Dans cette étude, je vais montré tout ce que peuvent réaliser les pages JSP, mais pas ce qui serait souhaitable de faire. Pour bien maîtriser ces concepts, il est préférable d'avoir traiter les chapitres antérieurs, notamment les deux derniers, savoir : les servlets techniques avancées et les descripteurs de déploiement. Je ne reviendrais donc pas sur certaines notions, comme la structure d'une application Web ainsi que sur la gestion et l'exploitation des bases de données. Je suppose que tout cela est déjà connu. Je reprendrais les mêmes exemples que ceux mis au point dans l'étude sur les servlets. C'est juste pour montrer un autre aspect, une autre approche. Ce n'est pas la meilleure solution, loin de là. Le but est de nous familiariser avec toutes les techniques propres aux pages JSP. Nous verrons d'ailleurs dans une prochaine étude que l'idéal est de mélanger les technologies dans les applications Web avec donc des servlets et des pages JSP. Ces dernières étant plutôt spécialisées pour la présentation alors que les premières sont plutôt prévues pour le traitement des requêtes.

Pour en savoir plus sur les servlets (techniques avancées) et sur le descripteur de déploiement.

Introduction

Les JavaServer Pages, ou JSP, servent, comme les servlets, à créer du contenu Web de manière dynamique. Ces deux types de composants représentent à eux seuls un très fort pourcentage du contenu des applications Web.

Créer des servlets consiste à construire des composants Java capables de produire du code HTML. Dans de nombreux cas, cela fonctionne sans problème. Toutefois, il n'est pas facile, pour les personnes chargées de concevoir l'aspect visuel des pages Web, de manipuler du code Java, auquel elles n'ont probablement pas été formées. C'est la raison d'être des JavaServer Pages. Les JSP sont des documents de type texte, contenant du code HTML ainsi que des scriptlets (et/ou des expressions), c'est-à-dire des morceaux de code Java.

Les développeur des pages JSP peuvent mélanger du contenu statique et du contenu dynamique. Ces pages étant basées sur du code HTML ou XML, elles peuvent être créées et manipulées par du personnel non technique. Un développeur Java peut être en charge de la création des scriptlets (et/