *Calculer* est identique. Voici un exemple d'exécution :




## 4.7 Conclusion

Nous avons montré différentes versions de notre application client-serveur de calcul d'impôts :

- **version 1** : le service est assuré par un ensemble de servlets et pages JSP, le client est un navigateur. Il fait une seule simulation et n'a pas de mémoire des précédentes.
- **version 2** : on ajoute quelques capacités côté navigateur en mettant des scripts javascript dans le document HTML chargé par celui-ci. Il contrôle la validité des paramètres du formulaire.
- **version 3** : on permet au service de se souvenir des différentes simulations opérées par un client en gérant une session. L'interface HTML est modifiée en conséquence pour afficher celles-ci.
- **version 4** : le client est désormais une application graphique autonome. Cela nous permet de revenir sur l'écriture de clients web programmés.
- **version 5** : le client se transforme en applet java. On a alors une application client-serveur entièrement programmée en Java que ce soit côté serveur ou côté client.

A ce point, on peut faire quelques remarques :

- les versions 1 à 3 autorisent des navigateurs sans capacité autre que celle de pouvoir exécuter des scripts javascript. On notera qu'un utilisateur a toujours la possibilité d'inhiber l'exécution de ces derniers. L'application ne fonctionnera alors que partiellement dans sa version 1 (option Effacer ne fonctionnera pas) et pas du tout dans ses versions 2 et 3 (options Effacer et Calculer ne fonctionneront pas). Il pourrait être intéressant de prévoir une version du service n'utilisant pas de scripts javascript.
- la version 4 nécessite que le poste client ait une machine virtuelle Java 2.
- la version 5 nécessite que le poste client ait un navigateur ayant une machine virtuelle Java 2.

Lorsqu'on écrit un service Web, il faut se demander quels types de clients on vise. Si on veut viser le plus grand nombre de clients, on écrira une application qui n'envoie aux navigateurs que du HTML (pas de javascript ni d'applet). Si on travaille au sein d'un intranet et qu'on maîtrise la configuration des postes de celui-ci on peut alors se permettre d'être plus exigeant au niveau du client et la version 5 précédente peut alors être acceptable.

Les versions 4 et 5 sont des clients web qui retrouvent l'information dont ils ont besoin au sein du flux HTML envoyé par le serveur. Très souvent on ne maîtrise pas ce flux. C'est le cas lorsqu'on a écrit un client pour un service web existant sur le réseau et géré par quelqu'un d'autre. Prenon un exemple. Supposons que notre service de simulations de calcul d'impôts ait été écrit par une société X. Actuellement le service envoie les simulations dans un tableau HTML et notre client exploite ce fait pour les récupérer. Il compare ainsi chaque ligne de la réponse du serveur à l'expression régulière :

```
    // le modèle d'une ligne du tableau des simulations
    Pattern
ptnSimulation=Pattern.compile("<tr>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</td>\\s*<td>(.*?)</t
d>\\s*</tr>");
```

Supposons maintenant que le concepteur de l'application change l'apparence visuelle de la réponse en mettant les simulations non pas dans un tableau mais dans une liste sous la forme :

```
Résultat des simulations
<ul>
  <li>oui,2,200000,22504
  <li>non,2,200000,33388
</ul>
```

Dans ce cas, notre client web devra être réécrit. C'est là la menace permanente pesant sur les clients web d'applications qu'on ne maîtrise pas soi-même. XML peut apporter une solution à ce problème :

- au lieu de générer du HTML, le service de simulations va générer du XML. Dans notre exemple, cela pourrait être

```
<simulations>
  <entetes marie="marié" enfants="enfants" salaire="salaire" impot="impôt"/>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504" />
  <simulation marie="non" enfants="2" salaire="200000" impot="33388" />
</simulations>
```

- une feuille de style pourrait être associée à cette réponse indiquant aux navigateurs la forme visuelle à donner à cette réponse XML
- les clients web programmés ignoreraient cette feuille de style et récuperaient l'information directement dans le flux XML de la réponse

Si le concepteur du service souhaite modifier la présentation visuelle des résultats qu'il fournit, il modifiera la feuille de style et non le XML. Grâce à la feuille de style, les navigateurs afficheront la nouvelle forme visuelle et les clients web programmés n'auront pas eux à être modifiés. De nouvelles version de notre service de simulations pourraient donc être écrites :

- **version 6** : le service fournit une réponse XML accompagnée d'une feuille de style à destination des navigateurs
- **version 7** : le client est une application graphique autonome exploitant la réponse XML du serveur
- **version 8** : le client est une applet java exploitant la réponse XML du serveur

# 5. XML et JAVA

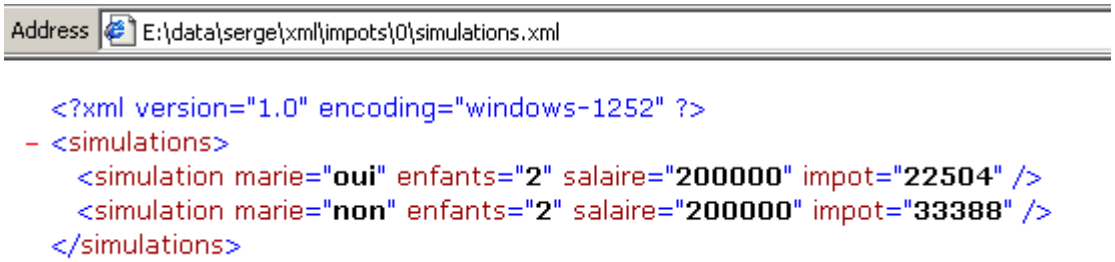
Dans ce chapitre, nous introduisons l'utilisation de documents XML avec Java. Nous le ferons dans le contexte de l'application **impôts** étudiée dans le chapitre précédent.

## 5.1 Fichiers XML et feuilles de style XSL

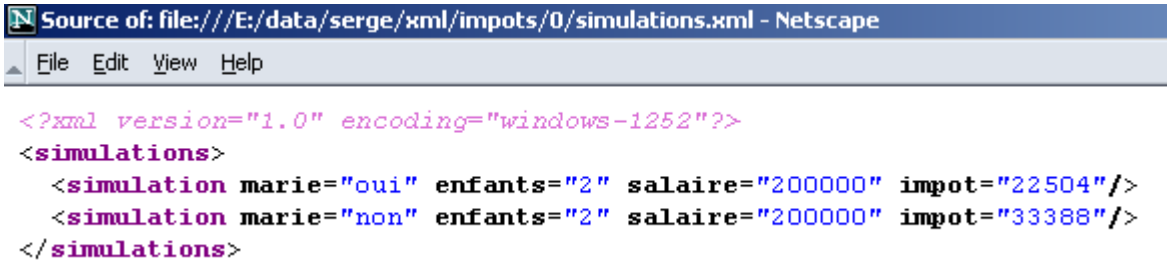
Considérons le fichier XML **simulations.xml** suivant qui pourrait représenter le résultat de simulations de calculs d'impôts :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

Si on le visualise avec IE 6, on obtient le résultat suivant :




IE6 reconnaît qu'il a affaire à un fichier XML (grâce au suffixe .xml du fichier) et le met en page d'une façon qui lui est propre. Avec Netscape on obtient une page vide. Cependant si on regarde le code source (View/Source) on a bien le fichier XML d'origine :




Pourquoi Netscape n'affiche-t-il rien ? Parce qu'il lui faut une feuille de style qui lui dise comment transformer le fichier XML en fichier HTML qu'il pourra alors afficher. Il se trouve que IE 6 a lui une feuille de style par défaut lorsque le fichier XML n'en propose pas ce qui était le cas ici.

Il existe un langage appelé **XSL** (eXtended StyleSheet Language) permettant de décrire les transformations à effectuer pour passer un fichier XML en un fichier texte quelconque. XSL permet l'utilisation de nombreuses instructions et ressemble fort aux langages de programmation. Nous le détaillerons pas ici car il y faudrait plusieurs dizaines de pages. Nous allons simplement décrire deux exemples de feuilles de style XSL. La première est celle qui va transformer le fichier XML **simulations.xml** en code HTML. On modifie ce dernier afin qu'il désigne la feuille de style que pourront utiliser les navigateurs pour le transformer en document HTML, document qu'ils pourront afficher :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

La commande XML

```
<?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
```

désigne le fichier **simulations.xsl** comme un feuille de style (**xml-stylesheet**) de type **text/xsl** c.a.d. un fichier texte contenant du code XSL. Cette feuille de style sera utilisée par les navigateurs pour transformer le texte XML en document HTML. Voici le résultat obtenu avec Netscape 7 lorsqu'on charge le fichier XML **simulations.xml** :


**Simulations de calculs d'impôts**

| marié | enfants | salaire | impôt |
|---|---|---|---|
| oui | 2 | 200000 | 22504 |
| non | 2 | 200000 | 33388 |

Lorsque nous regardons le code source du document (View/Source) nous retrouvons le document XML initial et non le document HTML affiché :


```
<?xml version="1.0" encoding="windows-1252"?>
<?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

Netscape a utilisé la feuille de style **simulations.xsl** pour transformer le document XML ci-dessus en document HTML affichable. Il est maintenant temps de regarder le contenu de cette feuille de style :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" indent="yes"/>

    <xsl:template match="/">
      <html>
        <head>
          <title>Simulations de calculs d'impôts</title>
         </head>
        <body>
          <center>
            <h3>Simulations de calculs d'impôts</h3>
              <hr/>
              <table border="1">
                <th>marié</th><th>enfants</th><th>salaire</th><th>impôt</th>
                <xsl:apply-templates select="/simulations/simulation"/>
              </table>
          </center>
        </body>
      </html>
    </xsl:template>


    <xsl:template match="simulation">
       <tr>
         <td><xsl:value-of select="@marie"/></td>
         <td><xsl:value-of select="@enfants"/></td>
         <td><xsl:value-of select="@salaire"/></td>
         <td><xsl:value-of select="@impot"/></td>
        </tr>
    </xsl:template>
```

```
  </xsl:stylesheet>
```

- une feuille de style XSL est un fichier XML et en suit donc les règles. Il doit être en autres choses "bien formé" c'est à dire que toute balise ouverte doit être fermée.
- le fichier commence par deux commandes XML qu'on pourra garder dans toute feuille de style XSL :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

L'attribut *encoding="ISO-8859-1"* permet d'utiliser les caractères accentués dans la feuille de style.

- La balise `<xsl:output method="html" indent="yes"/>` indique à l'interpréteur XSL qu'on veut produire du HTML "indenté".
- La balise `<xsl:template match="élément">` sert à définir l'élément du document XML sur lequel vont s'appliquer les instructions que l'on va trouver entre `<xsl:template ...>` et `</xsl:template>`.

```
    <xsl:template match="/">
.................
    </xsl:template>
```

Dans l'exemple ci-dessus l'élément `"/"` désigne la racine du document. Cela signifie que dès que le début du document XML va être rencontré, les commandes XSL situées entre les deux balises vont être exécutées.

- Tout ce qui n'est pas balise XSL est mis tel quel dans le flux de sortie. Les balises XSL elles sont exécutées. Certaines d'entre-elles produisent un résultat qui est mis dans le flux de sortie. Etudions l'exemple suivant :

```
    <xsl:template match="/">
      <html>
        <head>
          <title>Simulations de calculs d'impôts</title>
         </head>
        <body>
          <center>
            <h3>Simulations de calculs d'impôts</h3>
              <hr/>
              <table border="1">
                <th>marié</th><th>enfants</th><th>salaire</th><th>impôt</th>
                <xsl:apply-templates select="/simulations/simulation"/>
              </table>
           </center>
         </body>
       </html>
    </xsl:template>
```

Rappelons que le document XML analysé est le suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

Dès le début du document XML analysé (`match="/"`), l'interpréteur XSL va produire en sortie le texte

```
<html>
       <head>
         <title>Simulations de calculs d'impôts</title>
        </head>
       <body>
         <center>
           <h3>Simulations de calculs d'impôts</h3>
             <hr>
             <table border="1">
               <th>marié</th><th>enfants</th><th>salaire</th><th>impôt</th>
```

On remarquera que dans le texte initial on avait `<hr/>` et non pas `<hr>`. Dans le texte initial on ne pouvait pas écrire `<hr>` qui si elle est une balise HTML valide est une balise XML invalide. Or nous avons affaire ici à un texte XML qui doit être "bien formé", c.a.d que toute balise doit être fermée. On écrit donc `<hr/>` et parce qu'on a écrit `<xsl:output text="html ...>` l'interpréteur XSL transformera le texte `<hr/>` en `<hr>`. Derrière ce texte, viendra ensuite le texte produit par la commande XSL :

```
       <xsl:apply-templates select="/simulations/simulation"/>
```

Nous verrons ultérieurement quel est ce texte. Enfin l'interpréteur ajoutera le texte :

```
          </table>
         </center>
       </body>
     </html>
```

La commande <xsl:apply-templates select="/simulations/simulation"/> demande qu'on exécute le "template" (modèle) de l'élément */simulations/simulation*. Elle sera exécutée à chaque fois que l'interpréteur XSL rencontrera dans le texte XML analysé une balise <simulation>..</simulations> ou <simulation/> à l'intérieur d'une balise <simulations>..</simulations>. A la rencontre de la balise <*simulation*>, l'interpréteur exécutera les instructions du modèle suivant :

```
    <xsl:template match="simulation">
       <tr>
         <td><xsl:value-of select="@marie"/></td>
         <td><xsl:value-of select="@enfants"/></td>
         <td><xsl:value-of select="@salaire"/></td>
         <td><xsl:value-of select="@impot"/></td>
        </tr>
    </xsl:template>
```

Considérons les lignes XML suivantes :

```
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
```

La ligne <*simulation* ..> correspond au modèle de l'instruction XSL <*xsl:apply-templates select="/simulations/simulation*>". L'interpréteur XSL va donc chercher à lui appliquer les instructions qui correspondent à ce modèle. Il va trouver le modèle <*xsl:template match="simulation"*> et va l'exécuter. Rappelons que ce qui n'est pas une commande XSL est repris tel quel par l'interpréteur XSL et que les commandes XSL sont elles remplacées par le résultat de leur exécution. L'instruction XSL <*xsl:value-of select="@champ"*/> est ainsi remplacé par la valeur de l'attribut "champ" du noeud analysé (ici un noeud <*simulation*>). L'analyse de la ligne XML précédente va produire en sortie le résultat suivant :

| XSL | sortie |
|---|---|
| <tr><td> | <tr><td> |
| <xsl:value-of select="@marie"/> | oui |
| </td><td> | </td><td> |
| <xsl:value-of select="@enfants"/> | 2 |
| </td><td> | </td><td> |
| <xsl:value-of select="@salaire"/> | 200000 |
| </td><td> | </td><td> |
| <xsl:value-of select="@impot"/> | 22504 |
| </td></tr> | </td></tr> |

Au total, la ligne XML

```
<simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
```

va être transformée en ligne HTML :

```
<tr><td>oui</td><td>2</td><td>200000</td><td>22504</td></tr>
```

Toutes ces explications sont un peu rudimentaires mais il devrait apparaître maintenant au lecteur que le texte XML suivant :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

accompagné de la feuille de style XSL **simulations.xsl** suivante :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" indent="yes"/>

    <xsl:template match="/">
     <html>
       <head>
```

```
        <title>Simulations de calculs d'impôts</title>
       </head>
      <body>
        <center>
          <h3>Simulations de calculs d'impôts</h3>
           <hr/>
           <table border="1">
             <th>marié</th><th>enfants</th><th>salaire</th><th>impôt</th>
             <xsl:apply-templates select="/simulations/simulation"/>
           </table>
        </center>
      </body>
    </html>
  </xsl:template>


   <xsl:template match="simulation">
      <tr>
        <td><xsl:value-of select="@marie"/></td>
        <td><xsl:value-of select="@enfants"/></td>
        <td><xsl:value-of select="@salaire"/></td>
        <td><xsl:value-of select="@impot"/></td>
       </tr>
  </xsl:template>
</xsl:stylesheet>
```

produit le texte HTML suivant :

```
<html>
<head>
<title>Simulations de calculs d'impôts</title>
</head>
<body>
<center>
<h3>Simulations de calculs d'impots</h3>
<hr>
<table border="1">
<th>marié</th><th>enfants</th><th>salaire</th><th>impôt</th>
<tr>
<td>oui</td><td>2</td><td>200000</td><td>22504</td>
</tr>
<tr>
<td>non</td><td>2</td><td>200000</td><td>33388</td>
</tr>
</table>
</center>
</body>
</html>
```

Le fichier XML **simulations.xml** accompagné de la feuille de style **simulations.xsl** lu par un navigateur récent (ici Netscape 7) est alors affiché comme suit :


**Simulations de calculs d'impôts**

| marié | enfants | salaire | impôt |
|---|---|---|---|
| oui | 2 | 200000 | 22504 |
| non | 2 | 200000 | 33388 |

## 5.2 Application impôts : version 6

### 5.2.1 Les fichiers XML et feuilles de style XSL de l'application impôts

Revenons à l'application web **impôts** et modifions la afin que la réponse faite aux clients soit une réponse au format XML plutôt qu'une réponse HTML. Cette réponse XML sera accompagnée d'une feuille de style XSL afin que les navigateurs puissent l'afficher. Dans le paragraphe précédent, nous avons présenté :

- le fichier **simulations.xml** qui est le prototype d'une réponse XML comportant des simulations de calculs d'impôts
- le fichier **simulations.xsl** qui sera la feuille de style XSL qui accompagnera cette réponse XML

Il nous faut prévoir également le cas de la réponse avec des erreurs. Le prototype de la réponse XML dans ce cas sera le fichier **erreurs.xml** suivant :

```
<?xml version="1.0" encoding="windows-1252"?>
<?xml-stylesheet type="text/xsl" href="erreurs.xsl"?>
<erreurs>
  <erreur>erreur 1</erreur>
  <erreur>erreur 2</erreur>
</erreurs>
```

La feuille de style **erreurs.xsl** permettant d'afficher ce document XML dans un navigateur sera la suivante :

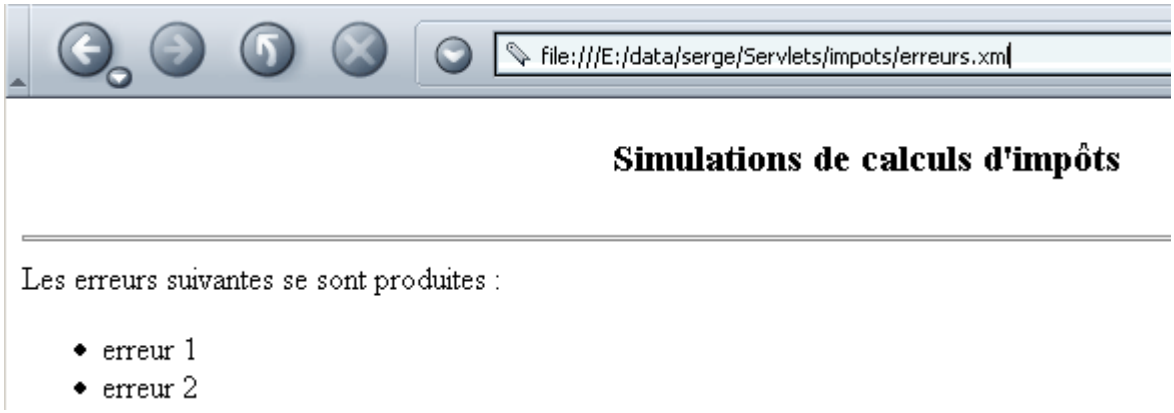
```
<?xml version="1.0" encoding="windows-1252"?>
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" indent="yes"/>

    <xsl:template match="/">
     <html>
       <head>
         <title>Simulations de calculs d'impôts</title>
        </head>
       <body>
         <center>
           <h3>Simulations de calculs d'impôts</h3>
          </center>
          <hr/>
          Les erreurs suivantes se sont produites :
          <ul>
           <xsl:apply-templates select="/erreurs/erreur"/>
         </ul>
        </body>
      </html>
   </xsl:template>

    <xsl:template match="erreur">
       <li><xsl:value-of select="."/></li>
   </xsl:template>
</xsl:stylesheet>
```

Cette feuille de style introduit une commande XSL non encore rencontrée : *<xsl:value-of select="."/>*. Cette commande produit en sortie la valeur du noeud analysé, ici un noeud *<erreur>texte</erreur>*. La valeur de ce noeud est le texte compris entre les deux balises d'ouverture et de fermeture, ici *texte*.

Le code **erreurs.xml** est transformé par la feuille de style **erreurs.xsl** en document HTML suivant :

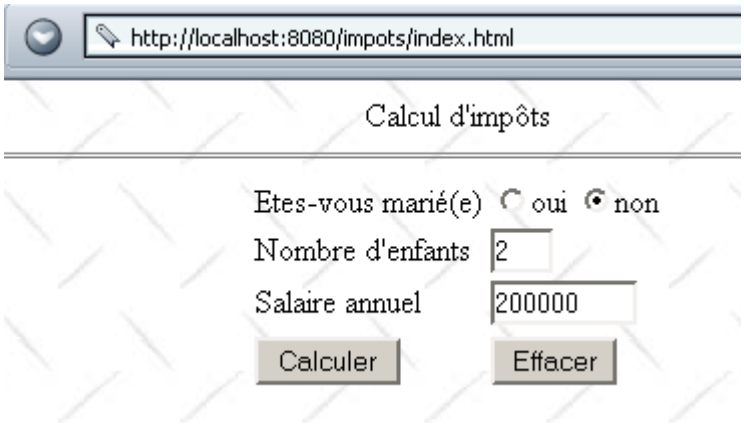
```
<html>
<head>
<title>Simulations de calculs d'impots</title>
</head>
<body>
<center>
<h3>Simulations de calculs d'impots</h3>
</center>
<hr>
          Les erreurs suivantes se sont produites :
<ul>
<li>erreur 1</li>
<li>erreur 2</li>
</ul>
</body>
</html>
```

Le fichier **erreurs.xml** accompagné de sa feuille de style est affiché par un navigateur de la façon suivante :




## 5.2.2 La servlet xmlsimulations

Nous créons un fichier **index.html** que nous mettons dans le répertoire de l'application **impots**. La page visualisée est la suivante :




Ce document HTML est un document **statique**. Son code est le suivant :

```
<html>
  <head>
    <title>impots</title>
    <script language="JavaScript" type="text/javascript">
     function effacer(){
       // raz du formulaire
        with(document.frmImpots){
         optMarie[0].checked=false;
          optMarie[1].checked=true;
          txtEnfants.value="";
          txtSalaire.value="";
          txtImpots.value="";
        }//with
      }//effacer

      function calculer(){
       // vérification des paramètres avant de les envoyer au serveur
        with(document.frmImpots){
          //nbre d'enfants
          champs=/^\s*(\d+)\s*$/.exec(txtEnfants.value);
          if(champs==null){
            // le modéle n'est pas vérifié
            alert("Le nombre d'enfants n'a pas été donné ou est incorrect");
            nbEnfants.focus();
            return;
          }//if
          //salaire
          champs=/^\s*(\d+)\s*$/.exec(txtSalaire.value);
          if(champs==null){
            // le modéle n'est pas vérifié
            alert("Le salaire n'a pas été donné ou est incorrect");
```

```
            salaire.focus();
            return;
          }//if
          // c'est bon - on envoie
          submit();
        }//with
      }//calculer
    </script>
  </head>

  <body background="/impots/images/standard.jpg">
    <center>
      Calcul d'impôts
      <hr>
      <form name="frmImpots" action="/impots/xmlsimulations" method="POST">
        <table>
          <tr>
            <td>Etes-vous marié(e)</td>
              <td>
               <input type="radio" name="optMarie" value="oui">oui
                 <input type="radio" name="optMarie" value="non" checked>non
              </td>
            </tr>
            <tr>
             <td>Nombre d'enfants</td>
              <td><input type="text" size="3" name="txtEnfants" value=""></td>
            </tr>
            <tr>
             <td>Salaire annuel</td>
              <td><input type="text" size="10" name="txtSalaire" value=""></td>
            </tr>
            <tr></tr>
            <tr>
             <td><input type="button" value="Calculer" onclick="calculer()"></td>
              <td><input type="button" value="Effacer" onclick="effacer()"></td>
            </tr>
          </table>
        </form>
      </center>
  </body>
</html>
```

On notera que les données du formulaire sont postées à l'URL **/impots/xmlsimulations**. Cette application est une servlet Java configurée de la façon suivante dans le fichier *web.xml* de l'application *impots* :

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

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
...........
  <servlet>
    <servlet-name>xmlsimulations</servlet-name>
    <servlet-class>xmlsimulations</servlet-class>
    <init-param>
    <param-name>xslSimulations</param-name>
      <param-value>simulations.xsl</param-value>
    </init-param>
    <init-param>
    <param-name>xslErreurs</param-name>
      <param-value>erreurs.xsl</param-value>
    </init-param>

    <init-param>
    <param-name>DSNimpots</param-name>
      <param-value>mysql-dbimpots</param-value>
    </init-param>
    <init-param>
    <param-name>admimpots</param-name>
      <param-value>admimpots</param-value>
    </init-param>
    <init-param>
    <param-name>mdpimpots</param-name>
      <param-value>mdpimpots</param-value>
    </init-param>
  </servlet>
........
  <servlet-mapping>
```

```
    <servlet-name>xmlsimulations</servlet-name>
     <url-pattern>/xmlsimulations</url-pattern>
  </servlet-mapping>
</web-app>
```

- la servlet s'appelle **xmlsimulations** et s'appuie sur la classe **xmlsimulations.class**.
- elle a pour paramètres les paramètres **DSNimpots**, **admimpots**, **mdpimpots** nécessaires pour accéder à la base de données des impôts. Par ailleurs, elle admet deux autres paramètres :
  - **xslSimulations** qui est le nom du fichier de style qui doit accompagner la réponse XML contenant les simulations
  - **xslErreurs** qui est le nom du fichier de style qui doit accompagner la réponse XML contenant les éventuelles erreurs
- elle a un alias **xmlsimulations** qui la rend accessible via l'URL *http://localhost:8080/impots/xmlsimulations*.

Le squelette de la servlet **xmlsimulations** est semblable à celui de la servlet **simulations** déjà étudiée. La principale différence vient du fait qu'elle doit générer du XML au lieu du HTML. Cela va entraîner la suppression des fichiers JSP utilisés dans les applications précédentes. Leur rôle principal était d'améliorer la lisibilité du code HTML généré en évitant que celui-ci soit noyé dans le code Java de la servlet. Ce rôle n'a maintenant plus lieu d'être. La servlet a deux types de code XML à générer :

- celui pour les simulations
- celui pour les erreurs

Nous avons précédemment présenté et étudié les deux types de réponse XML à fournir dans ces deux cas ainsi que les feuilles de style qui doivent les accompagner. Le code de la servlet est le suivant :

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

public class xmlsimulations extends HttpServlet{

  // variables d'instance
  String msgErreur=null;
  String xslSimulations=null;
  String xslErreurs=null;
  String DSNimpots=null;
  String admimpots=null;
  String mdpimpots=null;
  impotsJDBC impots=null;

  //-------- GET
  public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException{

    // on récupère le flux d'écriture vers le client
    PrintWriter out=response.getWriter();

    // on précise le type de la réponse
    response.setContentType("text/xml");

    // la liste des erreurs
    ArrayList erreurs=new ArrayList();

    // l'initialisation s'est-elle bien passée ?
    if(msgErreur!=null){
      // c'est fini - on envoie la réponse avec erreurs au serveur
      erreurs.add(msgErreur);
      sendErreurs(out,xslErreurs,erreurs);
      // c'est fini
      return;
    }

    // on récupère les simulations précédentes de la session
    HttpSession session=request.getSession();
    ArrayList simulations=(ArrayList)session.getAttribute("simulations");
    if(simulations==null) simulations=new ArrayList();

    // on récupère les paramètres de la requête courante
    String optMarie=request.getParameter("optMarie");              // statut marital
    String txtEnfants=request.getParameter("txtEnfants");       // nbre d'enfants
    String txtSalaire=request.getParameter("txtSalaire");       // salaire annuel

    // a-t-on tous les paramètres attendus
    if(optMarie==null || txtEnfants==null || txtSalaire==null){
      // il manque des paramètres
      // on envoie la réponse avec erreurs
      erreurs.add("Demande incomplète. Il manque des paramètres");
```

```
      sendErreurs(out,xslErreurs,erreurs);
      // c'est fini
      return;
    }

    // on a tous les paramètres - on les vérifie
    // état marital
    if( ! optMarie.equals("oui") && ! optMarie.equals("non")){
      // erreur
      erreurs.add("Etat marital incorrect");
    }
    // nombre d'enfants
    txtEnfants=txtEnfants.trim();
    if(! Pattern.matches("^\\d+$",txtEnfants)){
      // erreur
      erreurs.add("Nombre d'enfants incorrect");
    }
    // salaire
    txtSalaire=txtSalaire.trim();
    if(! Pattern.matches("^\\d+$",txtSalaire)){
      // erreur
      erreurs.add("Salaire incorrect");
    }

    if(erreurs.size()!=0){
      // s'il y a des erreurs, on les signale
      sendErreurs(out,xslErreurs,erreurs);
    }else{
      // pas d'erreurs
      try{
        // on peut calculer l'impôt à payer
        int nbEnfants=Integer.parseInt(txtEnfants);
        int salaire=Integer.parseInt(txtSalaire);
        String txtImpots=""+impots.calculer(optMarie.equals("oui"),nbEnfants,salaire);
        // on ajoute le résultat courant aux simulations précédentes
        String[] simulation={optMarie.equals("oui") ? "oui" : "non",txtEnfants, txtSalaire, txtImpots};
        simulations.add(simulation);
        // on envoie la réponse avec simulations
        sendSimulations(out,xslSimulations,simulations);
      }catch(Exception ex){}
    }//if-else
    // on remet la liste des simulations dans la session
    session.setAttribute("simulations",simulations);
  }//GET

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

```
  //-------- INIT
  public void init(){
   // on récupère les paramètres d'initialisation
   ServletConfig config=getServletConfig();
   xslSimulations=config.getInitParameter("xslSimulations");
   xslErreurs=config.getInitParameter("xslErreurs");
   DSNimpots=config.getInitParameter("DSNimpots");
   admimpots=config.getInitParameter("admimpots");
   mdpimpots=config.getInitParameter("mdpimpots");

   // paramètres ok ?
   if(xslSimulations==null || DSNimpots==null || admimpots==null || mdpimpots==null){
     msgErreur="Configuration incorrecte";
     return;
   }

   // on crée une instance d'impotsJDBC
   try{
     impots=new impotsJDBC(DSNimpots,admimpots,mdpimpots);
   }catch(Exception ex){
     msgErreur=ex.getMessage();
   }
  }//init
```

```
  //-------- sendErreurs
  private void sendErreurs(PrintWriter out,String xslErreurs,ArrayList erreurs){
   String réponse="<?xml version=\"1.0\" encoding=\"windows-1252\"?>"
         + "<?xml-stylesheet type=\"text/xsl\" href=\""+xslErreurs+"\"?>\n"
          +"<erreurs>\n";
   for(int i=0;i<erreurs.size();i++){
```

```
      réponse+="<erreur>"+(String)erreurs.get(i)+"</erreur>\n";
    }//for
    réponse+="</erreurs>\n";
    // on envoie la réponse
    out.println(réponse);
    }

    //-------- sendSimulations
    private void sendSimulations(PrintWriter out, String xslSimulations, ArrayList simulations){
     String réponse="<?xml version=\"1.0\" encoding=\"windows-1252\"?>"
          + "<?xml-stylesheet type=\"text/xsl\" href=\""+xslSimulations+"\"?>\n"
           + "<simulations>\n";
     String[] simulation=null;
     for(int i=0;i<simulations.size();i++){
       // simulation n° i
       simulation=(String[])simulations.get(i);
       réponse+="<simulation "
            +"marie=\""+(String)simulation[0]+"\" "
            +"enfants=\""+(String)simulation[1]+"\" "
            +"salaire=\""+(String)simulation[2]+"\" "
             +"impot=\""+(String)simulation[3]+"\" />\n";
     }//for
     réponse+="</simulations>\n";
     // on envoie la réponse
     out.println(réponse);
    }
}
```

Détaillons les principales nouveautés de ce code par rapport à ce que nous connaissions déjà :

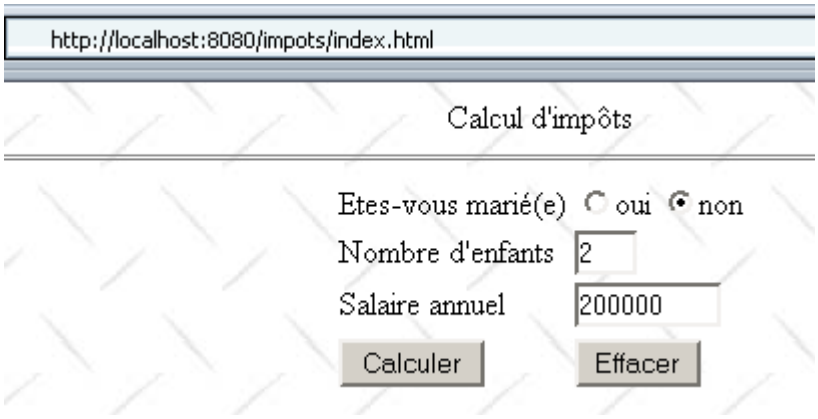
- la procédure **init** récupère de nouveaux paramètres dans le fichier de configuration **web.xml** : les noms des deux feuilles de style XSL qui doivent accompagner la réponse sont placés dans les variables **xslSimulations** et **xslErreurs**. Ces deux feuilles de style sont les fichiers **simulations.xsl** et **erreurs.xsl** étudiés précédemment. Ceux-ci sont placés dans le répertoire de l'application **impots** :

```
dos>dir E:\data\serge\Servlets\impots\*.xsl
27/08/2002  08:15               1 030 simulations.xsl
27/08/2002  09:23                 795 erreurs.xsl
```

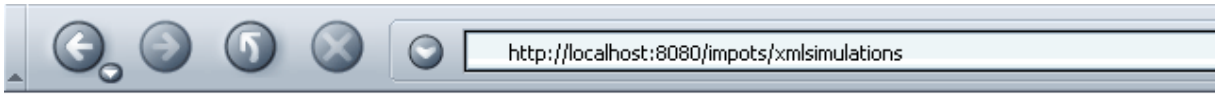
- la procédure GET commence par regarder s'il y a eu une erreur lors de l'initialisation. Si oui, elle appelle la procédure **sendErreurs** qui génère la réponse XML adaptée à ce cas puis s'arrête. Dans cette réponse XML est insérée l'instruction désignant la feuille de style à utiliser.
- s'il n'y a pas eu d'erreurs, la procédure GET analyse les paramètres de la requête du client. Si elle trouve une erreur quelconque elle le signale en utilisant là aussi la procédure **sendErreurs**. Sinon, elle calcule la nouvelle simulation, l'ajoute aux anciennes mémorisées dans la session courante et termine en envoyant sa réponse XML via la procédure **sendSimulations**. Cette dernière procède de façon analogue à la procédure **sendErreurs**.
- on remarquera que la servlet annonce sa réponse comme étant de type *text/xml* :

```
    // on précise le type de la réponse
    response.setContentType("text/xml");
```

Voici des exemples d'exécution. Le formulaire de départ est rempli de la façon suivante :



La base de données MySQL n'a pas été lancée rendant impossible la construction de l'objet **impots** dans la procédure **init** de la servlet. La réponse de celle-ci est alors la suivante :

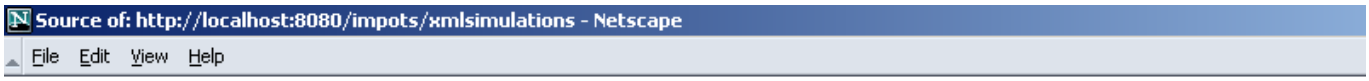



**Simulations de calculs d'impôts**

Les erreurs suivantes se sont produites :

- [TCX][MyODBC]Can't connect to MySQL server on 'localhost' (10061)

Le code reçu par le navigateur (View/Source) est le suivant :




```
<?xml version="1.0" encoding="windows-1252"?><?xml-stylesheet type="text/xsl" href="erreurs.xsl"?>
<erreurs>
<erreur>[TCX][MyODBC]Can't connect to MySQL server on 'localhost' (10061)</erreur>
</erreurs>
```

Si maintenant on refait deux simulations après avoir lancé la base de données MySQL, on obtient le résultat suivant :


**Simulations de calculs d'impôts**

| marié | enfants | salaire | impôt |
|---|---|---|---|
| oui | 2 | 200000 | 22504 |
| non | 2 | 200000 | 33388 |

Le navigateur a cette fois reçu le code suivant :


```
<?xml version="1.0" encoding="windows-1252"?><?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
<simulations>
<simulation marie="oui" enfants="2" salaire="200000" impot="22504" />
<simulation marie="non" enfants="2" salaire="200000" impot="33388" />
</simulations>
```

On remarquera que notre nouvelle application est plus simple qu'auparavant du fait de la suppression des fichiers JSP. Une partie du travail fait par ces pages a été transférée sur les feuilles de style XSL. L'intérêt de notre nouvelle répartition des tâches est qu'une fois le format XML des réponses de la servlet a été fixé, le développement des feuilles de style est indépendant de celui de la servlet.

## 5.3 Analyse d'un document XML en Java

Les versions 7 et 8 de notre application **impots** vont être des clients programmés de la servlet précédente **xmlsimulations**. Ceux-ci vont recevoir du code XML qu'ils vont devoir analyser pour en retirer les informations qui les intéressent. Nous allons faire ici une

pause dans nos différentes versions et apprendre comment on peut analyser un document XML en Java. Nous le ferons à partir d'un exemple livré avec JBuilder 7 appelé *MySaxParser*. Le programme s'appelle de la façon suivante :

```
dos>java MySaxParser
Usage: java MySaxParser [URI]
```

L'application *MySaxParser* admet un paramètre : l'URI (Uniform Resource Identifier) du document XML à analyser. Dans notre exemple, cette URI sera simplement le nom d'un fichier XML placé dans le répertoire de l'application *MySaxParser*. Considérons deux exemples d'exécution. Dans le premier exemple, le fichier XML analysé est le fichier **erreurs.xml** :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="erreurs.xsl"?>
<erreurs>
  <erreur>erreur 1</erreur>
  <erreur>erreur 2</erreur>
</erreurs>
```

L'analyse donne les résultats suivants :

```
dos> java MySaxParser erreurs.xml
Début du document
Début élément <erreurs>
Début élément <erreur>
[erreur 1]
Fin élément <erreur>
Début élément <erreur>
[erreur 2]
Fin élément <erreur>
Fin élément <erreurs>
Fin du document
```

Nous n'avions pas encore indiqué ce que faisait l'application *MySaxParser* mais l'on voit ici qu'elle affiche la structure du document XML analysé. Le second exemple analyse le fichier XML **simulations.xml** :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="simulations.xsl"?>
<simulations>
  <simulation marie="oui" enfants="2" salaire="200000" impot="22504"/>
  <simulation marie="non" enfants="2" salaire="200000" impot="33388"/>
</simulations>
```

L'analyse donne les résultats suivants :

```
dos>java MySaxParser simulations.xml
Début du document
Début élément <simulations>
Début élément <simulation>
marie = oui
enfants = 2
salaire = 200000
impot = 22504
Fin élément <simulation>
Début élément <simulation>
marie = non
enfants = 2
salaire = 200000
impot = 33388
Fin élément <simulation>
Fin élément <simulations>
Fin du document
```

La classe *MySaxParser* contient tout ce dont nous avons besoin dans notre application **impots** puisqu'elle a été capable de retrouver soit les erreurs soit les simulations que pourrait envoyer le serveur web. Examinons son code :

```
import java.io.IOException;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;
import java.util.regex.*;

// la classe
public class MySaxParser extends DefaultHandler {
```

```
// valeur d'un élement de l'arbre XML
private StringBuffer valeur=new StringBuffer();
// une expression régulière de la valeur d'un élément lorsqu'on veut ignorer
// les "blancs" qui la précèdent ou la suivent
private static Pattern ptnValeur=null;
private static Matcher résultats=null;
```

```
// -------- main
public static void main(String[] argv) {
 // vérification du nombre de paramètres
 if (argv.length != 1) {
   System.out.println("Usage: java MySaxParser [URI]");
   System.exit(0);
 }
 // on récupère l'URI du fichier XML à analyser
 String uri = argv[0];
 try {
   // création d'un analyseur XML (parseur)
   XMLReader parser = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
   // on indique au parseur l'objet qui va implémenter les méthodes
   // startDocument, endDocument, startElement, endElement, characters
   MySaxParser MySaxParserInstance = new MySaxParser();
   parser.setContentHandler(MySaxParserInstance);
   // on initialise le modèle de valeur d'un élément
   ptnValeur=Pattern.compile("^\\s*(.*?)\\s*$");
   // on indique au parseur le document XML à analyser
   parser.parse(uri);
 }
 catch(Exception ex) {
   // erreur
   System.err.println("Erreur : " + ex);
   // trace
   ex.printStackTrace();
 }
}//main
```

```
// -------- startDocument
public void startDocument() throws SAXException {
 // procédure appelée lorsque le parseur rencontre le début du document
 System.out.println("Début du document");
}//startDocument
```

```
// -------- endDocument
public void endDocument() throws SAXException {
 // procédure appelée lorsque le parseur rencontre la fin du document

 System.out.println("Fin du document");
}//endDocument
```

```
// -------- startElement
public void startElement(String uri, String localName, String qName,
 Attributes attributes) throws SAXException {
 // procédure appelée par le parseur lorsqu'il rencontre un début de balise
 // uri : URI du document analysé ?
 // localName : nom de l'élément en cours d'analyse
 // qName : idem mais "qualifié" par un espace de noms s'il y en a un
 // attributes : liste des attributs de l'éélment

 // suivi
 System.out.println("Début élément <"+localName+">");
 // l'élément a-t-il des attributs ?
 for (int i = 0; i < attributes.getLength(); i++) {
   System.out.println(attributes.getLocalName(i) + " = " + attributes.getValue(i));
 }//for
}//startElement
```

```
// -------- characters
public void characters(char[] ch, int start, int length) throws SAXException {
 // procédure appelée de façon répétée par le parseur lorsqu'il rencontre du texte
 // entre deux balises <balise>texte</balise>
 // le texte est dans ch à partir du caractère start sur length caractères

 // le texte est ajouté au buffer valeur
 valeur.append(ch, start, length);
}//characters
```

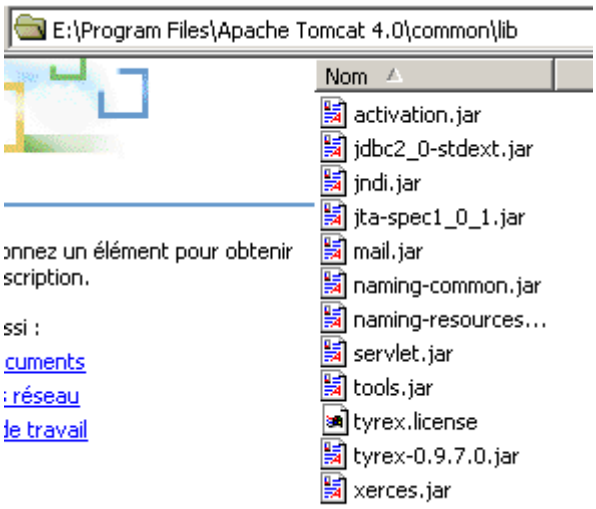
```
    // -------- endElement
   public void endElement(String uri, String localName, String qName)
     throws SAXException {
    // procédure appelée par le parseur lorsqu'il rencontre une fin de balise
    // uri : URI du document analysé ?
    // localName : nom de l'élément en cours d'analyse
    // qName : idem mais "qualifié" par un espace de noms s'il y en a un

    // on affiche la valeur de l'élément
    String strValeur=valeur.toString();
    if (ptnValeur==null) System.out.println("null");
    résultats=ptnValeur.matcher(strValeur);
    if (résultats.find() && ! résultats.group(1).equals("")){
      System.out.println("["+résultats.group(1)+"]");
    }//if
    // on met la valeur de l'élément à vide
    valeur.setLength(0);

    // suivi
    System.out.println("Fin élément <"+localName+">");
   }//endElement
}//classe
```

Définissons tout d'abord un sigle qui revient fréquemment dans l'analyse de documents XML : **SAX** qui signifie **Simple Api for Xml**. C'est un ensemble de classes java facilitant le travail avec les documents XML. Il y a deux versions d'API : SAX1 et SAX2. L'application ci-dessus utilise l'API SAX2.

L'application importe un certain nombre de paquetages :

```
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;
```

Les deux premiers viennent avec le JDK 1.4, le troisième non. Le paquetage **xerces.jar** est disponible sur le site du serveur Web Apache. Il vient avec JBuilder 7 mais aussi avec Tomcat 4.x :




Si donc on veut compiler l'application précédente en-dehors de JBuilder 7 et qu'on dispose du JDK 1.4 et de Tomcat 4.x on pourra écrire :

```
dos>javac -classpath ".;E:\Program Files\Apache Tomcat 4.0\common\lib\xerces.jar" MySaxParser.java
```

A l'exécution, on fera de même :

```
dos>java -classpath ".;E:\Program Files\Apache Tomcat 4.0\common\lib\xerces.jar" MySaxParser
simulations.xml
```

La classe *MySaxParser* dérive de la classe *DefaultHandler*. Nous y reviendrons. Etudions le code de la procédure *main* :

```
    // on récupère l'URI du fichier XML à analyser
    String uri = argv[0];
    try {
      // création d'un analyseur XML (parseur)
      XMLReader parser = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
      // on indique au parseur l'objet qui va implémenter les méthodes
      // startDocument, endDocument, startElement, endElement, characters
      MySaxParser MySaxParserInstance = new MySaxParser();
      parser.setContentHandler(MySaxParserInstance);
      // on initialise le modèle de valeur d'un élément
      ptnValeur=Pattern.compile("^\\s*(.*?)\\s*$");
      // on indique au parseur le document XML à analyser
      parser.parse(uri);
    }
    catch(Exception ex) {
      // erreur
      System.err.println("Erreur : " + ex);
      // trace
      ex.printStackTrace();
    }
```

Pour analyser un document XML, notre application a besoin d'un analyseur de code XML appelé "parseur".

```
      XMLReader parser = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
```

Le parseur XML utilisé est celui fourni par le paquetage *xerces.jar*. L'objet récupéré est de type *XMLReader*. XMLReader est une interface dont nous utilisons ici deux méthodes :

| | |
|---|---|
| `void setContentHandler(ContentHandler handler)` | indique au parseur l'objet de type *ContentHandler* qui va traiter les événements qu'il va générer lors de l'analyse du document XML |
| `void parse(InputSource input)` | démarre l'analyse du document XML passé en paramètre |

Lorsque le parseur va analyser le document XML, il va émettre des événements tels que : "*j'ai rencontré le début du document, le début d'une balise, un attribut de balise, le contenu d'une balise, la fin d'une balise, la fin du document, ...*". Il transmet ces événements à l'objet *ContentHandler* qu'on lui a donné. *ContentHandler* est une interface qui définit les méthodes à implémenter pour traiter tous les événements que le parseur XML peut générer. *DefaultHandler* est une classe qui fait une implémentation par défaut de ces méthodes. Les méthodes implémentées dans *DefaultHandler* ne font rien mais elles existent. Lorsqu'on doit indiquer au parseur quel objet va traiter les événements qu'il va générer au moyen de l'instruction

```
void setContentHandler(ContentHandler handler)
```

il est alors pratique de passer en paramètre un objet de type *DefaultHandler*. Si on s'en tenait là, aucun événement du parseur ne serait traité mais notre programme serait syntaxiquement correct. Dans la pratique, on passe en paramètre au parseur, un objet dérivé de la classe *DefaultHandler*, dans lequel sont redéfinies les méthodes traitant les seuls événements qui nous intéressent. C'est ce qui est fait ici :

```
      // on indique au parseur l'objet qui va implémenter les méthodes
      // startDocument, endDocument, startElement, endElement, characters
      MySaxParser MySaxParserInstance = new MySaxParser();
      parser.setContentHandler(MySaxParserInstance);
      // on indique au parseur le document XML à analyser
      parser.parse(uri);
```

On passe au parseur un exemplaire de la classe *mySaxParser* qui est notre classe et qui a été définie plus haut par la déclaration

```
public class MySaxParser extends DefaultHandler {
```

et on lance l'analyse du document dont on a passé l'URI en paramètre. A partir de là, l'analyse du document XML commence. Le parseur émet des événements et pour chacun d'eux appelle une méthode précise de l'objet chargé de traiter ces événements, ici notre objet *MySaxParser*. Celui-ci traite cinq événements particuliers, les autres étant ignorés :

| événement émis par le parseur | méthode de traitement |
|---|---|
| `début du document` | **void startDocument()** |
| `fin du document` | **void endDocument()** |
| `début d'un élément :<balise>` | **public void startElement(String uri, String localName, String qName, Attributes attributes)**<br>*uri* : ? |

*localName* : nom de l'élément analysé. Si l'élément rencontré est <*simulations*>, on aura localName="simulations".
*qName* : nom qualifié par un espace de noms de l'élément analysé. Un document XML peut définir un espace de noms, XX par exemple. Le nom qualifié de la balise précédente serait alors XX:simulations.
*attributes* : liste des attributs de la balise

```
valeur d'un élément :
<balise>valeur</balise>
```

**public void characters(char[] ch, int start, int length)**
*ch* : tableau de caractères
*start* : indice du 1er caractère à utiliser dans le tableau ch
*length* : nombre de caractères à prendre dans le tableau ch

La méthode *characters* peut être appelée de façon répétée. Pour construire la valeur d'un élément, on utilise alors un buffer qu'on :

- vide au départ de l'élément
- complète à chaque nouvel appel de la méthode *characters*
- termine à la fin de l'élément

```
fin d'un élément :
</balise> ou <balise
.../>
```

**void endElement(String uri, String localName, String qName)**

les paramètres sont ceux de la méthode *startElement*.

La méthode *startElement* permet de récupérer les attributs de l'élément grâce au paramètre **attributes** de type **Attributes** :

- le nombre d'attributs est disponible dans **attributes.getLength()**
- le nom de l'attribut i est disponible dans **attributes.getLocalName(i)**
- la valeur de l'attribut i est disponible dans **attributes.getValue(i)**
- la valeur de l'attribut de nom *localName* dans **attributes.getValue(localName)**

Une fois ceci expliqué, le programme précédent accompagné des exemples d'exécution se comprend de lui-même. Une expression régulière a été utilisée pour récupérer les valeurs des éléments afin qu'un texte XML comme :

```
<erreur>
  erreur 1
</erreur>
```

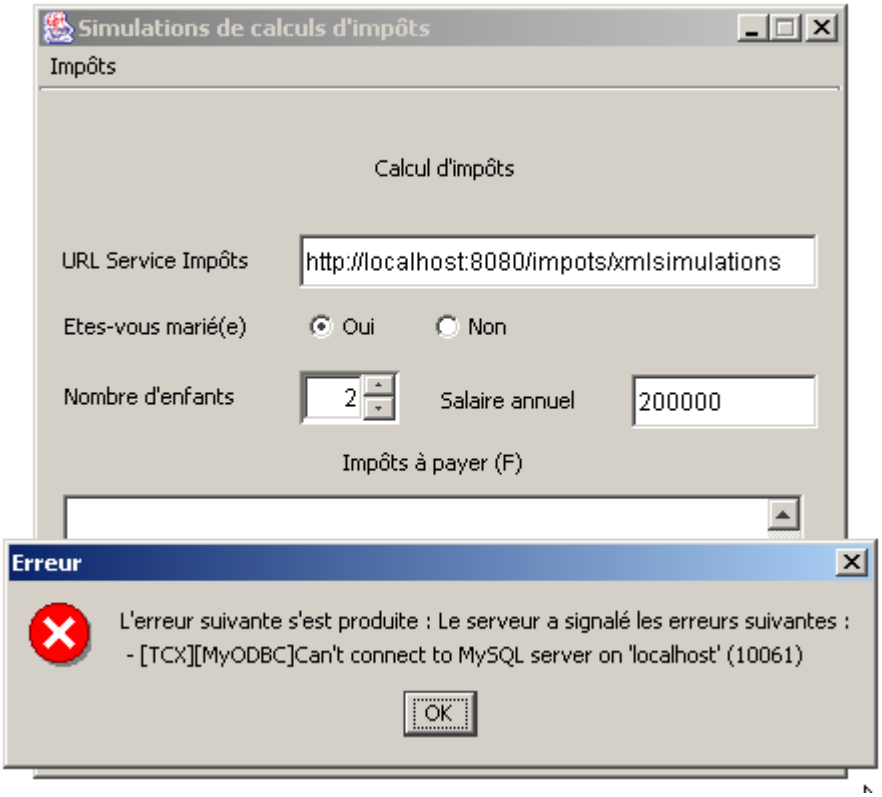
donne pour valeur associée à la balise <*erreur*> le texte **erreur 1** débarrassé des espaces et sauts de lignes qui pourraient le précéder et/ou le suivre.

## 5.4 Application impôts : version 7

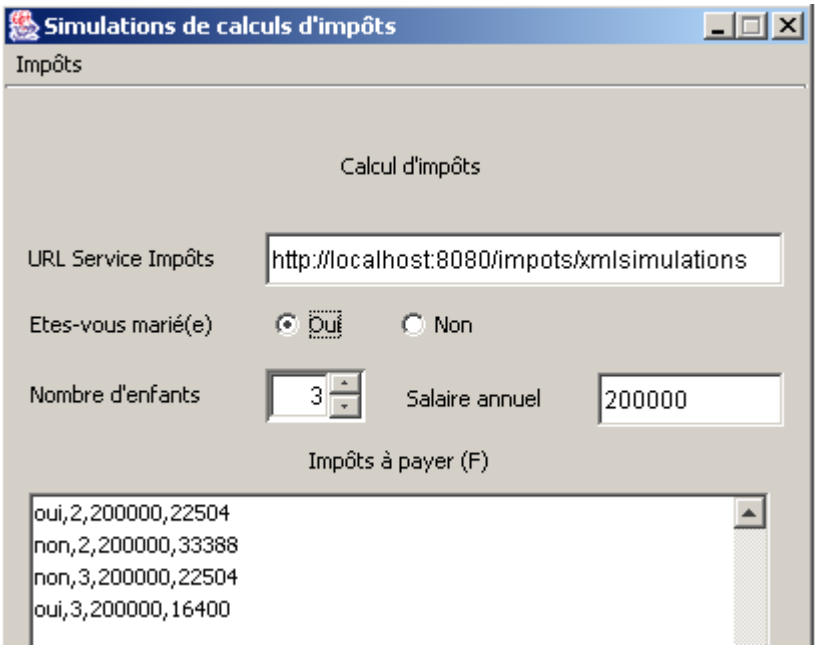
Nous avons maintenant tous les éléments pour écrire des clients programmés pour notre service d'impôts qui délivre du XML. Nous reprenons la version 4 de notre application pour faire le client et nous gardons la version 6 pour ce qui est du serveur. Dans cette application client-serveur :

- le service de simulations du calcul d'impôts est fait la servlet **xmlsimulations**. La réponse du serveur est donc au format XML comme nous l'avons vu dans la version 6.
- le client n'est plus un navigateur mais un client java autonome. Son interface graphique est celle de la version 4.

Voici quelques exemples d'exécution. Tout d'abord un cas d'erreur : le client interroge la servlet **xmlsimulations** alors que celle-ci n'a pas pu s'initialiser correctement du fait que le SGBD MySQL n'était pas lancé :




On lance MySQL et on fait quelques simulations :




Le client de cette nouvelle version ne diffère du client de la version 4 que par la façon dont il traite la réponse du serveur. Rien d'autre ne change. Dans la version 4, le client recevait du code HTML dans lequel il allait chercher les informations qui l'intéressaient en utilisant des expressions régulières. Ici le client reçoit du code XML dans lequel il va récupérer les informations qui l'intéressent à l'aide d'un parseur XML.

Rappelons les grandes lignes de la procédure liée au menu **Calculer** de la version 4 de notre client puisque c'est principalement là que se font les changements :

```
  void mnuCalculer_actionPerformed(ActionEvent e) {
....
    try{
      // on calcule l'impôt
      calculerImpots(urlImpots,rdOui.isSelected(),nbEnfants.intValue(),salaire);
    }catch (Exception ex){
      // on affiche l'erreur
      JOptionPane.showMessageDialog(this,"L'erreur suivante s'est produite : " +
ex.getMessage(),"Erreur",JOptionPane.ERROR_MESSAGE);
    }
....
   }//mnuCalculer_actionPerformed
```

```
   public void calculerImpots(URL urlImpots,boolean marié, int nbEnfants, int salaire)
        throws Exception{
    // calcul de l'impôt
    // urlImpots : URL du service des impôts
    // marié : true si marié, false sinon
    // nbEnfants : nombre d'enfants
    // salaire : salaire annuel

    // on retire d'urlImpots les infos nécessaire à la connexion au serveur d'impôts
....
    try{
      // on se connecte au serveur
....

      // on crée les flux d'entrée-sortie du client TCP
....

      // on demande l'URL - envoi des entêtes HTTP
....

      // on lit la 1ère ligne de la réponse
....
      // on lit la réponse jusqu'à la fin des entêtes en cherchant l'éventuel cookie
      while((réponse=IN.readLine())!=null){
....      }//while
```

```
      // c'est fini pour les entêtes HTTP - on passe au code HTML
      // pour récupérer les simulations
      ArrayList listeSimulations=getSimulations(IN,OUT,simulations);
      simulations.clear();
      for (int i=0;i<listeSimulations.size();i++){
        simulations.addElement(listeSimulations.get(i));
      }
```

```
      // c'est fini
....
  }//calculerImpots

  private ArrayList getSimulations(BufferedReader IN, PrintWriter OUT, DefaultListModel simulations)
throws Exception{
....
  }
```

Tout ce code reste valide dans la nouvelle version. Seul le traitement de la réponse HTML du serveur (partie encadrée ci-dessus) et son affichage doivent être remplacés par le traitement de la réponse XML du serveur et son affichage :

```
      // c'est fini pour les entêtes HTTP - on passe au code XML
      // pour récupérer les simulations ou les erreurs
      ImpotsSaxParser parseur=new ImpotsSaxParser(IN);
      ArrayList listeErreurs=parseur.getErreurs();
      ArrayList listeSimulations=parseur.getSimulations();
      // fermeture connexion au serveur
      client.close();

      // nettoyage liste d'affichage
      simulations.clear();
      // erreurs
      if(listeErreurs.size()!=0){
        // on concatène toutes les erreurs
        String msgErreur="Le serveur a signalé les erreurs suivantes :\n";
        for(int i=0;i<listeErreurs.size();i++){
          msgErreur+=" - "+(String)listeErreurs.get(i);
        }
        // affichage erreurs
```

```
        throw new Exception(msgErreur);
      }//if

      // simulations
      for (int i=0;i<listeSimulations.size();i++){
        simulations.addElement(listeSimulations.get(i));
      }
      return;
```

Que fait la portion de code ci-dessus ?

- elle crée un parseur XML et lui passe le flux IN qui contient le code XML envoyé par le serveur. Ce flux contenait aussi les entêtes HTTP mais ceux-ci ont déjà été lus et traités. Ne reste donc que la partie XML de la réponse. Le parseur produit deux listes de chaînes de caractères : la liste des erreurs s'il y en a eu, sinon celle des simulations. Ces deux listes sont exclusives l'une de l'autre.
- si la liste des erreurs est non vide, les messages contenus dans la liste sont concaténés en un seul message d'erreur et une exception est lancée avec comme paramètre ce message. Cette exception fait l'objet d'un affichage dans la procédure *mnuCalculer_actionPeformed* qui a appelé *calculerImpots*.
- si la liste des simulations est non vide, elle fait l'objet d'un affichage dans le composant jList de l'interface graphique.

Découvrons maintenant le parseur de la réponse XML du serveur, parseur qui découle directement de l'étude que nous avons faite précédemment sur la façon d'analyser un document XML en java :

```
import java.io.IOException;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.apache.xerces.parsers.SAXParser;
import java.util.regex.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

// la classe
public class ImpotsSaxParser extends DefaultHandler {

  // valeur d'un élement de l'arbre XML
  private StringBuffer valeur=new StringBuffer();
  // une expression régulière de la valeur d'un élément lorsqu'on veut ignorer
  // les "blancs" qui la précèdent ou la suivent
  private Pattern ptnValeur=null;
  private Matcher résultats=null;
  // les listes d'éléments XML
  private ArrayList listeSimulations=new ArrayList();
  private ArrayList listeErreurs=new ArrayList();
  // éléments XML
  private ArrayList éléments=new ArrayList();
  String élément="";

    // -------- constructeur
    public ImpotsSaxParser(BufferedReader IN) throws Exception{
     // création d'un analyseur XML (parseur)
     XMLReader parser = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.SAXParser");
     // on indique au parseur l'objet qui va implémenter les méthodes
     // startDocument, endDocument, startElement, endElement, characters
     parser.setContentHandler(this);
     // on initialise le modèle de valeur d'un élément
     ptnValeur=Pattern.compile("^\\s*(.*?)\\s*$");
     // au départ pas d'élément XML courant
     éléments.add("");
     // on analyse le document
     parser.parse(new InputSource(IN));
    }//constructeur

    // -------- startElement
    public void startElement(String uri, String localName, String qName,
     Attributes attributes) throws SAXException {
     // procédure appelée par le parseur lorsqu'il rencontre un début de balise
     // uri : URI du document analysé ?
     // localName : nom de l'élément en cours d'analyse
     // qName : idem mais "qualifié" par un espace de noms s'il y en a un
     // attributes : liste des attributs de l'éélment

     // on note le nom de l'élément
     élément=localName.toLowerCase();
```

```
    éléments.add(élément);
    // l'élément a-t-il des attributs ?
    if(élément.equals("simulation") && attributes.getLength()==4){
      // c'est une simulation - on récupère les attributs
      String simulation=attributes.getValue("marie")+","+
                    attributes.getValue("enfants")+","+
                    attributes.getValue("salaire")+","+
                    attributes.getValue("impot");
      // on ajoute la simulation à la liste des simulations
      listeSimulations.add(simulation);
    }//if
   }//startElement
```

```
   // -------- characters
   public void characters(char[] ch, int start, int length) throws SAXException {
    // procédure appelée de façon répétée par le parseur lorsqu'il rencontre du texte
    // entre deux balises <balise>texte</balise>
    // le texte est dans ch à partir du caractère start sur length caractères

    // le texte est ajouté au buffer valeur s'il s'agit de l'élément erreur
    if (élément.equals("erreur"))
       valeur.append(ch, start, length);
   }//characters
```

```
    // -------- endElement
   public void endElement(String uri, String localName, String qName)
     throws SAXException {
    // procédure appelée par le parseur lorsqu'il rencontre une fin de balise
    // uri : URI du document analysé ?
    // localName : nom de l'élément en cours d'analyse
    // qName : idem mais "qualifié" par un espace de noms s'il y en a un

    // cas de l'erreur
    if(élément.equals("erreur")){
      // on récupère la valeur de l'élément erreur
      String strValeur=valeur.toString();
      // on la débarrasse de ses "blancs" inutiles et on l'enregistre dans la liste des
      // erreurs si elle est non vide
      résultats=ptnValeur.matcher(strValeur);
      if (résultats.find() && ! résultats.group(1).equals("")){
        listeErreurs.add(résultats.group(1));
      }//if
    }
    // on met la valeur de l'élément à vide
    valeur.setLength(0);
    // on réinitialise le nom de l'élément
    éléments.remove(éléments.size()-1);
    élément=(String)éléments.get(éléments.size()-1);
   }//endElement
```

```
  // --------- getErreurs
  public ArrayList getErreurs(){
    return listeErreurs;
  }

  // --------- getSimulations
  public ArrayList getSimulations(){
    return listeSimulations;
  }

}//classe
```

- le constructeur reçoit le flux XML **IN** à analyser et fait immédiatement cette analyse. Celle-ci terminée, l'objet a été construit ainsi que les listes (*ArrayList*) d'erreurs (**listeErreurs**) et de simulations (**listeSimulations**) ont été construites. Il ne reste plus à la procédure qui a construit l'objet qu'à récupérer les deux listes à l'aide des méthodes **getErreurs** et **getSimulations**.
- seuls trois événements générés par le parseur XML nous intéressent ici :
  - début d'un élément XML, événement qui sera traité par la procédure **startElement**. Celle-ci aura à traiter les balises <*simulation marie=".." enfants=".." salaire=".." impot=".."*> et <*erreur*>...</*erreur*>.
  - valeur d'un élément XML, événement qui sera traité par la procédure **characters**.
  - fin d'un élément XML, événement qui sera traité par la procédure **endElement**.
- dans la procédure **startElement**, si on a affaire à l'élément <*simulation marie=".." enfants=".." salaire=".." impot=".."*> on récupère les quatre attributs par **attributes.getValue("nom de l'attribut")**. Dans tous les cas, on mémorise le nom de l'élément dans une variable **élément** et on l'ajoute à une liste (**ArrayList**) d'éléments : **élé1 élé2 ... élén**. Cette liste est

gérée comme une pile dont le dernier élément est l'élément XML en cours d'analyse. Lorsqu'on a l'événement *"fin d'élément"* le dernier élément de la liste est retiré et le nouvel élément courant est modifié. Ceci est fait dans la procédure **endElement**.

- la procédure **characters** est identique à celle qui avait été étudiée dans un exemple précédent. On prend simplement soin de vérifier que l'élément courant est bien l'élément <*erreur*>, précaution inutile normalement ici. Ce type de précaution avait été également pris dans la procédure **startElement** pour vérifier qu'on avait affaire à un élément <*simulation*>.

## 5.5 Conclusion

Grâce à sa réponse XML l'application **impots** est devenue plus facile à gérer à la fois pour son concepteur et les concepteurs des application clientes.

- la conception de l'application serveur peut maintenant être confiée à deux types de personnes : le développeur Java de la servlet et l'infographiste qui va gérer l'apparence de la réponse du serveur dans les navigateurs. Il suffit à ce dernier de connaître la structure de la réponse XML du serveur pour construire les feuilles de style qui vont l'accompagner. Rappelons que celles-ci font l'objet de fichiers XSL à part et indépendants de la servlet java. L'infographiste peut donc travailler indépendamment du développeur Java.
- les concepteurs des applications clientes ont eux aussi simplement besoin de connaître la structure de la réponse XML du serveur. Les modifications que pourrait apporter l'infographiste aux feuilles de style n'ont aucune répercussion sur cette réponse XML qui reste toujours la même. C'est un énorme avantage.
- Comment le développeur peut-il faire évoluer sa servlet Java sans tout casser ? Tout d'abord, tant que sa réponse XML ne change pas, il peut organiser sa servlet comme il le veut. Il peut aussi faire évoluer la réponse XML tant qu'il garde les éléments **<erreur>** et **<simulation>** attendus par ses clients. Il peut ainsi ajouter de nouvelles balises à cette réponse. L'infographiste les prendra en compte dans ses feuilles de style et les navigateurs pourront avoir les nouvelles versions de la réponse. Les clients programmés eux continueront à fonctionner avec l'ancien modèle, les nouvelles balises étant tout simplement ignorées. Pour que cela soit possible, il faut que dans l'analyse XML de la réponse du serveur, les balises cherchées soient bien identifiées. C'est ce qui a été fait dans notre client XML de l'application **impôts** où dans les procédures on disait spécifiquement qu'on traitait les balises <*erreur*> et <*simulation*>. Du coup les autres balises sont ignorées.

# 6. A suivre...

Nous avons donné dans ce document suffisamment d'informations pour démarrer sérieusement la programmation web en Java. Certains thèmes n'ont été qu'effleurés et mériteraient des approfondissements :

- **XML (eXtended Markup Language), Feuilles de style CSS (Cascading Style Sheets) et XSL (eXtended Stylesheet Language)**
  Nous avons abordé ces thèmes mais beaucoup reste à faire. Ils méritent à eux trois un polycopié.
- **Javascript**
  Nous avons utilisé ici et là quelques scripts Javascript exécutés par le navigateur sans jamais s'y attarder. Nous ne sommes par exemple jamais entrés dans les détails du langage Javascript. Par manque de place et de temps. Ce langage peut lui aussi nécessiter un livre à lui tout seul. Nous avons rajouté en annexe, trois exemples significatifs en javascript là encore sans les expliquer. Néanmoins ils sont compréhensibles par la simple lecture de leurs commentaires et peuvent servir d'exemples.
- **Java beans, balises JSP**
  Nous avons montré comment nous pouvions séparer le code Java et le code HTML dans une application Web. Le code Java est rassemblé dans une ou plusieurs servlets et le code HTML dans une ou plusieurs pages JSP. Néanmoins dans nos exemples, il restait souvent quelques lignes de code Java dans les pages JSP. On peut améliorer les choses en utilisant dans les pages JSP des composants appelés **Java beans** et des balises JSP prédéfinies ou à créer soi-même (extensions de balises). Cet aspect des pages JSP n'est pas indispensable (on s'en est bien passé dans nos exemples qui sont réalistes) mais cependant utile à connaître.
- **Enterprise Java Beans (EJB)**
  Les EJB sont des composants fournis par les conteneurs dans lesquels s'exécutent servlets et pages JSP. Ils fournissent des services "système" qui évitent au développeur de longs et difficiles développements. Les services fournis par les EJB peuvent concerner les domaines suivants : transactions, sécurité, pooling de connexions à des bases de données, ...

Tous ces thèmes non développés ici le sont excellemment dans le livre "Programmation J2EE" aux éditions WROX et distribué par Eyrolles.

# Annexes

# 7. Les outils du développement web

Nous indiquons ici où trouver et comment installer les outils nécessaires au développement web. Certains outils ont vu leurs versions évoluer et il se peut que les explications données ici ne conviennent plus pour les versions les plus récentes. Le lecteur sera alors amené à s'adpater... Dans le cours de programmation web, nous utiliserons essentiellement les outils suivants, tous disponibles gratuitement :

- un **navigateur** récent capable d'afficher du XML. Les exemples du cours ont été testés avec Internet Explorer 6.
- un **JDK** (Java Development Kit) récent. Les exemples du cours ont été testés avec le JDK 1.4. Ce JDK amène avec lui le Plug-in Java 1.4 pour les navigateurs ce qui permet à ces derniers d'afficher des applets Java utilisant le JDK 1.4.
- un **environnement de développement Java** pour écrire des servlets Java. Ici c'est JBuilder 7.
- des **serveurs web** : Apache, PWS (Personal Web Server), Tomcat.
  - Apache sera utilisé pour le développement d'applications web en PERL (Practical Extracting and Reporting Language) ou PHP (Personal Home Page)
  - PWS sera utilisé pour le développement d'applications web en ASP (Active Server Pages) ou PHP
  - Tomcat sera utilisé pour le développement d'applications web à l'aide de servlets Java ou de pages JSP (Java Server pages)
- une **application de gestion de base de données** : MySQL
- **EasyPHP** : un outil qui amène ensemble le serveur Web Apache, le langage PHP et le SGBD MySQL

## 7.1 Serveurs Web, Navigateurs, Langages de scripts

1. **Serveurs Web principaux**
   - Apache (Linux, Windows)
   - Interner Information Server IIS (NT), Personal Web Server PWS (Windows 9x)
2. **Navigateurs principaux**
   - Internet Explorer (Windows)
   - Netscape (Linux, Windows)
3. **Langages de scripts côté serveur**
   - VBScript (IIS, PWS)
   - JavaScript (IIS, PWS)
   - Perl (Apache, IIS, PWS)
   - PHP (Apache, IIS, PWS)
   - Java (Apache, Tomcat)
   - Langages .NET
4. **Langages de scripts côté navigateur**
   - VBScript (IE)
   - Javascript (IE, Netscape)
   - Perlscript (IE)
   - Java (IE, Netscape)

## 7.2 Où trouver les outils

| | |
|---|---|
| `Netscape` | **http://www.netscape.com/ (lien downloads)** |
| `Internet Explorer` | **http://www.microsoft.com/windows/ie/default.asp** |
| `PHP` | **http://www.php.net**<br>**http://www.php.net/downloads.php (Windows Binaries)** |
| `PERL` | **http://www.activestate.com http://www.activestate.com/Products/**<br>**http://www.activestate.com/Products/ActivePerl/** |

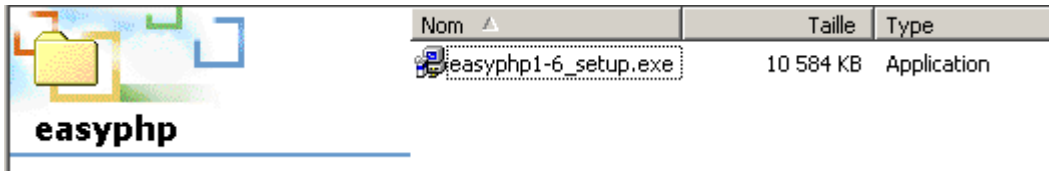
| | |
|---|---|
| `Vbscript, Javascript JAVA` | **http://msdn.microsoft.com/scripting (suivre le lien windows script)** |
| | **http://java.sun.com/** |
| | **http://java.sun.com/downloads.html (JSE)** |
| | **http://java.sun.com/j2se/1.4/download.html** |
| `Apache` | **http://www.apache.org/** |
| | **http://www.apache.org/dist/httpd/binaries/win32/** |
| `PWS` | **inclus dans NT 4.0 Option pack for Windows 95** |
| | **inclus dans le CD de Windows 98** |
| | **http://www.microsoft.com/ntserver/nts/downloads/recommended/NT4OptPk/win95.asp** |
| `IIS (windows NT/2000)` | **http://www.microsoft.com** |
| `Tomcat` | **http://jakarta.apache.org/tomcat/** |
| `JBuilder` | **http://www.borland.com/jbuilder/** |
| | **http://www.borland.com/products/downloads/download_jbuilder.html** |
| `EasyPHP` | **http://www.easyphp.org/** |
| | **http://www.easyphp.org/telechargements.php3** |

## 7.3 EasyPHP

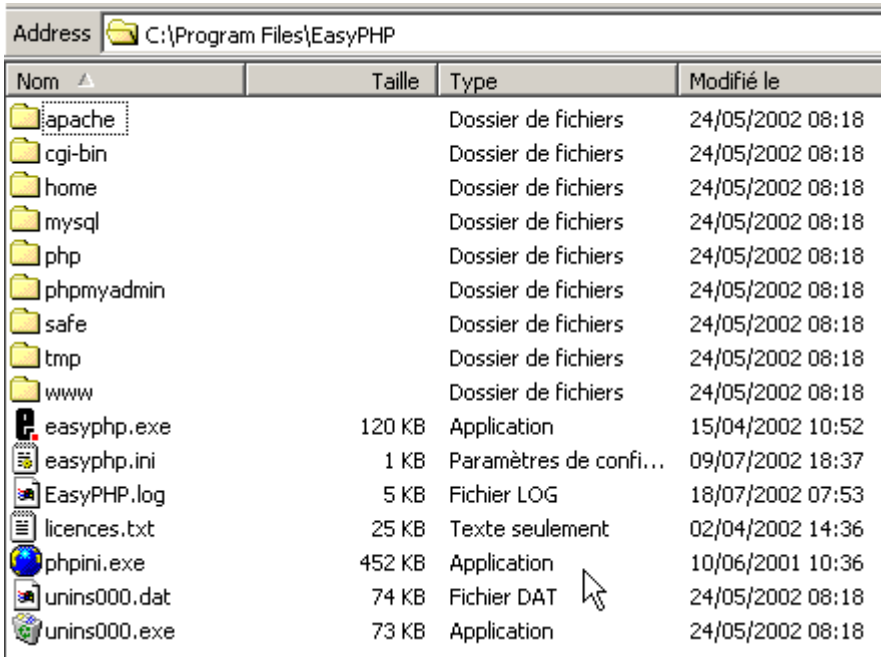
Cette application est très pratique en ce qu'elle amène dans un même paquetage :

- le serveur Web Apache (1.3.x)
- le langage PHP (4.x)
- le SGBD MySQL (3.23.x)
- un outil d'administration de MySQL : PhpMyAdmin

L'application d'installation se présente sous la forme suivante :




L'installation d'EasyPHP ne pose pas de problème et une arborescence est créée dans le système de fichiers :




| | |
|---|---|
| `easyphp.exe` | l'exécutable de l'application |
| `apache` | l'arborescence du serveur apache |

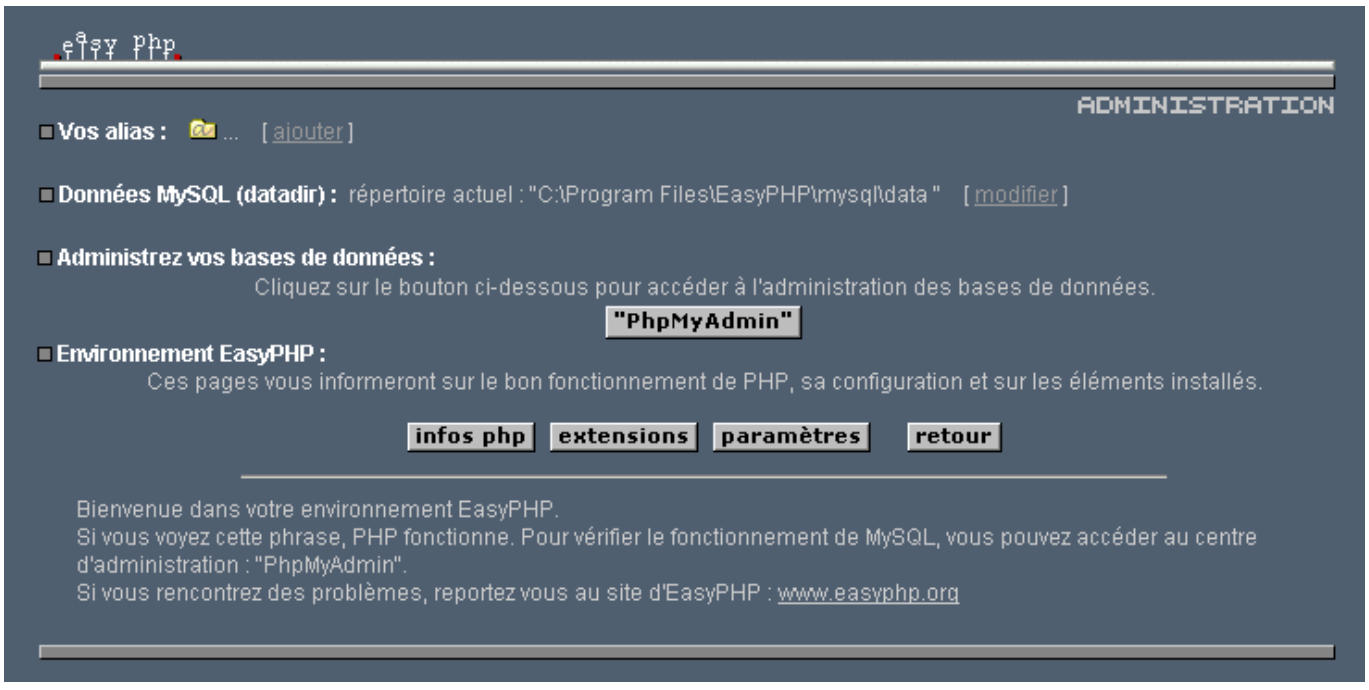
| | |
|---|---|
| `mysql` | l'arborescence du SGBD mysql |
| `phpmyadmin` | l'arborescence de l'application phpmyadmin |
| `php` | l'arborescence de php |
| `www` | racine de l'arborescence des pages web délivrées par le serveur apache d'EasyPHP |
| `cgi-bin` | arborescence où l'on peut palcer des script CGI pour le serveur Apache |

L'intérêt principal d'EasyPHP est que l'application arrive préconfigurée. Ainsi Apache, PHP, MySQL sont déjà configurés pour travailler ensemble. Lorsqu'on lance *EasyPhp* par son lien dans le menu des programmes, une icône se met en place en bas à droite de l'écran.



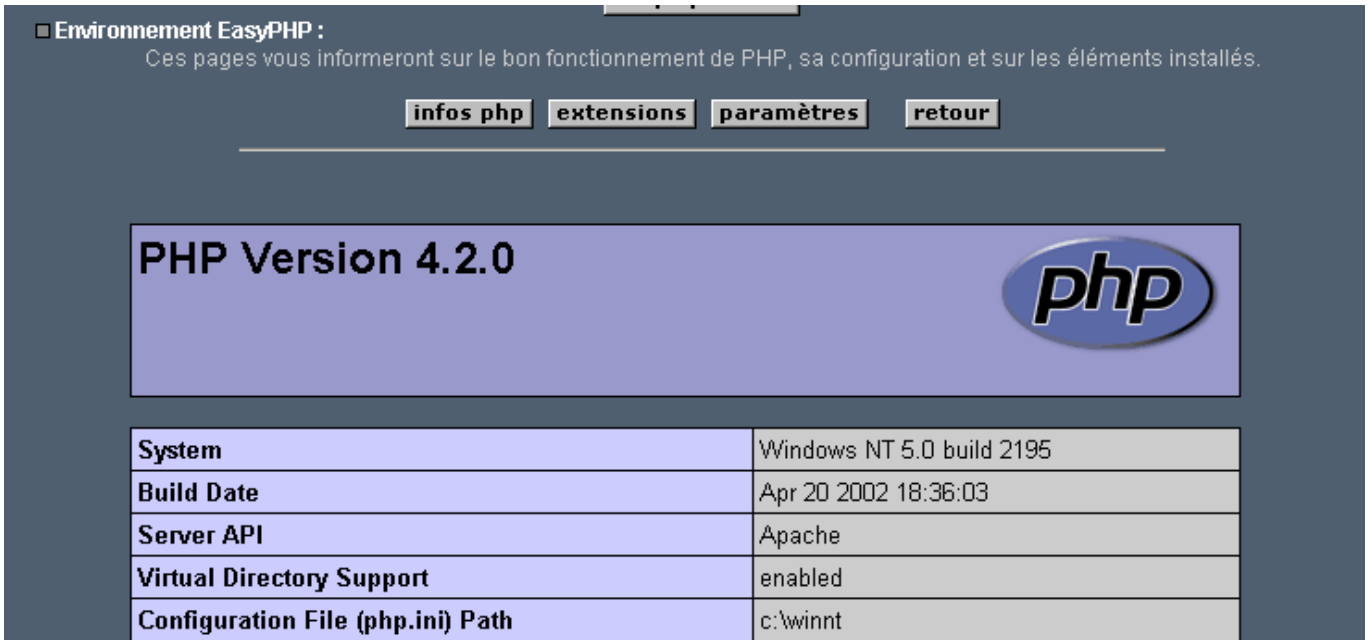
C'est le E avec un point rouge qui doit clignoter si le serveur web *Apache* et la base de données *MySQL* sont opérationnels. Lorsqu'on clique dessus avec le bouton droit de la souris, on accède à des options de menu :


L'option *Administration* permet de faire des réglages et des tests de bon fonctionnement :




## 7.3.1 Administration PHP

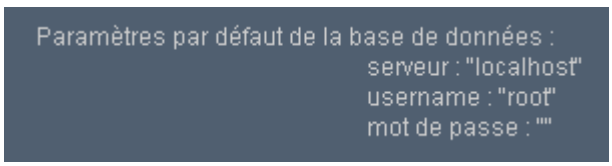
Le bouton **infos php** doit vous permettre de vérifier le bon fonctionnement du couple Apache-PHP : une page d'informations PHP doit apparaître :




Le bouton **extensions** donne la liste des extensions installées pour php. Ce sont en fait des bibliothèques de fonctions.


L'écran ci-dessus montre par exemple que les fonctions nécessaires à l'utilisation de la base MySQL sont bien présentes.

Le bouton **paramètres** donne le *login*/*motdepasse* de l'administrateur de la base de données MySQL.




L'utilisation de la base MySQL dépasse le cadre de cette présentation rapide mais il est clai ici qu'il faudrait mettre un mot de passe à l'administrateur de la base.

## 7.3.2 Administration Apache

Toujours dans la page d'administration d'EasyPHP, le lien **vos alias** permet de définir des alias associés à un répertoire. Cela permet de mettre des pages Web ailleurs que dans le répertoire www de l'arborescence d'easyPhp.


Si dans la page ci-dessus, on met les informations suivantes :


et qu'on utilise le bouton *valider* les lignes suivantes sont ajoutées au fichier *<easyphp>\apache\conf\httpd.conf* :

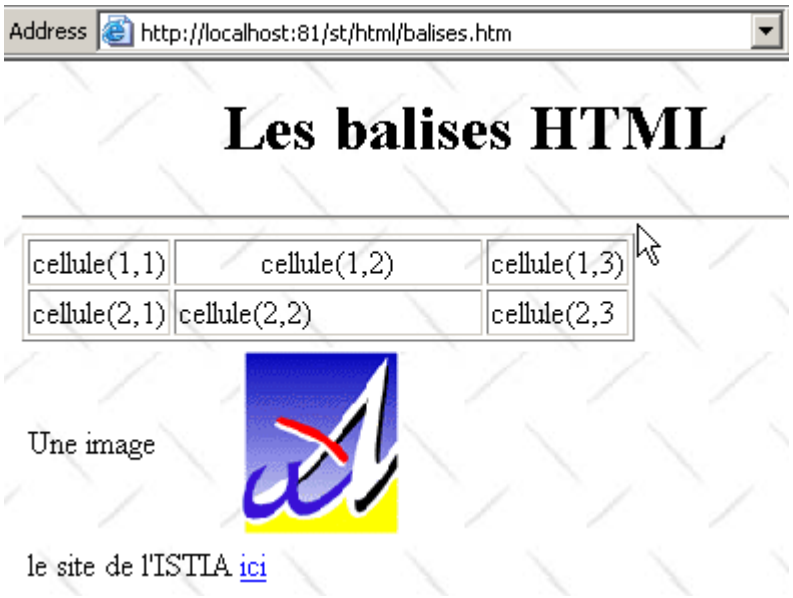
```
Alias /st/ "e:/data/serge/web/"

<Directory "e:/data/serge/web">
    Options FollowSymLinks Indexes
    AllowOverride None
    Order deny,allow
    allow from 127.0.0.1
    deny from all
</Directory>
```

*<easyphp>* désigne le répertoire d'installation d'EasyPHP. *httpd.conf* est le fichier de configuration du serveur Apache. On peut donc faire la même chose en éditant directement ce fichier. Une modification du fichier *httpd.conf* est normalement prise en compte immédiatement par Apache. Si ce n'était pas le cas, il faudrait l'arrêter puis le relancer, toujours avec l'icône d'easyphp :


Pour terminer notre exemple, on peut maintenant placer des pages web dans l'arborescence *e:\data\serge\web* :

```
C:\winnt\system32>dir e:\data\serge\web\html\balises.htm

14/07/2002  17:02               3 767 balises.htm
```

et demander cette page en utilisant l'alias **st** :

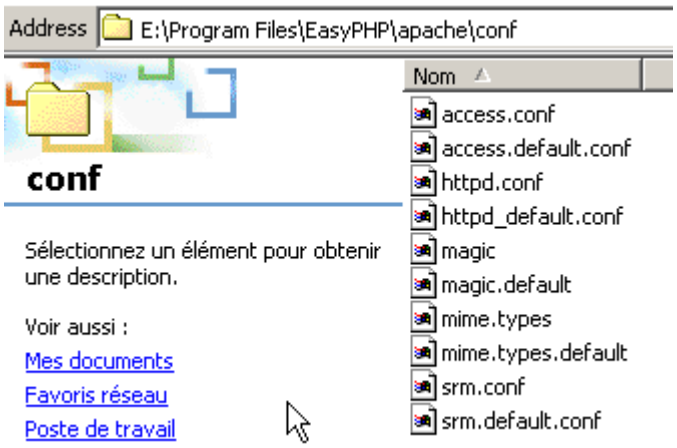



Dans cet exemple, le serveur Apache a été configuré pour travailler sur le port 81. Son port par défaut est 80. Ce point est contrôlé par la ligne suivante du fichier *httpd.conf* déjà rencontré :

```
Port 81
```

## 7.3.3 Le fichier de configuration d'Apache htpd.conf

Lorsqu'on veut configurer un peu finement Apache, on est obligé d'aller modifier "à la main" son fichier de configuration *httpd.conf* situé ici dans le dossier *<easyphp>\apache\conf* :



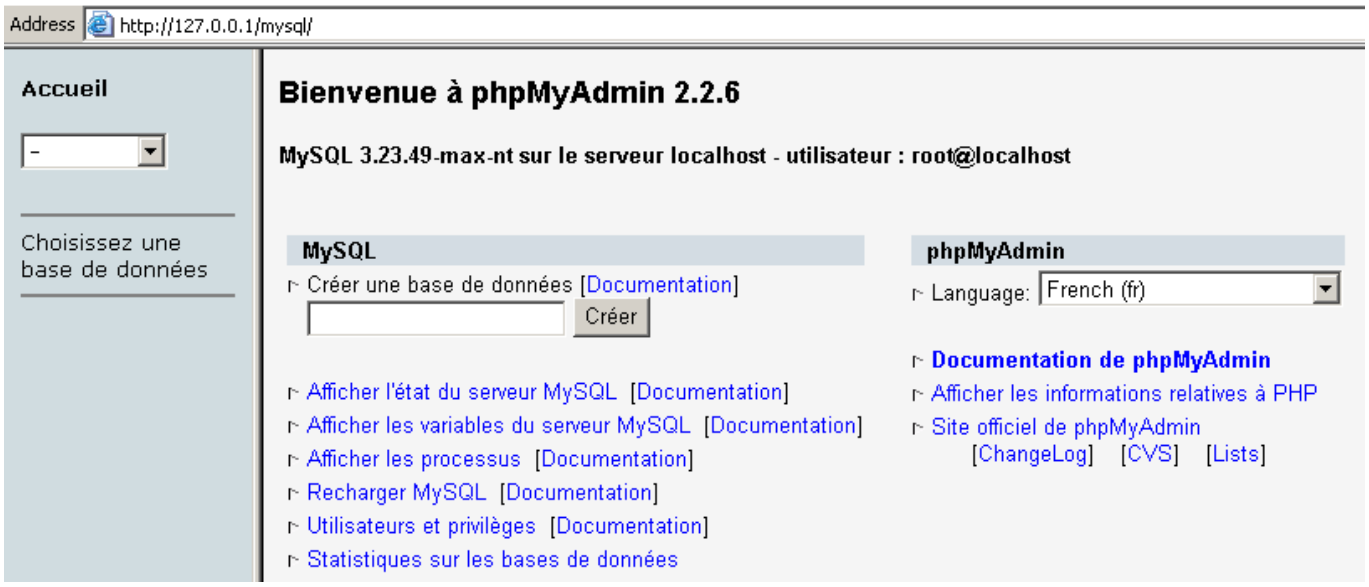

Voici quelques points à retenir de ce fichier de configuration :

| ligne(s) | rôle |
|---|---|
| `ServerRoot "D:/Program Files/Apache Group/Apache"` | indique le dossier où se trouve l'arborescence de Apache |
| `Port 80` | indique sur quel port va travailler le serveur Web. Classiquement c'est 80. En changeant cette ligne, on peut faire travailler le serveur Web sur un autre port |
| `ServerAdmin root@istia.univ-angers.fr` | l'adresse email de l'administrateur du serveur Apache |
| `ServerName stahe.istia.uang` | le nom de la machine sur laquelle "tourne" le serveur Apache |

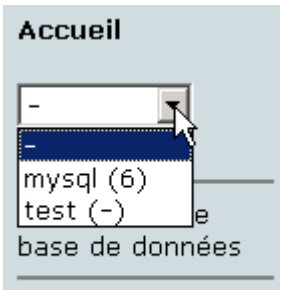
| | |
|---|---|
| `ServerRoot "E:/Program Files/EasyPHP/apache"` | le répertoire d'installation du serveur Apache. Lorsque dans le fichier de configuration, apparaissent des noms relatifs de fichiers, ils sont relatifs par rapport à ce dossier. |
| `DocumentRoot "E:/Program Files/EasyPHP/www"` | le dossier racine de l'arborescence des pages Web délivrées par le serveur. Ici, l'url *http://machine/rep1/fic1.html* correspondra au fichier *E:\Program Files\EasyPHP\www \rep1\fic1.html* |
| `<Directory "E:/Program Files/EasyPHP/www">` | fixe les propriétés du dossier précédent |
| `ErrorLog logs/error.log` | dossier des logs, donc en fait *<ServerRoot>\logs\error.log* : *E:\Program Files\EasyPHP\apache\logs\error.log*. C'est le fichier à consulter si vous constatez que le serveur Apache ne fonctionne pas. |
| `ScriptAlias /cgi-bin/ "E:/Program Files/EasyPHP/cgi-bin/"` | *E:\Program Files\EasyPHP\cgi-bin* sera la racine de l'arborescence où l'on pourra mettre des scripts CGI. Ainsi l'URL *http://machine/cgi-bin/rep1/script1.pl* sera l'url du script CGI *E:\Program Files\EasyPHP\cgi-bin \rep1\script1.pl*. |
| `<Directory "E:/Program Files/EasyPHP/cgi-bin/">` | fixe les propriétés du dossier ci-dessus |
| `LoadModule php4_module "E:/Program Files/EasyPHP/php/php4apache.dll"`<br>`AddModule mod_php4.c` | lignes de chargement des modules permettant à Apache de travailler avec PHP4. |
| `AddType application/x-httpd-php .phtml .pwml .php3 .php4 .php .php2 .inc` | fixe les suffixes des fichiers à considérer comme des fichiers comme devant être traités par PHP |

## 7.3.4 Administration de MySQL avec PhpMyAdmin

Sur la page d'administration d'EasyPhp, on clique sur le bouton **PhpMyAdmin** :



La liste déroulante sous *Accueil* permet de voir les bases de données actuelles.




Le nombre entre parenthèses est le nombre de tables. Si on choisit une base, les tables de celles-ci s'affichent :

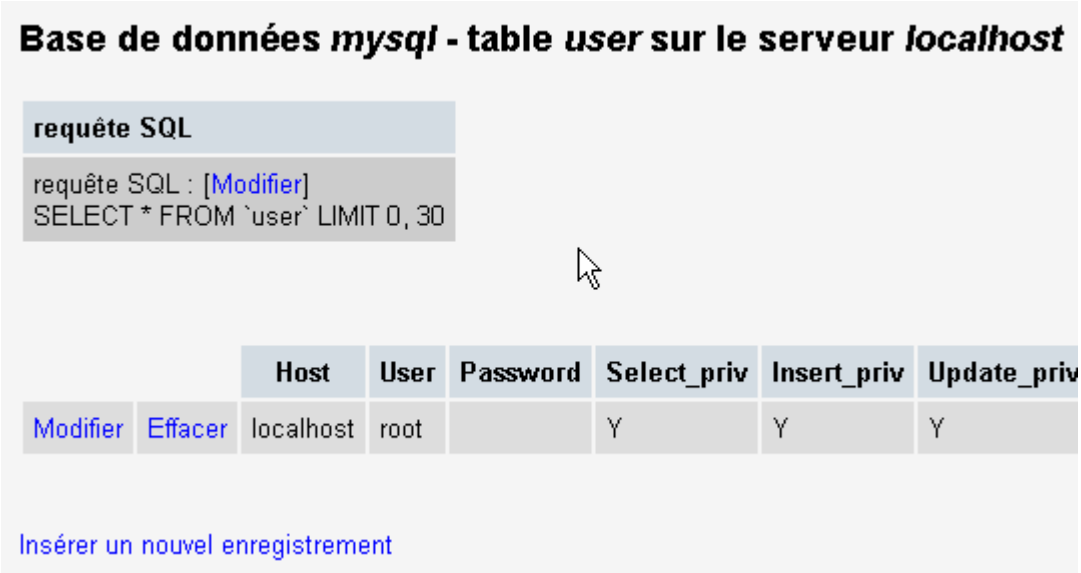

La page Web offre un certain nombre d'opérations sur la base :

**Base de données *mysql* sur le serveur *localhost***

| | Table | Action | | | | | | Enregistrements | Type | Taille |
|---|---|---|---|---|---|---|---|---|---|---|
| ☐ | **columns_priv** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 0 | MyISAM | 1,0 Ko |
| ☐ | **db** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 1 | MyISAM | 3,1 Ko |
| ☐ | **func** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 0 | MyISAM | 1,0 Ko |
| ☐ | **host** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 0 | MyISAM | 1,0 Ko |
| ☐ | **tables_priv** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 0 | MyISAM | 1,0 Ko |
| ☐ | **user** | Afficher | Sélectionner | Insérer | Propriétés | Supprimer | Vider | 1 | MyISAM | 2,4 Ko |
| | **6 table(s)** | **Somme** | | | | | | **2** | -- | **9,6 Ko** |

Tout cocher / Tout décocher  Pour la sélection :

Si on clique sur le lien *Afficher* de *user* :




Il n'y a ici qu'un seul utilisateur : root, qui est l'administrateur de MySQL. En suivant le lien *Modifier*, on pourrait changer son mot de passe qui est actuellement vide, ce qui n'est pas conseillé pour un administrateur.

Nous n'en dirons pas plus sur *PhpMyAdmin* qui est un logiciel riche et qui mériterait un développement de plusieurs pages.

## 7.4 PHP

Nous avons vu comment obtenir PHP au travers de l'application EasyPhp. Pour obtenir PHP directement, on ira sur le site **http://www.php.net.**

PHP n'est pas utilisable que dans le cadre du Web. On peut l'utiliser comme langage de scripts sous Windows. Créez le script suivant et sauvegardez-le sous le nom *date.php* :

```
<?
  // script php affichant l'heure
  $maintenant=date("j/m/y, H:i:s",time());
  echo "Nous sommes le $maintenant";
?>
```

Dans une fenêtre DOS, placez-vous dans le répertoire de date.php et exécutez-le **:**

```
E:\data\serge\php\essais>"e:\program files\easyphp\php\php.exe" date.php
X-Powered-By: PHP/4.2.0
Content-type: text/html

Nous sommes le 18/07/02, 09:31:01
```

## 7.5 PERL

Il est préférable que Internet Explorer soit déjà installé. S'il est présent, Active Perl va le configurer afin qu'il accepte des scripts PERL dans les pages HTML, scripts qui seront exécutés par IE lui-même côté client. Le site de Active Perl est à l'URL **http://www.activestate.com**A l'installation, PERL sera installé dans un répertoire que nous appelerons **<perl>**. Il contient l'arborescence suivante :

```
DEISL1   ISU        32 403  23/06/00  17:16 DeIsL1.isu
BIN           <REP>        23/06/00  17:15 bin
LIB           <REP>        23/06/00  17:15 lib
HTML          <REP>        23/06/00  17:15 html
EG            <REP>        23/06/00  17:15 eg
SITE          <REP>        23/06/00  17:15 site
HTMLHELP      <REP>        28/06/00  18:37 htmlhelp
```

L'exécutable *perl.exe* est dans **<perl>\bin**. Perl est un langage de scripts fonctionnant sous Windows et Unix. Il est de plus utilisé dans la programmation WEB. Écrivons un premier script :

```
# script PERL affichant l'heure

# modules
use strict;

# programme
my ($secondes,$minutes,$heure)=localtime(time);
print "Il est $heure:$minutes:$secondes\n";
```

Sauvegardez ce script dans un fichier *heure.pl*. Ouvrez une fenêtre DOS, placez-vous dans le répertoire du script précédent et exécutez-le :

```
E:\data\serge\Perl\Essais>e:\perl\bin\perl.exe heure.pl
Il est 9:34:21
```

## 7.6 Vbscript, Javascript, Perlscript

Ces langages sont des langages de script pour windows. Ils peuvent fonctionner dans différents conteneurs tels

- *Windows Scripting Host* pour une utilisation directe sous Windows notamment pour écrire des scripts d'administration système
- *Internet Explorer*. Il est alors utilisé au sein de pages HTML auxquelles il amène une certaine interactivité impossible à atteindre avec le seul langage HTML.
- *Internet Information Server* (IIS) le serveur Web de Microsoft sur NT/2000 et son équivalent *Personal Web Server* (PWS) sur Win9x. Dans ce cas, vbscript est utilisé pour faire de la programmation côté serveur web, technologie appelée ASP (*Active Server Pages*) par Microsoft.

On récupère le fichier d'installation à l'URL : *http://msdn.microsoft.com/scripting* et on suit les liens *Windows Script*. Sont installés :

- le conteneur *Windows Scripting Host*, conteneur permettant l'utilisation de divers langages de scripts, tels Vbscript et Javascript mais aussi d'autres tel PerlScript qui est amené avec Active Perl.
- un interpréteur VBscript
- un interpréteur Javascript

Présentons quelques tests rapides. Construisons le programme *vbscript* suivant :

```
' une classe
class personne
  Dim nom
  Dim age
End class

' création d'un objet personne
Set p1=new personne
With p1
  .nom="dupont"
  .age=18
End With

' affichage propriétés personne p1
With p1
  wscript.echo "nom=" & .nom
  wscript.echo "age=" & .age
End With
```

Ce programme utilise des objets. Appelons-le *objets.vbs* (le suffixe vbs désigne un fichier vbscript). Positionnons-nous sur le répertoire dans lequel il se trouve et exécutons-le :

```
E:\data\serge\windowsScripting\vbscript\poly\objets>cscript objets.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

nom=dupont
age=18
```

Maintenant construisons le programme javascript suivant qui utilise des tableaux :

```
// tableau dans un variant
// tableau vide
tableau=new Array();
affiche(tableau);
// tableau croît dynamiquement
for(i=0;i<3;i++){
  tableau.push(i*10);
}
// affichage tableau
affiche(tableau);
// encore
for(i=3;i<6;i++){
  tableau.push(i*10);
}
affiche(tableau);

// tableaux à plusieurs dimensions
WScript.echo("----------------------------");

tableau2=new Array();
for(i=0;i<3;i++){
  tableau2.push(new Array());
  for(j=0;j<4;j++){
    tableau2[i].push(i*10+j);
  }//for j
}// for i
affiche2(tableau2);

// fin
WScript.quit(0);

// ------------------------------------------------------
function affiche(tableau){
  // affichage tableau
  for(i=0;i<tableau.length;i++){
    WScript.echo("tableau[" + i + "]=" + tableau[i]);
  }//for
}//function

// ------------------------------------------------------
function affiche2(tableau){
  // affichage tableau
  for(i=0;i<tableau.length;i++){
    for(j=0;j<tableau[i].length;j++){
      WScript.echo("tableau[" + i + "," + j + "]=" + tableau[i][j]);
    }// for j
  }//for i
}//function
```

Ce programme utilise des tableaux. Appelons-le *tableaux.js* (le suffixe js désigne un fichier javascript). Positionnons-nous sur le répertoire dans lequel il se trouve et exécutons-le :

```
E:\data\serge\windowsScripting\javascript\poly\tableaux>cscript tableaux.js
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

tableau[0]=0
tableau[1]=10
tableau[2]=20
tableau[0]=0
tableau[1]=10
tableau[2]=20
tableau[3]=30
tableau[4]=40
tableau[5]=50
----------------------------
tableau[0,0]=0
tableau[0,1]=1
tableau[0,2]=2
tableau[0,3]=3
tableau[1,0]=10
tableau[1,1]=11
tableau[1,2]=12
tableau[1,3]=13
tableau[2,0]=20
tableau[2,1]=21
tableau[2,2]=22
tableau[2,3]=23
```

Un dernier exemple en Perlscript pour terminer. Il faut avoir installé Active Perl pour avoir accès à Perlscript.

```
<job id="PERL1">
  <script language="PerlScript">
```

```
    # du Perl classique
    %dico=("maurice"=>"juliette","philippe"=>"marianne");
    @cles= keys %dico;
    for ($i=0;$i<=$#cles;$i++){
      $cle=$cles[$i];
       $valeur=$dico{$cle};
      $WScript->echo ("clé=".$cle.", valeur=".$valeur);
    }
    # du perlscript utilisant les objets Windows Script
    $dico=$WScript->CreateObject("Scripting.Dictionary");
    $dico->add("maurice","juliette");
    $dico->add("philippe","marianne");
    $WScript->echo($dico->item("maurice"));
    $WScript->echo($dico->item("philippe"));
  </script>
</job>
```

Ce programme montre la création et l'utilisation de deux dictionnaires : l'un à la mode Perl classique, l'autre avec l'objet *Scripting Dictionary* de Windows Script. Sauvegardons ce code dans le fichier *dico.wsf* (wsf est le suffixe des fichiers Windows Script). Positionnons-nous dans le dossier de ce programme et exécutons-le :

```
E:\data\serge\windowsScripting\perlscript\essais>cscript dico.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. Tous droits réservés.

clé=philippe, valeur=marianne
clé=maurice, valeur=juliette
juliette
marianne
```

Perlscript peut utiliser les objets du conteneur dans lequel il s'exécute. Ici c'était des objets du conteneur Windows Script. Dans le contexte de la programmation Web, les scripts VBscript, Javascript, Perlscript peuvent être exécutés soit au sein du navigateur IE, soit au sein d'un serveur PWS ou IIS. Si le script est un peu complexe, il peut être judicieux de le tester hors du contexte Web, au sein du conteneur Windows Script comme il a été vu précédemment. On ne pourra tester ainsi que les fonctions du script qui n'utilisent pas des objets propres au navigateur ou au serveur. Même avec cette restriction, cette possibilité reste intéressante car il est en général assez peu pratique de déboguer des scripts s'exécutant au sein des serveurs web ou des navigateurs.

## 7.7 JAVA

Java est disponible à l'URL : **http://www.sun.com** (cf début de ce document) et s'installe dans une arborescence qu'on appellera **<java>** qui contient les éléments suivants :

```
22/05/2002  05:51       <DIR>          .
22/05/2002  05:51       <DIR>          ..
22/05/2002  05:51       <DIR>          bin
22/05/2002  05:51       <DIR>          jre
07/02/2002  12:52                 8 277 README.txt
07/02/2002  12:52                13 853 LICENSE
07/02/2002  12:52                 4 516 COPYRIGHT
07/02/2002  12:52                15 290 readme.html
22/05/2002  05:51       <DIR>          lib
22/05/2002  05:51       <DIR>          include
22/05/2002  05:51       <DIR>          demo
07/02/2002  12:52            10 377 848 src.zip
11/02/2002  12:55       <DIR>          docs
```

Dans **bin**, on trouvera **javac.exe**, le compilateur Java et **java.exe** la machine virtuelle Java. On pourra faire les tests suivants :

1. Écrire le script suivant :

```
//programme Java affichant l'heure

import java.io.*;
import java.util.*;

public class heure{
  public static void main(String arg[]){
     // on récupère date & heure
     Date maintenant=new Date();
     // on affiche
     System.out.println("Il est "+maintenant.getHours()+
        ":"+maintenant.getMinutes()+":"+maintenant.getSeconds());
  }//main
```

```
}//class
```

2. Sauvegarder ce programme sous le nom *heure.java*. Ouvrir une fenêtre DOS. Se mettre dans le répertoire du fichier *heure.java* et le compiler :

```
D:\data\java\essais>c:\jdk1.3\bin\javac heure.java
Note: heure.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

   Dans la commande ci-dessus `c:\jdk1.3\bin\javac` doit être remplacé par le chemin exact du compilateur *javac.exe*. Vous devez obtenir dans le même répertoire que *heure.java* un fichier *heure.class* qui est le programme qui va maintenant être exécuté par la machine virtuelle *java.exe*.

3. Exécuter le programme :

```
D:\data\java\essais>c:\jdk1.3\bin\java heure
Il est 10:44:2
```

## 7.8 Serveur Apache

Nous avons vu que l'on pouvait obtenir le serveur Apache avec l'application EasyPhp. Pour l'avoir directement, on ira sur le site d'Apache : **http://www.apache.org.** L'installation crée une arborescence où on trouve tous les fichiers nécessaires au serveur. Appelons ***<apache>*** ce répertoire. Il contient une arborescence analogue à la suivante :

```
UNINST   ISU       118 805  23/06/00  17:09 Uninst.isu
HTDOCS         <REP>        23/06/00  17:09 htdocs
APACHE~1 DLL       299 008  25/02/00  21:11 ApacheCore.dll
ANNOUN~1             3 000  23/02/00  16:51 Announcement
ABOUT_~1            13 197  31/03/99  18:42 ABOUT_APACHE
APACHE   EXE        20 480  25/02/00  21:04 Apache.exe
KEYS                36 437  20/08/99  11:57 KEYS
LICENSE              2 907  01/01/99  13:04 LICENSE
MAKEFI~1 TMP        27 370  11/01/00  13:47 Makefile.tmpl
README               2 109  01/04/98   6:59 README
README   NT         3 223  19/03/99   9:55 README.NT
WARNIN~1 TXT           339  21/09/98  13:09 WARNING-NT.TXT
BIN            <REP>        23/06/00  17:09 bin
MODULES        <REP>        23/06/00  17:09 modules
ICONS          <REP>        23/06/00  17:09 icons
LOGS           <REP>        23/06/00  17:09 logs
CONF           <REP>        23/06/00  17:09 conf
CGI-BIN        <REP>        23/06/00  17:09 cgi-bin
PROXY          <REP>        23/06/00  17:09 proxy
INSTALL  LOG         3 779  23/06/00  17:09 install.log
```

`conf` dossier des fichiers de configuration d'Apache
`logs` dossier des fichiers de logs (suivi) d'Apache
`bin` les exécutables d'Apache

### 7.8.1 Configuration

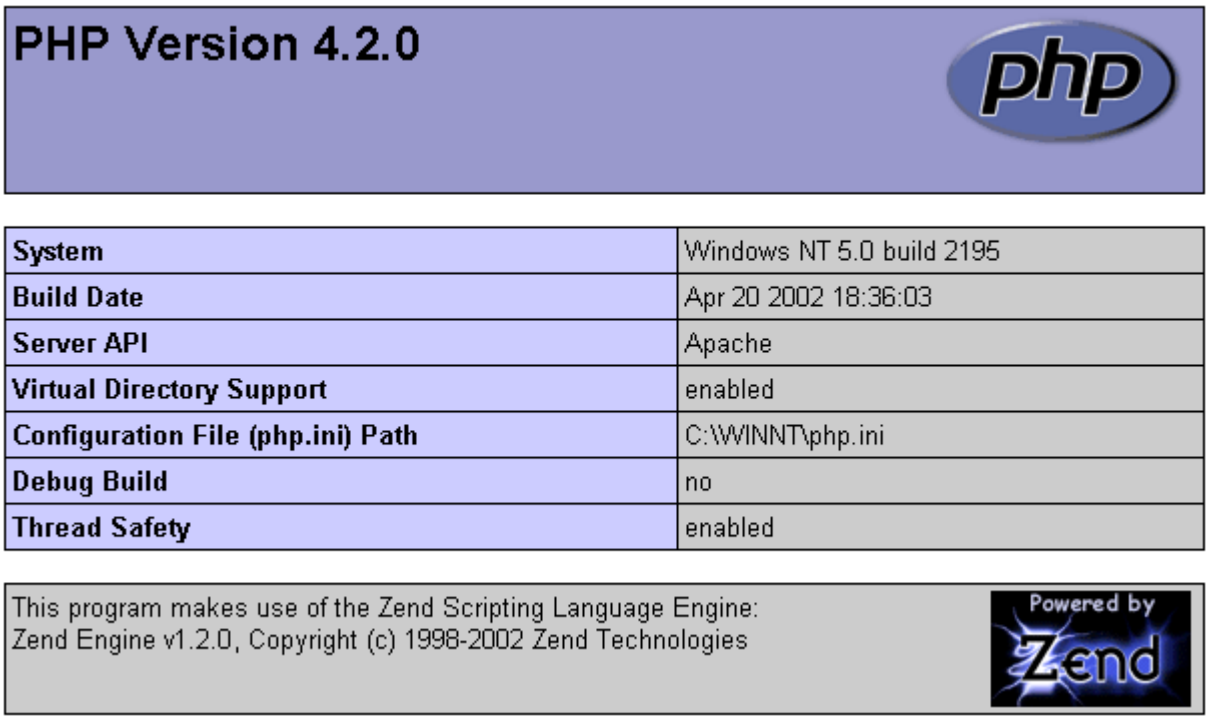
Dans le dossier ***<Apache>\conf***, on trouve les fichiers suivants : *httpd.conf, srm.conf, access.conf*. Dans les dernières versions d'Apache, les trois fichiers ont été réunis dans *httpd.conf*. Nous avons déjà présenté les points importants de ce fichier de configuration. Dans les exemples qui suivent c'est la version Apache d'EasyPhp qui a servi aux tests et donc son fichier de configuration. Dans celui-ci *DocumentRoot* qui désigne la racine de l'arborescence des pages Web est *e:\program files\easyphp\www*.

### 7.8.2 Lien PHP - Apache

Pour tester, créer le fichier **intro.php** avec la seule ligne suivante :

**<? phpinfo() ?>**

et le mettre à la racine des pages du serveur Apache(*DocumentRoot* ci-dessus). Demander l'URL **http://localhost/intro.php**. On doit voir une liste d'informations *php* :




Le script PHP suivant affiche l'heure. Nous l'avons déjà rencontré :

```
<?
  // time : nb de millisecondes depuis 01/01/1970
  // "format affichage date-heure
  // d: jour sur 2 chiffres
  // m: mois sur 2 chiffres
  // y : année sur 2 chiffres
  // H : heure 0,23
  // i : minutes
  // s: secondes
  print "Nous sommes le " . date("d/m/y H:i:s",time());
?>
```

Plaçons ce fichier texte à la racine des pages du serveur Apache (*DocumentRoot* ) et appelons-le date.*php*. Demandons avec un navigateur l'URL **http://localhost/date.php**. On obtient la page suivante :


Nous sommes le 18/07/02, 09:46:58

## 7.8.3 Lien PERL-APACHE

Il est fait grâce à une ligne de la forme : *ScriptAlias /cgi-bin/ "E:/Program Files/EasyPHP/cgi-bin/"* du fichier **<apache>\conf\httpd.conf**. Sa syntaxe est **ScriptAlias /cgi-bin/ "<cgi-bin>"** où **<cgi-bin>** est le dossier où on pourra placer des scripts CGI. CGI (Common Gateway Interface) est une norme de dialogue serveur WEB <--> Applications. Un client demande au serveur Web une page dynamique, c.a.d. une page générée par un programme. Le serveur WEB doit donc demander à un programme de générer la page. CGI définit le dialogue entre le serveur et le programme, notamment le mode de transmission des informations entre ces deux entités.

Si besoin est, modifiez la ligne *ScriptAlias /cgi-bin/ "<cgi-bin>"* et relancez le serveur Apache. Faites ensuite le test suivant :

1. Écrire le script :

```
#!c:\perl\bin\perl.exe
```

```
# script PERL affichant l'heure

# modules
use strict;

# programme
my ($secondes,$minutes,$heure)=localtime(time);
print <<FINHTML
Content-Type: text/html

<html>
 <head>
  <title>heure</title>
 </head>
 <body>
  <h1>Il est $heure:$minutes:$secondes</h1>
 </body>
FINHTML
;
```

2. Mettre ce script dans *<cgi-bin>\heure.pl* où *<cgi-bin>* est le dossier pouvant recevoir des scripts CGI (cf *httpd.conf*). La première ligne *#!c:\perl\bin\perl.exe* désigne le chemin de l'exécutable *perl.exe*. Le modifier si besoin est.
3. Lancer Apache si ce n'est fait
4. Demander avec un navigateur l'URL *http://localhost/cgi-bin/heure.pl*. On obtient la page suivante :


**Il est 9:52:55**

## 7.9 Le serveur PWS

### 7.9.1 Installation

Le serveur PWS (Personal Web Server) est une version personnelle du serveur IIS (Internet Information server) de Microsoft. Ce dernier est disponible sur les machines NT et 2000. Sur les machines win9x, PWS est normalement disponible avec le paquetage d'installation Internet Explorer. Cependant il n'est pas installé par défaut. Il faut prendre une installation personnalisée d'IE et demander l'installation de PWS. Il est par ailleurs disponibles dans le *NT 4.0 Option pack for Windows 95*.

### 7.9.2 Premiers tests

La racine des pages Web du serveur PWS est *lecteur:\inetpub\wwwroot* où *lecteur* est le disque sur lequel vous avez installé PWS. Nous supposons dans la suite que ce lecteur est D. Ainsi l'url *http://machine/rep1/page1.html* correspondra au fichier *d:\inetpub\wwwroot\rep1\page1.html*. Le serveur PWS interprète tout fichier de suffixe **.asp** (Active Server pages) comme étant un script qu'il doit exécuter pour produire une page HTML.

PWS travaille par défaut sur le port 80. Le serveur web Apache aussi... Il faut donc arrêter Apache pour travailler avec PWS si vous avez les deux serveurs. L'autre solution est de configurer Apache pour qu'il travaille sur un autre port. Ainsi dans le fichier httpd.conf de configuration d'Apache, on remplace la ligne Port 80 par Port 81, Apache travaillera désormais sur le port 81 et pourra être utilisé en même temps que PWS. Si PWS ayant été lancé, on demande l'URL http://localhost, on obtient une page analogue à la suivante :


### 7.9.3 Lien PHP - PWS

1. Ci-dessous on trouvera un fichier **.reg** destiné à modifier la base de registres. Double-cliquer sur ce fichier pour modifier la base. Ici la *dll* nécessaire se trouve dans *d:\php4* avec l'exécutable de php. Modifier si besoin est. Les \ doivent être doublés dans le chemin de la dll.

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\w3svc\parameters\Script Map]
".php"="d:\\php4\\php4isapi.dll"
```

2. Relancer la machine pour que la modification de la base de registres soit prise en compte.

3. Créer un dossier **php** dans *d:\inetpub\wwwroot* qui est la racine du serveur PWS. Ceci fait, activez PWS et prendre l'onglet « Avancé ». Sélectionner le bouton « Ajouter » pour créer un dossier virtuel :

   *Répertoire/Parcourir : d:\inetpub\wwwroot\php*
   *Alias : php*
   *Cocher la case exécuter.*

4. Valider le tout et relancer PWS. Mettre dans *d:\inetpub\wwwroot\php* le fichier **intro.php** ayant la seule ligne suivante :

**<? phpinfo() ?>**

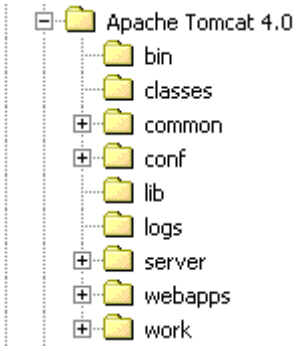
5. Demander au serveur PWS l'URL *http://localhost/php/intro.php*. On doit voir la liste d'informations php déjà présentées avec Apache.

## 7.10 Tomcat : servlets Java et pages JSP (Java Server Pages)

Tomcat est un serveur Web permettant de générer des pages HTML grâce à des servlets (programmes Java exécutés par le serveur web) où des pages JSP (Java Server Pages), pages mélangeant code Java et code HTML. C'est l'équivalent des pages ASP (Active Server Pages) du serveur IIS/PWS de Microsoft où là on mélange code VBScript ou Javascript avec du code HTML.

### 7.10.1 Installation

Tomcat est disponible à l'URL : **http://jakarta.apache.org.** On récupère un fichier .exe d'installation. Lorsqu'on lance ce programme, il commence par indiquer quel JDK il va utiliser. En effet Tomcat a besoin d'un JDK pour s'installer et ensuite compiler et exécuter les servlets Java. Il faut donc que vous ayez installé un JDK Java avant d'installer Tomcat. Le JD le plus récent est conseillé. L'installation va créer une arborescence <tomcat> :




consiste simplement à décompresser cette archive dans un répertoire. Prenez un répertoire ne contenant dans son chemin que des noms sans espace (pas par exemple "Program Files"), ceci parce qu'il y a un bogue dans le processus d'installation de Tomcat. Prenez par exemple C:\tomcat ou D:\tomcat. Appelons ce répertoire **<tomcat>**. On y trouvera dedans un dossier appelé **jakarta-tomcat** et dans celui-ci l'arborescence suivante :

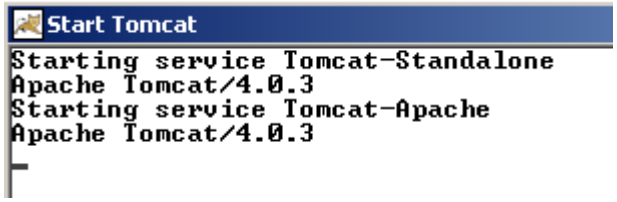
```
LOGS            <REP>          15/11/00   9:04 logs
LICENSE               2 876    18/04/00  15:56 LICENSE
CONF            <REP>          15/11/00   8:53 conf
DOC             <REP>          15/11/00   8:53 doc
LIB             <REP>          15/11/00   8:53 lib
SRC             <REP>          15/11/00   8:53 src
WEBAPPS         <REP>          15/11/00   8:53 webapps
BIN             <REP>          15/11/00   8:53 bin
WORK            <REP>          15/11/00   9:04 work
```

## 7.10.2 Démarrage/Arrêt du serveur Web Tomcat

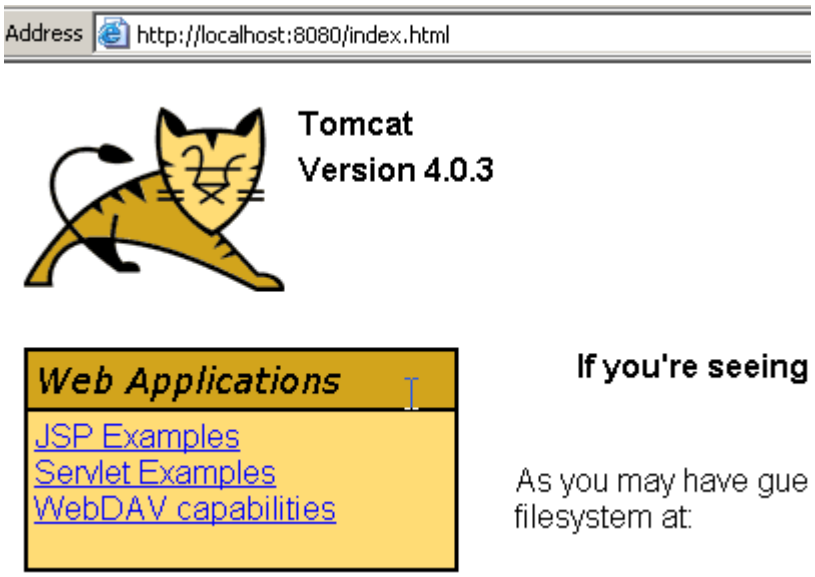
Tomcat est un serveur Web comme l'est Apache ou PWS. Pour le lancer, on dispose de liens dans le menu des programmes :

`Start Tomcat` pour lancer Tomcat
`Stop Tomcat` pour l'arrêter

Lorsqu'on lance Tomcat, une fenêtre Dos s'affiche avec le contenu suivant :




On peut mettre cette fenêtre Dos en icône. Elle restera présente pendant tant que Tomcat sera actif. On peut alors passer aux premiers tests. Le serveur Web Tomcat travaille sur le port 8080. Une fois Tomcat lancé, prenez un navigateur Web et demandez l'URL **http://localhost:8080**. Vous devez obtenir la page suivante :




Suivez le lien **Servlet Examples** :


Cliquez sur le lien *Execute* de *RequestParameters* puis sur celui de *Source*. Vous aurez un premier aperçu de ce qu'est une servlet Java. Vous pourrez faire de même avec les liens sur les pages JSP.

Pour arrêter Tomcat, on utilisera le lien *Stop Tomcat* dans le menu des programmes.

## 7.11 Jbuilder

Jbuilder est un environnement de développement d'applications Java. Pour construire des servlets Java où il n'y a pas d'interfaces graphiques, il n'est pas indispensable d'avoir un tel environnement. Un éditeur de textes et un JDK font l'affaire. Seulement JBuilder apporte avec lui quelques plus par rapport à la technique précédente :

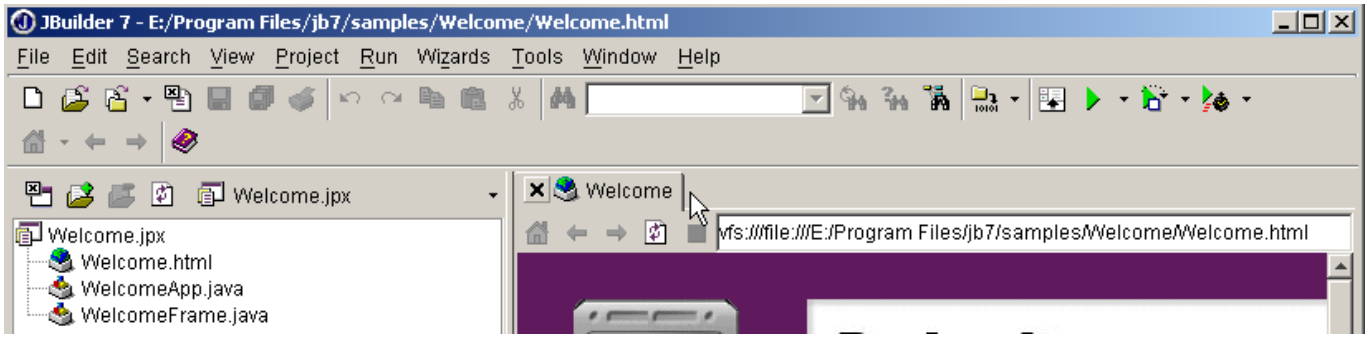
- facilité de débogage : le compilateur signale les lignes erronées d'un programme et il est facile de s'y positionner
- suggestion de code : lorsqu'on utilise un objet Java, JBuilder donne en ligne la liste des propriétés et méthodes de celui-ci. Cela est très pratique lorsqu'on sait que la plupart des objets Java ont de très nombreuses propriétés et méthodes qu'il est difficile de se rappeler.

On trouvera JBuilder sur le site **http://www.borland.com/jbuilder**. Il faut remplir un formulaire pour obtenir le logiciel. Une clé d'activation est envoyée par mél. Pour installer JBuilder 7, il a par exemple été procédé ainsi :

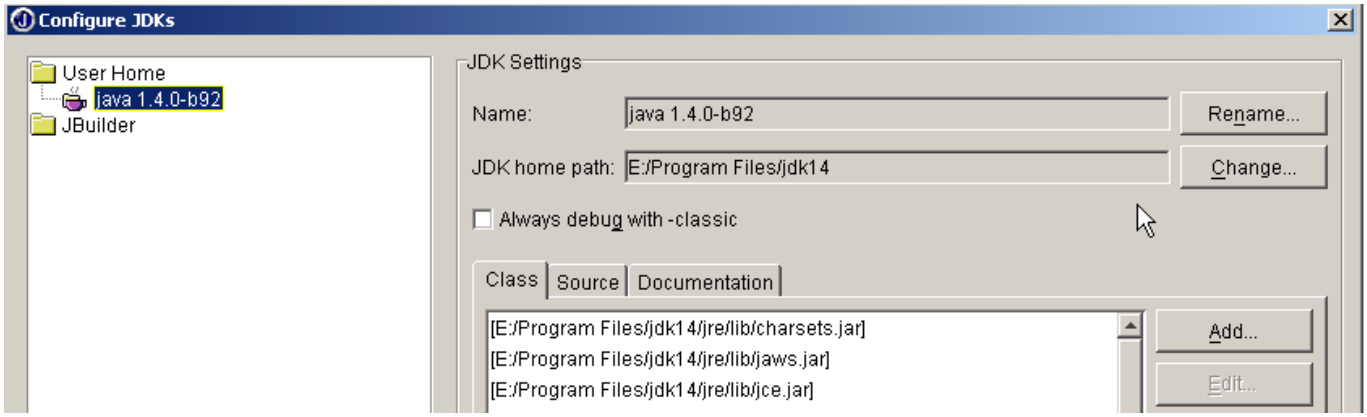
- trois fichiers zip ont été obtenus : pour l'application, pour la documentation, pour les exemples. Chacun de ces zip fait l'objet d'un lien séparé sur le site de JBuilder.
- on a installé d'abord l'application, puis la documentation et enfin les exemples
- lorsqu'on lance l'application la première fois, une clé d'activation est demandée : c'est celle qui vous a été envoyée par mél. Dans la version 7, cette clé est en fait la totalité d'un fichier texte que l'on peut placer, par exemple, dans le dossier d'installation de JB7. Au moment où la clé est demandée, on désigne alors le fichier en question. Ceci fait, la clé ne sera plus redemandée.

Il y a quelques configurations utiles à faire si on veut utiliser JBuilder pour construire des servlets Java. En effet, la version dite Jbuilder personnel est une version allégée qui ne vient notamment pas avec toutes les classes nécessaires pour faire du développement web en Java. On peut faire en sorte que JBuilder utilise les bibliothèques de classes amenées par Tomcat. On procède ainsi :

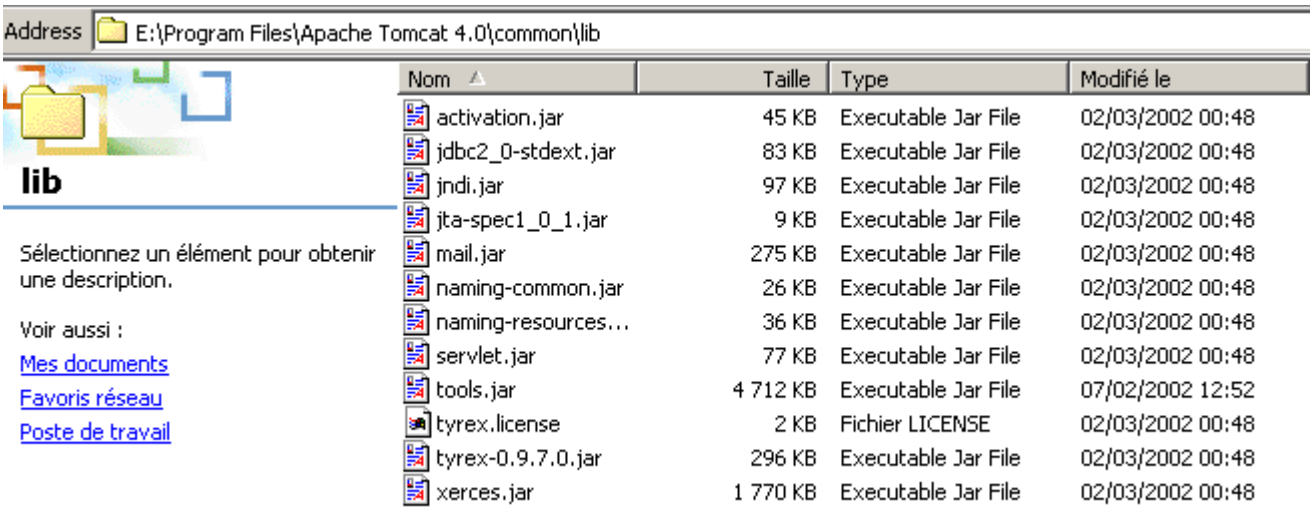
- lancer JBuilder



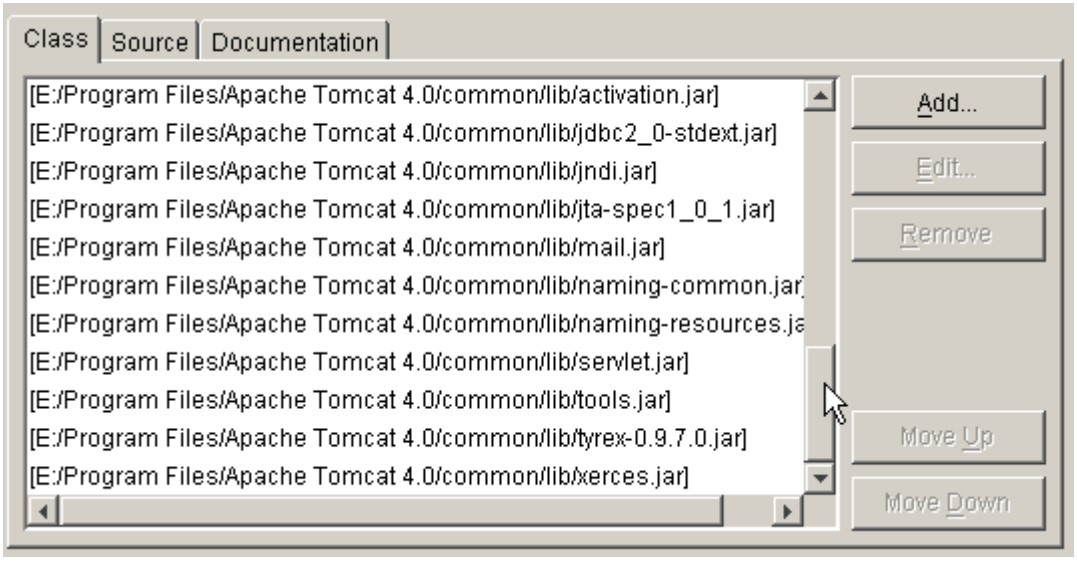

- activer l'option *Tools/Configure JDKs*




Dans la partie *JDK Settings* ci-dessus, on a normalement dans le champ *Name* un JDK 1.3.1. Si vous avez un JDK plus récent, utilisez le bouton *Change* pour désigner le répertoire d'installation de ce dernier. Ci-dessus, on a désigné le répertoire *E:\Program Files\jdk14* où avait installé un JDK 1.4. Désormais, JBuilder utilisera ce JDK pour ses compilations et exécutions. Dans la partie (Class, Siurce, Documentation) on a la liste de toutes les bibliothèques de classes qui seront explorées par JBuilder, ici les classes du JDK 1.4. Les classes de celui-ci ne suffisent pas pour faire du développement web en Java. Pour ajouter d'autres bibliothèques de classes on utilise le bouton *Add* et on désigne les fichiers *.jar* supplémentaires que l'on veut utiliser. Les fichiers .jar sont des bibliothèques de classes. Tomcat 4.x amène avec lui toutes les bibliothèques de classes nécessaires au développement web. Elles se trouvent dans *<tomcat>\common\lib* où *<tomcat>* est le répertoire d'installation de Tomcat :




Avec le bouton *Add*, on va ajouter ces bibliothèques, une à une, à la liste des bibliothèques explorées par JBuilder :




A partir de maintenant, on peut compiler des programmes java conformes à la norme J2EE, notamment les servlets Java. Jbuilder ne sert qu'à la compilation, l'exécution étant ultérieurement assurée par Tomcat selon des modalités expliquées dans le cours.

# 8. Code source de programmes

## 8.1 Le client TCP générique

Beaucoup de services créés à l'origine de l'Internet fonctionnent selon le modèle du serveur d'écho étudié précédemment : les échanges client-serveur se font pas échanges de lignes de texte. Nous allons écrire un client tcp générique qui sera lancé de la façon suivante : **java cltTCPgenerique serveur port**

Ce client TCP se connectera sur le port *port* du serveur *serveur*. Ceci fait, il créera deux threads :

1. un thread chargé de lire des commandes tapées au clavier et de les envoyer au serveur
2. un thread chargé de lire les réponses du serveur et de les afficher à l'écran

Pourquoi deux threads? Chaque service TCP-IP a son protocole particulier et on trouve parfois les situations suivantes :

- le client doit envoyer plusieurs lignes de texte avant d'avoir une réponse
- la réponse d'un serveur peut comporter plusieurs lignes de texte

Aussi la boucle envoi d'une unique ligne au serveur - réception d'une unique ligne envoyée par le serveur ne convient-elle pas toujours. On va donc créer deux boucles dissociées :

- une boucle de lecture des commandes tapées au clavier pour être envoyées au serveur. L'utilisateur signalera la fin des commandes avec le mot clé *fin*.
- une boucle de réception et d'affichage des réponses du serveur. Celle-ci sera une boucle infinie qui ne sera interrompue que par la fermeture du flux réseau par le serveur ou par l'utilisateur au clavier qui tapera la commande *fin*.

Pour avoir ces deux boucles dissociées, il nous faut deux threads indépendants. Montrons un exemple d'excécution où notre client tcp générique se connecte à un service SMTP (SendMail Transfer Protocol). Ce service est responsable de l'acheminement du courrier électronique à leurs destinataires. Il fonctionne sur le port 25 et a un protocole de dialogue de type échanges de lignes de texte.

```
Dos>java clientTCPgenerique istia.univ-angers.fr 25
Commandes :
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
help
<-- 502 5.3.0 Sendmail 8.11.6 -- HELP not implemented
mail from: machin@univ-angers.fr
<-- 250 2.1.0 machin@univ-angers.fr... Sender ok
rcpt to: serge.tahe@istia.univ-angers.fr
<-- 250 2.1.5 serge.tahe@istia.univ-angers.fr... Recipient ok
data
<-- 354 Enter mail, end with "." on a line by itself
Subject: test

ligne1
ligne2
ligne3
.
<-- 250 2.0.0 g4D6bks25951 Message accepted for delivery
quit
<-- 221 2.0.0 istia.univ-angers.fr closing connection
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

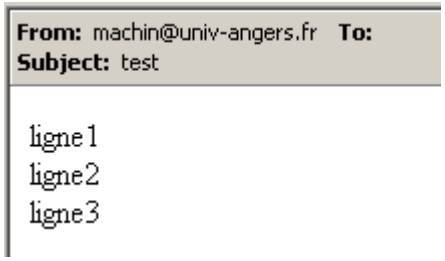
Commentons ces échanges client-serveur :

- le service SMTP envoie un message de bienvenue lorsqu'un client se connecte à lui :

```
<-- 220 istia.univ-angers.fr ESMTP Sendmail 8.11.6/8.9.3; Mon, 13 May 2002 08:37:26 +0200
```

- certains services ont une commande *help* donnant des indications sur les commandes utilisables avec le service. Ici ce n'est pas le cas. Les commandes SMTP utilisées dans l'exemple sont les suivantes :
    - **mail from:** *expéditeur*, pour indiquer l'adresse électronique de l'expéditeur du message
    - **rcpt to:** *destinataire*, pour indiquer l'adresse électronique du destinataire du message. S'il y a plusieurs destinataires, on ré-émet autant de fois que nécessaire la commande **rcpt to:** pour chacun des destinataires.
    - **data** qui signale au serveur SMTP qu'on va envoyer le message. Comme indiqué dans la réponse du serveur, celui-ci est une suite de lignes terminée par une ligne contenant le seul caractère point. Un message peut avoir des entêtes séparés du corps du message par une ligne vide. Dans notre exemple, nous avons mis un sujet avec le mot clé **Subject:**
- une fois le message envoyé, on peut indiquer au serveur qu'on a terminé avec la commande **quit**. Le serveur ferme alors la connexion réseau. Le thread de lecture peut détecter cet événement et s'arrêter.
- l'utilisateur tape alors **fin** au clavier pour arrêter également le thread de lecture des commandes tapées au clavier.

Si on vérifie le courrier reçu, nous avons la chose suivante (Outlook) :



On remarquera que le service SMTP ne peut détecter si un expéditeur est valide ou non. Aussi ne peut-on jamais faire confiance au champ *from* d'un message. Ici l'expéditeur *machin@univ-angers.fr* n'existait pas.

Ce client tcp générique peut nous permettre de découvrir le protocole de dialogue de services internet et à partir de là construire des classes spécialisées pour des clients de ces services. Découvrons le protocole de dialogue du service POP (Post Office Protocol) qui permet de retrouver ses méls stockés sur un serveur. Il travaille sur le port 110.

```
Dos> java clientTCPgenerique istia.univ-angers.fr 110
Commandes :
<-- +OK Qpopper (version 4.0.3) at istia.univ-angers.fr starting.
help
<-- -ERR Unknown command: "help".
user st
<-- +OK Password required for st.
pass monpassword
<-- +OK st has 157 visible messages (0 hidden) in 11755927 octets.
list
<-- +OK 157 visible messages (11755927 octets)
<-- 1 892847
<-- 2 171661
...
<-- 156 2843
<-- 157 2796
<-- .
retr 157
<-- +OK 2796 octets
<-- Received: from lagaffe.univ-angers.fr (lagaffe.univ-angers.fr [193.49.144.1])
<--     by istia.univ-angers.fr (8.11.6/8.9.3) with ESMTP id g4D6wZs26600;
<--     Mon, 13 May 2002 08:58:35 +0200
<-- Received: from jaume ([193.49.146.242])
<--     by lagaffe.univ-angers.fr (8.11.1/8.11.2/GeO20000215) with SMTP id g4D6wSd37691;
<--     Mon, 13 May 2002 08:58:28 +0200 (CEST)
...
<-- ------------------------------------------------------------------------
<-- NOC-RENATER2                  Tl.  : 0800 77 47 95
<-- Fax : (+33) 01 40 78 64 00 ,  Email : noc-r2@cssi.renater.fr
<-- ------------------------------------------------------------------------
<--
<-- .
quit
<-- +OK Pop server at istia.univ-angers.fr signing off.
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Les principales commandes sont les suivantes :

- **user** *login*, où on donne son login sur la machine qui détient nos méls
- **pass** *password*, où on donne le mot de passe associé au login précédent
- **list**, pour avoir la liste des messages sous la forme numéro, taille en octets
- **retr** *i*, pour lire le message n° i
- **quit**, pour arrêter le dialogue.

Découvrons maintenant le protocole de dialogue entre un client et un serveur Web qui lui travaille habituellement sur le port 80 :

```
Dos> java clientTCPgenerique istia.univ-angers.fr 80
Commandes :
GET /index.html HTTP/1.0

<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

```
<--
<-- <head>
<-- <meta http-equiv="Content-Type"
<-- content="text/html; charset=iso-8859-1">
<-- <meta name="GENERATOR" content="Microsoft FrontPage Express 2.0">
<-- <title>Bienvenue a l'ISTIA - Universite d'Angers</title>
<-- </head>
....
<-- face="Verdana"> - Dernire mise  jour le <b>10 janvier 2002</b></font></p>
<-- </body>
<-- </html>
<--
[fin du thread de lecture des réponses du serveur]
fin
[fin du thread d'envoi des commandes au serveur]
```

Un client Web envoie ses commandes au serveur selon le schéma suivant :

*commande1*
*commande2*
...
*commanden*
*[ligne vide]*

Ce n'est qu'après avoir reçu la ligne vide que le serveur Web répond. Dans l'exemple nous n'avons utilisé qu'une commande :

```
GET /index.html HTTP/1.0
```

qui demande au serveur l'URL **/index.html** et indique qu'il travaille avec le protocole HTTP version 1.0. La version la plus récente de ce protocole est 1.1. L'exemple montre que le serveur a répondu en renvoyant le contenu du fichier *index.html* puis qu'il a fermé la connexion puisqu'on voit le thread de lecture des réponses se terminer. Avant d'envoyer le contenu du fichier *index.html*, le serveur web a envoyé une série d'entêtes terminée par une ligne vide :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

La ligne <*html*> est la première ligne du fichier */index.html*. Ce qui précède s'appelle des entêtes HTTP (HyperText Transfer Protocol). Nous n'allons pas détailler ici ces entêtes mais on se rappellera que notre client générique y donne accès, ce qui peut être utile pour les comprendre. La première ligne par exemple :

```
<-- HTTP/1.1 200 OK
```

indique que le serveur Web contacté comprend le protocole HTTP/1.1 et qu'il a bien trouvé le fichier demandé (200 OK), 200 étant un code de réponse HTTP. Les lignes

```
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
```

disent au client qu'il va recevoir 11251 octets représentant du texte HTML (HyperText Markup Language) et qu'à la fin de l'envoi, la connexion sera fermée.

On a donc là un client tcp très pratique. Il fait sans doute moins que le programme *telnet* que nous avons utilisé précédemment mais il était intéressant de l'écrire nous-mêmes. Le programme du client tcp générique est le suivant :

```
// paquetages importés
import java.io.*;
import java.net.*;

public class clientTCPgenerique{
```

```
  // reçoit en paramètre les caractéristiques d'un service sous la forme
  // serveur port
  // se connecte au service
  // crée un thread pour lire des commandes tapées au clavier
  // celles-ci seront envoyées au serveur
  // crée un thread pour lire les réponses du serveur
  // celles-ci seront affichées à l'écran
  // le tout se termine avec la commande fin tapée au clavier

  // variable d'instance
  private static Socket client;

  public static void main(String[] args){

    // syntaxe
    final String syntaxe="pg serveur port";

    // nombre d'arguments
    if(args.length != 2)
      erreur(syntaxe,1);

    // on note le nom du serveur
    String serveur=args[0];

    // le port doit être entier >0
    int port=0;
    boolean erreurPort=false;
    Exception E=null;
    try{
      port=Integer.parseInt(args[1]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

    client=null;
    // il peut y avoir des problèmes
    try{
      // on se connecte au service
      client=new Socket(serveur,port);
    }catch(Exception ex){
      // erreur
      erreur("Impossible de se connecter au service ("+ serveur
        +","+port+"), erreur : "+ex.getMessage(),3);
      // fin
      return;
    }//catch

    // on crée les threads de lecture/écriture
    new ClientSend(client).start();
    new ClientReceive(client).start();

    // fin thread main
    return;
  }// main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe

class ClientSend extends Thread {
  // classe chargée de lire des commandes tapées au clavier
  // et de les envoyer à un serveur via un client tcp passé en paramètre

  private Socket client; // le client tcp

  // constructeur
  public ClientSend(Socket client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    PrintWriter OUT=null;    // flux d'écriture réseau
    BufferedReader IN=null;   // flux clavier
```

```
    String commande=null;    // commande lue au clavier

    // gestion des erreurs
    try{
      // création du flux d'écriture réseau
      OUT=new PrintWriter(client.getOutputStream(),true);
       // création du flux d'entrée clavier
       IN=new BufferedReader(new InputStreamReader(System.in));
      // boucle saisie-envoi des commandes
      System.out.println("Commandes : ");
      while(true){
        // lecture commande tapée au clavier
        commande=IN.readLine().trim();
        // fini ?
        if (commande.toLowerCase().equals("fin")) break;
        // envoi commande au serveur
        OUT.println(commande);
        // commande suivante
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("Envoi : L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      OUT.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[Envoi : fin du thread d'envoi des commandes au serveur]");
  }//run
}//classe

class ClientReceive extends Thread{
  // classe chargée de lire les lignes de texte destinées à un
  // client tcp passé en paramètre

  private Socket client; // le client tcp

  // constructeur
  public ClientReceive(Socket client){
    // on note le client tcp
    this.client=client;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    BufferedReader IN=null;    // flux lecture réseau
    String réponse=null;   // réponse serveur

    // gestion des erreurs
    try{
      // création du flux lecture réseau
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      // boucle lecture lignes de texte du flux IN
      while(true){
        // lecture flux réseau
        réponse=IN.readLine();
        // flux fermé ?
        if(réponse==null) break;
        // affichage
        System.out.println("<-- "+réponse);
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("Réception : L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      IN.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[Réception : fin du thread de lecture des réponses du serveur]");
  }//run
}//classe
```

## 8.2 Le serveur Tcp générique

Maintenant nous nous intéressons à un serveur

- qui affiche à l'écran les commandes envoyées par ses clients

- leur envoie comme réponse les lignes de texte tapées au clavier par un utilisateur. C'est donc ce dernier qui fait office de serveur.

Le programme est lancé par : **java serveurTCPgenerique portEcoute**, où *portEcoute* est le port sur lequel les clients doivent se connecter. Le service au client sera assuré par deux threads :

- un thread se consacrant exclusivement à la lecture des lignes de texte envoyées par le client
- un thread se consacrant exclusivement à la lecture des réponses tapées au clavier par l'utilisateur. Celui-ci signalera par la commande **fin** qu'il clôt la connexion avec le client.

Le serveur crée deux threads par client. S'il y a n clients, il y aura 2n threads actifs en même temps. Le serveur lui ne s'arrête jamais sauf par un Ctrl-C tapé au clavier par l'utilisateur. Voyons quelques exemples.

Le serveur est lancé sur le port 100 et on utilise le client générique pour lui parler. La fenêtre du client est la suivante :

```
E:\data\serge\MSNET\c#\réseau\client tcp générique> java clientTCPgenerique localhost 100
Commandes :
commande 1 du client 1
<-- réponse 1 au client 1
commande 2 du client 1
<-- réponse 2 au client 1
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

Les lignes commençant par <-- sont celles envoyées du serveur au client, les autres celles du client vers le serveur. La fenêtre du serveur est la suivante :

```
Dos> java serveurTCPgenerique 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
```

Les lignes commençant par <-- sont celles envoyées du client au serveur. Les lignes *N :* sont les lignes envoyées du serveur au client n° N. Le serveur ci-dessus est encore actif alors que le client 1 est terminé. On lance un second client pour le même serveur :

```
Dos> java clientTCPgenerique localhost 100
Commandes :
commande 3 du client 2
<-- réponse 3 au client 2
fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du thread de lecture des réponses du serveur]
[fin du thread d'envoi des commandes au serveur]
```

La fenêtre du serveur est alors celle-ci :

```
Dos> java serveurTCPgenerique 100
Serveur générique lancé sur le port 100
Thread de lecture des réponses du serveur au client 1 lancé
1 : Thread de lecture des demandes du client 1 lancé
<-- commande 1 du client 1
réponse 1 au client 1
1 : <-- commande 2 du client 1
réponse 2 au client 1
1 : [fin du Thread de lecture des demandes du client 1]
fin
[fin du Thread de lecture des réponses du serveur au client 1]
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- commande 3 du client 2
réponse 3 au client 2
```

```
2 : [fin du Thread de lecture des demandes du client 2]
fin
[fin du Thread de lecture des réponses du serveur au client 2]
^C
```

Simulons maintenant un serveur web en lançant notre serveur générique sur le port 88 :

```
Dos> java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
```

Prenons maintenant un navigateur et demandons l'URL *http://localhost:88/exemple.html*. Le navigateur va alors se connecter sur le port *88* de la machine *localhost* puis demander la page */exemple.html* :


Regardons maintenant la fenêtre de notre serveur :

```
Dos>java serveurTCPgenerique 88
Serveur générique lancé sur le port 88
Thread de lecture des réponses du serveur au client 2 lancé
2 : Thread de lecture des demandes du client 2 lancé
<-- GET /exemple.html HTTP/1.1
<-- Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/msword, */*
<-- Accept-Language: fr
<-- Accept-Encoding: gzip, deflate
<-- User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705; .NET CLR 1.0.2
914)
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
```

On découvre ainsi les entêtes HTTP envoyés par le navigateur. Cela nous permet de découvrir peu à peu le protocole HTTP. Lors d'un précédent exemple, nous avions créé un client Web qui n'envoyait que la seule commande GET. Cela avait été suffisant. On voit ici que le navigateur envoie d'autres informations au serveur. Elles ont pour but d'indiquer au serveur quel type de client il a en face de lui. On voit aussi que les entêtes HTTP se terminent par une ligne vide.

Elaborons une réponse à notre client. L'utilisateur au clavier est ici le véritable serveur et il peut élaborer une réponse à la main. Rappelons-nous la réponse faite par un serveur Web dans un précédent exemple :

```
<-- HTTP/1.1 200 OK
<-- Date: Mon, 13 May 2002 07:30:58 GMT
<-- Server: Apache/1.3.12 (Unix)  (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21
<-- Last-Modified: Wed, 06 Feb 2002 09:00:58 GMT
<-- ETag: "23432-2bf3-3c60f0ca"
<-- Accept-Ranges: bytes
<-- Content-Length: 11251
<-- Connection: close
<-- Content-Type: text/html
<--
<-- <html>
```

Essayons de donner une réponse analogue :

```
...
<-- Host: localhost:88
<-- Connection: Keep-Alive
<--
2 : HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
```

```
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
2 :     <center>
2 :       <h2>Reponse du serveur generique</h2>
2 :     </center>
2 :    </body>
2 : </html>
2 : fin
L'erreur suivante s'est produite : Impossible de lire les données de la connexion de transport.
[fin du Thread de lecture des demandes du client 2]
[fin du Thread de lecture des réponses du serveur au client 2]
```

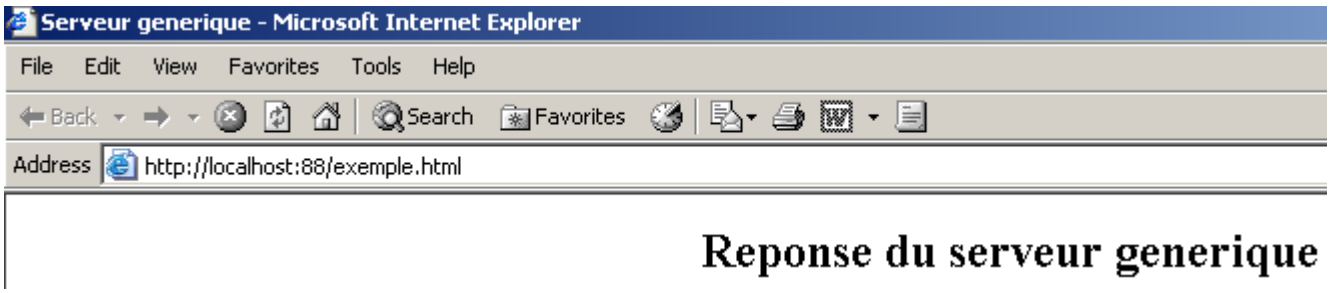
Les lignes commençant par *2 :* sont envoyées du serveur au client n° 2. La commande *fin* clôt la connexion du serveur au client. Nous nous sommes limités dans notre réponse aux entêtes HTTP suivants :

```
HTTP/1.1 200 OK
2 : Server: serveur tcp generique
2 : Connection: close
2 : Content-Type: text/html
2 :
```

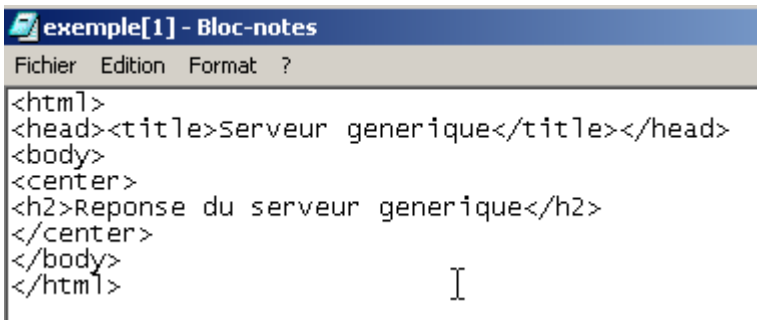
Nous ne donnons pas la taille du fichier que nous allons envoyer (*Content-Length*) mais nous contentons de dire que nous allons fermer la connexion (*Connection: close*) après envoi de celui-ci. Cela est suffisant pour le navigateur. En voyant la connexion fermée, il saura que la réponse du serveur est terminée et affichera la page HTML qui lui a été envoyée. Cette dernière est la suivante :

```
2 : <html>
2 :   <head><title>Serveur generique</title></head>
2 :   <body>
2 :     <center>
2 :       <h2>Reponse du serveur generique</h2>
2 :     </center>
2 :    </body>
2 : </html>
```

L'utilisateur ferme ensuite la connexion au client en tapant la commande *fin*. Le navigateur sait alors que la réponse du serveur est terminée et peut alors l'afficher :




Si ci-dessus, on fait *View/Source* pour voir ce qu'a reçu le navigateur, on obtient :




c'est à dire exactement ce qu'on a envoyé depuis le serveur générique.

Le code du serveur tcp générique est le suivant :

```
// paquetages
import java.io.*;
import java.net.*;

public class serveurTCPgenerique{

  // programme principal
  public static void main (String[] args){

  // reçoit le port d'écoute des demandes des clients
  // crée un thread pour lire les demandes du client
  // celles-ci seront affichées à l'écran
  // crée un thread pour lire des commandes tapées au clavier
  // celles-ci seront envoyées comme réponse au client
  // le tout se termine avec la commande fin tapée au clavier

  final String syntaxe="Syntaxe : pg port";
  // variable d'instance
     // y-a-t-il un argument
   if(args.length != 1)
    erreur(syntaxe,1);

    // le port doit être entier >0
    int port=0;
    boolean erreurPort=false;
    Exception E=null;
    try{
      port=Integer.parseInt(args[0]);
    }catch(Exception e){
      E=e;
      erreurPort=true;
    }
    erreurPort=erreurPort || port <=0;
    if(erreurPort)
      erreur(syntaxe+"\n"+"Port incorrect ("+E+")",2);

   // on crée le servive d'écoute
  ServerSocket ecoute=null;
  int nbClients=0;  // nbre de clients traités
    try{
      // on crée le service
      ecoute=new ServerSocket(port);
      // suivi
      System.out.println("Serveur générique lancé sur le port " + port);

      // boucle de service aux clients
      Socket client=null;
      while (true){ // boucle infinie - sera arrêtée par Ctrl-C
        // attente d'un client
        client=ecoute.accept();

        // le service est assuré des threads séparés
        nbClients++;

        // on crée les threads de lecture/écriture
         new ServeurSend(client,nbClients).start();
         new ServeurReceive(client,nbClients).start();

        // on retourne à l'écoute des demandes
      }// fin while
    }catch(Exception ex){
      // on signale l'erreur
      erreur("L'erreur suivante s'est produite : " + ex.getMessage(),3);
    }//catch
  }// fin main

  // affichage des erreurs
  public static void erreur(String msg, int exitCode){
    // affichage erreur
    System.err.println(msg);
    // arrêt avec erreur
    System.exit(exitCode);
  }//erreur
}//classe

class ServeurSend extends Thread{
  // classe chargée de lire des réponses tapées au clavier
  // et de les envoyer à un client via un client tcp passé au constructeur

  Socket client;// le client tcp
  int numClient;  // n° de client

  // constructeur
  public ServeurSend(Socket client, int numClient){
    // on note le client tcp
    this.client=client;
    // et son n°
```

```
    this.numClient=numClient;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    PrintWriter OUT=null;  // flux d'écriture réseau
    String réponse=null;   // réponse lue au clavier
     BufferedReader IN=null; // flux clavier

    // suivi
    System.out.println("Thread de lecture des réponses du serveur au client "+ numClient + " lancé");
    // gestion des erreurs
    try{
      // création du flux d'écriture réseau
      OUT=new PrintWriter(client.getOutputStream(),true);
       // création du flux clavier
       IN=new BufferedReader(new InputStreamReader(System.in));
      // boucle saisie-envoi des commandes
      while(true){
        // identification client
        System.out.print("--> " + numClient + " : ");
        // lecture réponse tapée au clavier
        réponse=IN.readLine().trim();
        // fini ?
        if (réponse.toLowerCase().equals("fin")) break;
        // envoi réponse au serveur
        OUT.println(réponse);
        // réponse suivante
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
      OUT.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[fin du Thread de lecture des réponses du serveur au client "+ numClient+ "]");
  }//run
}//classe

class ServeurReceive extends Thread{
  // classe chargée de lire les lignes de texte envoyées au serveur
  // via un client tcp passé au constructeur

  Socket client;// le client tcp
  int numClient;  // n° de client

  // constructeur
  public ServeurReceive(Socket client, int numClient){
    // on note le client tcp
    this.client=client;
    // et son n°
    this.numClient=numClient;
  }//constructeur

  // méthode Run du thread
  public void run(){

    // données locales
    BufferedReader IN=null;   // flux lecture réseau
    String réponse=null;   // réponse serveur

    // suivi
    System.out.println("Thread de lecture des demandes du client "+ numClient + " lancé");
    // gestion des erreurs
    try{
      // création du flux lecture réseau
      IN=new BufferedReader(new InputStreamReader(client.getInputStream()));
      // boucle lecture lignes de texte du flux IN
      while(true){
        // lecture flux réseau
        réponse=IN.readLine();
        // flux fermé ?
        if(réponse==null) break;
        // affichage
        System.out.println("<-- "+réponse);
      }//while
    }catch(Exception ex){
      // erreur
      System.err.println("L'erreur suivante s'est produite : " + ex.getMessage());
    }//catch
    // fin - on ferme les flux
    try{
```

```
      IN.close();client.close();
    }catch(Exception ex){}
    // on signale la fin du thread
    System.out.println("[fin du Thread de lecture des demandes du client "+ numClient+"]");
  }//run
}//classe
```

# 9. JAVASCRIPT

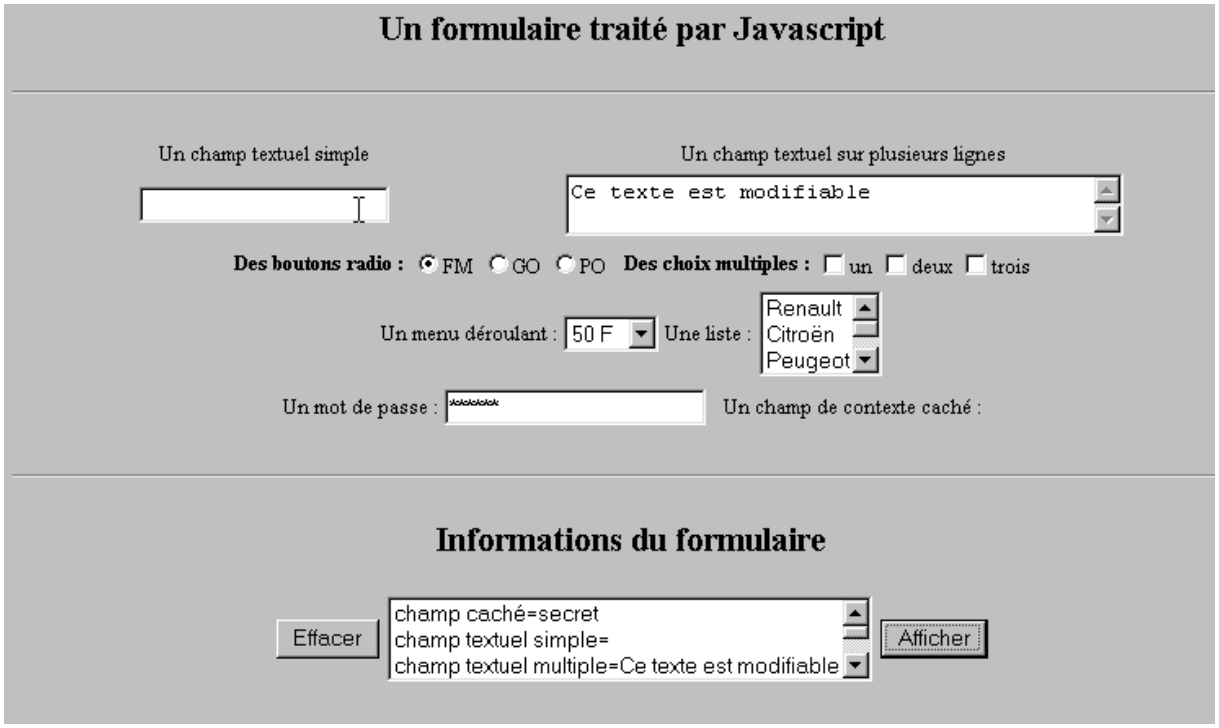
Nous montrons dans cette partie trois exemples d'utilisation de Javascript dans les pages WEB. Nous nous centrons sur la gestion des formulaires mais Javascript peut faire bien davantage.

## 9.1 Récupérer les informations d'un formulaire

L'exemple ci-dessous montre comment récupérer au sein du navigateur les données entrées par l'utilisateur au sein d'un formulaire. Cela permet en général de faire des pré-traitements avant de les envoyer au serveur.

### 9.1.1 Le formulaire

On a un formulaire rassemblant les composants les plus courants et d'un bouton **Afficher** permettant d'afficher les saisies faites par l'utilisateur.




### 9.1.2 Le code

```
<html>

  <head>
    <title>Un formulaire traité par Javascript</title>
    <script language="javascript">
      function afficher(){
        // affiche dans une liste les infos du formulaire
```

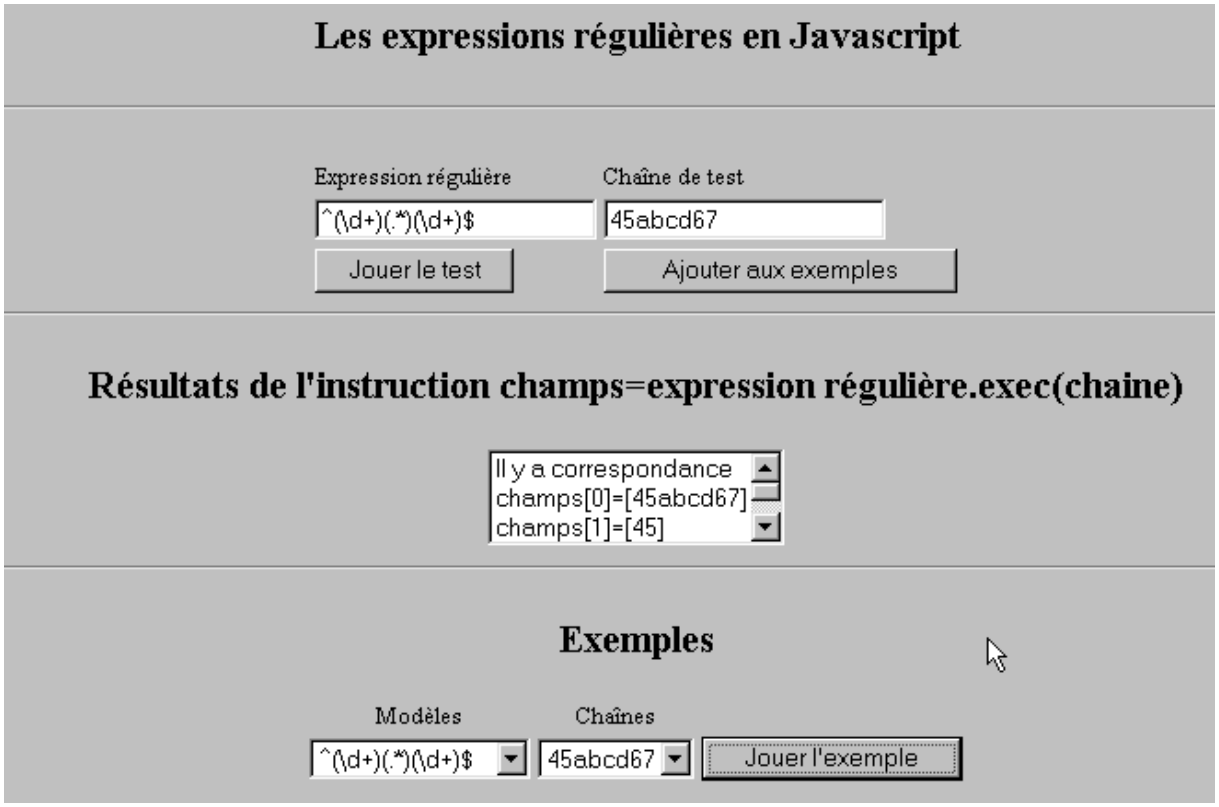
```
        // on efface d'abord
        effacerInfos();

        // on affiche la valeur des # champs
        with(document.frmExemple){
          // champ caché
          ecrire("champ caché="+cache.value);
          // champ textuel simple
          ecrire("champ textuel simple="+simple.value);
          // champ textuel multiple
          ecrire("champ textuel multiple="+lignes.value);
          // boutons radio
          for(i=0;i<radio.length;i++){
            texte="radio["+i+"]="+radio[i].value;
            if(radio[i].checked) texte+=", coché";
            ecrire(texte);
          }//for
          //  cases à cocher
          for(i=0;i<qcm.length;i++){
            texte="qcm["+i+"]="+qcm[i].value;
            if(qcm[i].checked) texte+=", coché";
            ecrire(texte);
          }//for
          //liste déroulante
          ecrire("index sélectionné dans le menu="+menu.selectedIndex);
          for(i=0;i<menu.length;i++){
            texte="menu["+i+"]="+menu.options[i].text;
            if(menu.options[i].selected) texte+=",sélectionné";
            ecrire(texte);
          }//for
          //liste à choix multiple
          for(i=0;i<lstVoitures.length;i++){
            texte="lstVoitures["+i+"]="+lstVoitures.options[i].text;
            if(lstVoitures.options[i].selected) texte+=",sélectionné";
            ecrire(texte);
          }//for
          //mot de passe
          ecrire("mot de passe="+passwd.value);
        }//with
      }//function

      function ecrire(texte){
        // écrit texte dans la liste des infos
        frmInfos.lstInfos.options[frmInfos.lstInfos.length]=new Option(texte);
      }//écrire

      function effacerInfos(){
        frmInfos.lstInfos.length=0;
      }//effacerInfos
    </script>
  </head>

  <body bgcolor="#C0C0C0" onload="afficher()">
    <center>
     <h2>Un formulaire traité par Javascript</h2>
    <hr>
    <form method="POST" name="frmExemple">
        <input type="hidden" name="cache" value="secret">
        <table border="0">
        <tr>
            <td align="center">Un champ textuel simple</td>
            <td align="center" width="100"> </td>
            <td align="center">Un champ textuel sur plusieurs lignes</td>
        </tr>
        <tr>
            <td align="center"><input type="text" size="20" name="simple"></td>
            <td align="center" width="100"> </td>
            <td align="center">
              <textarea name="lignes" rows="2" cols="40">Ce texte est modifiable</textarea>
            </td>
        </tr>
    </table>
    <table border="0">
      <tr>
       <td><strong>Des boutons radio :</strong></td>
        <td>
```

```
          <input type="radio" checked name="radio" value="FM">FM
        </td>
        <td>
          <input type="radio" name="radio" value="GO">GO
        </td>
        <td>
          <input type="radio" name="radio" value="PO">PO
        </td>
        <td> </td>
        <td><strong>Des choix multiples :</strong></td>
        <td>
          <input type="checkbox" name="qcm" value="un">un
        </td>
        <td>
          <input type="checkbox" name="qcm" value="deux">deux
        </td>
        <td>
          <input type="checkbox" name="qcm" value="trois">trois
        </td>
     </tr>
    </table>
    <table border="0">
      <tr>
       <td>Un menu déroulant : </td>
        <td>
          <select name="menu" size="1">
             <option>50 F</option>
              <option>60 F</option>
              <option>70 F</option>
              <option>100 F</option>
            </select>
        </td>
          <td>Une liste :</td>
           <td>
             <select name="lstVoitures" multiple size="3">
                <option>Renault</option>
                <option>Citroën</option>
                <option>Peugeot</option>
                <option>Fiat</option>
                <option>Audi</option>
              </select>
           </td>
        </tr>
    </table>
    <table border="0">
        <tr>
            <td>Un mot de passe : </td>
            <td><input type="password" size="21" name="passwd"></td>
            <td> </td>
            <td>Un champ de contexte caché : </td>
        </tr>
    </table>
    </form>
    <hr>
    <h2>Informations du formulaire</h2>
    <form name="frmInfos">
      <table>
        <tr>
          <td><input type="button" value="Effacer" onclick="effacerInfos()"></td>
          <td>
            <select name="lstInfos" multiple size="3">
            </select>
          </td>
          <td>
            <input type="button" name="cmdAfficher" value="Afficher" onclick="afficher()">
          </td>
        </tr>
    </form>
  </body>
</html>
```

## 9.2 Les expressions régulières en Javascript

Côté navigateur, Javascript peut être utilisé pour vérifier la validité des données entrées par l'utilisateur avant de les envoyer au serveur. Voici un programme de test de ces expressions régulières.

## 9.2.1 La page de test




## 9.2.2 Le code de la page

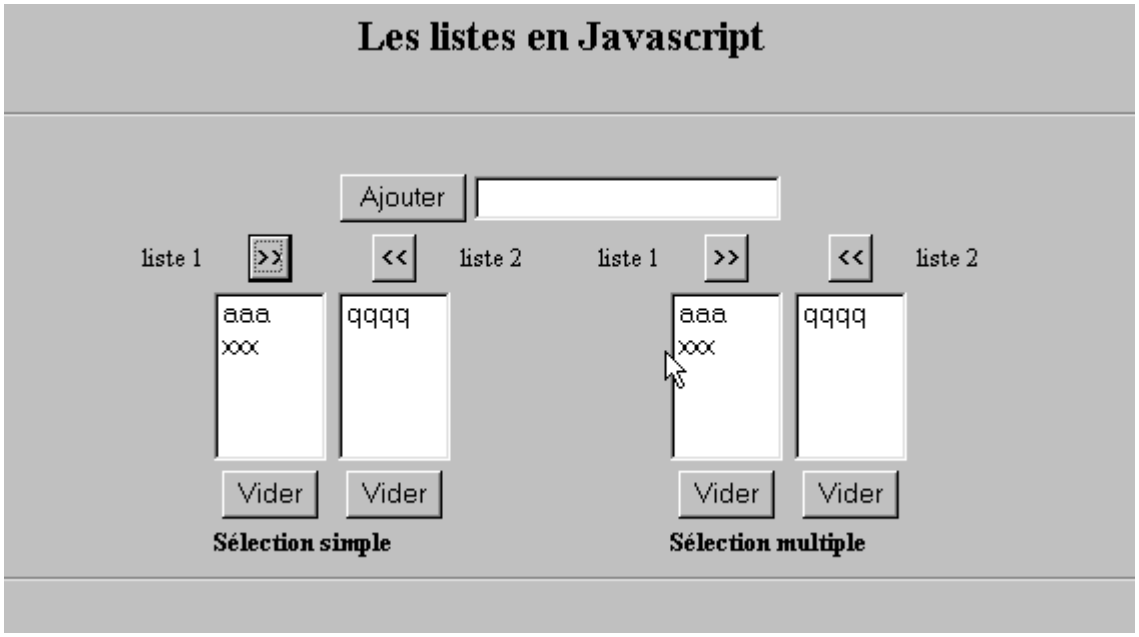
```
<html>

  <head>
    <title>Les expressions régulières en Javascript</title>
    <script language="javascript">
      function afficherInfos(){
        with(document.frmRegExp){
          // qq chose à faire ?
          if (! verifier()) return;
          // c'est bon - on efface les résultats précédents
          effacerInfos();
          // vérification du modèle
          modele=new RegExp(txtModele.value);
          champs=modele.exec(txtChaine.value);
          if(champs==null)
            // pas de correspondance entre modèle et chaîne
            ecrireInfos("pas de correspondance");
          else{
            // correspondance - on affiche les résultats obtenus
            ecrireInfos("Il y a correspondance");
            for(i=0;i<champs.length;i++)
              ecrireInfos("champs["+i+"]=["+champs[i]+"]");
          }//else
        }//with
      }//function

      function ecrireInfos(texte){
        // écrit texte dans la liste des infos
        document.frmRegExp.lstInfos.options[document.frmRegExp.lstInfos.length]=new Option(texte);
```

```
      }//écrire

      function effacerInfos(){
        frmRegExp.lstInfos.length=0;
      }//effacerInfos

      function jouer(){
        // teste le modèle contre la chaîne dans l'exemple choisi
        with(document.frmRegExp){
          txtModele.value=lstModeles.options[lstModeles.selectedIndex].text
          txtChaine.value=lstChaines.options[lstChaines.selectedIndex].text
          afficherInfos();
        }//with
      }//jouer

      function ajouter(){
        //ajoute le test courant aux exemples
        with(document.frmRegExp){
          // qq chose à faire ?
          if (! verifier()) return;
          // ajout
          lstModeles.options[lstModeles.length]=new Option(txtModele.value);
          lstChaines.options[lstChaines.length]=new Option(txtChaine.value);
          // raz saisies
          txtModele.value="";
          txtChaine.value="";
        }//with
      }//ajouter

      function verifier(){
        // vérifie que les champs de saisie sont non vides
        with(document.frmRegExp){
          champs=/^\s*$/.exec(txtModele.value);
          if(champs!=null){
            alert("Vous n'avez pas indiqué de modèle");
            txtModele.focus();
            return false;
          }//if
          champs=/^\s*$/.exec(txtChaine.value);
          if(champs!=null){
            alert("Vous n'avez pas indiqué de chaîne de test");
            txtChaine.focus();
            return false;
          }//if
          // c'est bon
          return true;
        }//with
      }//verifier
    </script>
  </head>

  <body bgcolor="#C0C0C0">
    <center>
     <h2>Les expressions régulières en Javascript</h2>
    <hr>
    <form name="frmRegExp">
      <table>
        <tr>
          <td>Expression régulière</td>
          <td>Chaîne de test</td>
        </tr>
        <tr>
          <td><input type="text" name="txtModele" size="20"></td>
          <td><input type="text" name="txtChaine" size="20"></td>
        </tr>
        <tr>
          <td>
            <input type="button" name="cmdAfficher" value="Jouer le test" onclick="afficherInfos()">
          </td>
          <td>
            <input type="button" name="cmdAjouter" value="Ajouter aux exemples" onclick="ajouter()">
          </td>
        </tr>
      </table>
      <hr>
      <h2>Résultats de l'instruction champs=expression régulière.exec(chaine)</h2>
      <table>
        <tr>
```

```
          <td>
            <select name="lstInfos" size="3">
            </select>
          </td>
        </tr>
      </table>
      <hr>
      <h2>Exemples</h2>
      <table>
 <tr>
          <td align="center">Modèles</td>
          <td align="center">Chaînes</td>
        </tr>
        <tr>
          <td>
            <select name="lstModeles" size="1">
              <option>^\d+$</option>
              <option>^(\d+) (\d+)$</option>
              <option>^(\d+)(.*)(\d+)$</option>
              <option>^(\d+)(\s+)(\d+)$</option>
            </select>
          </td>
          <td>
            <select name="lstChaines" size="1">
              <option>67</option>
              <option>56 84</option>
              <option>45abcd67</option>
              <option>45   67</option>
            </select>
          </td>
          <td>
            <input type="button" name="cmdJouer" value="Jouer l'exemple" onclick="jouer()">
          </td>
        </tr>
    </form>
  </body>
</html>
```

## 9.3 Gestion des listes en JavaScript

### 9.3.1 Le formulaire




### 9.3.2 Le code

```
<html>
```

```
<head>
  <title>Les listes en Javascript</title>

    <script language="javascript">
      // ajouter
      function ajouter(L1,L2,T){
        // ajoute la valeur du champ T aux listes L1,L2
          // qq chose à faire ?
          champs=/^\s*$/.exec(T.value);
          if(champs!=null){
            // le champ est vide
            alert("Vous n'avez pas indiqué la valeur à ajouter");
            txtElement.focus();
            return;
          }//if
          // on ajoute l'élément
          L1.options[L1.length]=new Option(T.value);
          L2.options[L2.length]=new Option(T.value);
          T.value="";
      }//ajouter

      //vider
      function vider(L){
        // vide la liste L
        L.length=0;
      }//vider

     //transfert
      function transfert(L1,L2,simple){
        //transfére dans L2 les éléments sélectionnés dans la liste L1

        // qq chose à faire ?
        // index de l'élément sélectionné dans L1
        index1=L1.selectedIndex;
        if(index1==-1){
          alert("Vous n'avez pas sélectionné d'élément");
          return;
        }//if
        // quel est le mode de sélection des éléments des listes
        if(simple){ // sélection simple
          element1=L1.options[index1].text;
          //ajout dans L2
          L2.options[L2.length]=new Option(element1);
          //suppression dans L1
          L1.options[index1]=null;
        }//simple
        if(! simple){ //sélection multiple
          //on parcourt la liste 1 en sens inverse
          for(i=L1.length-1;i>=0;i--){
            //élément sélectionné ?
            if(L1.options[i].selected){
              //on l'ajoute à L2
              L2.options[L2.length]=new Option(L1.options[i].text);
              //on le supprime de L1
              L1.options[i]=null;
            }//if
          }//for i
        }//if ! simple
      }//transfert
   </script>
  </head>

  <body bgcolor="#C0C0C0">
    <center>
     <h2>Les listes en Javascript</h2>
    <hr>
    <form name="frmListes">
      <table>
        <tr>
          <td>
            <input type="button" name="cmdAjouter" value="Ajouter"
onclick="ajouter(lst1A,lst1B,txtElement)">
          </td>
          <td>
            <input type="text" name="txtElement">
          </td>
        </tr>
      </table>
```

```
      <table>
        <tr>
          <td align="center">liste 1</td>
          <td align="center"><input type="button" value=">>" onclick="transfert(lst1A,lst2A,true)"</td>
          <td align="center"><input type="button" value="<<" onclick="transfert(lst2A,lst1A,true)"</td>
          <td align="center">liste 2</td>
          <td width="30"></td>
          <td align="center">liste 1</td>
          <td align="center"><input type="button" value=">>" onclick="transfert(lst1B,lst2B,false)"</td>
          <td align="center"><input type="button" value="<<" onclick="transfert(lst2B,lst1B,false)"</td>
          <td align="center">liste 2</td>
        </tr>
         <tr>
          <td></td>
          <td align="center">
            <select name="lst1A" size="5">
            </select>
          </td>
          <td align="center">
            <select name="lst2A" size="5">
            </select>
          </td>
          <td></td>
          <td></td>
          <td></td>
          <td align="center">
            <select name="lst1B" size="5" multiple >
            </select>
          </td>
          <td align="center">
            <select name="lst2B" size="5" multiple>
            </select>
          </td>

        </tr>
        <tr>
          <td></td>
          <td align="center"><input type="button" value="Vider" onclick="vider(lst1A)"</td>
          <td align="center"><input type="button" value="Vider" onclick="vider(lst2A)"</td>
          <td></td>
          <td></td>
          <td></td>
          <td align="center"><input type="button" value="Vider" onclick="vider(lst1B)"</td>
          <td align="center"><input type="button" value="Vider" onclick="vider(lst2B)"</td>
          <td></td>
        </tr>
        <tr>
          <td></td>
          <td colspan="2"><strong>Sélection simple</strong></td>
          <td></td>
          <td></td>
          <td></td>
          <td colspan="2"><strong>Sélection multiple</strong></td>
          <td></td>
        </tr>

      </table>
      <hr>
    </form>
  </body>
</html>
```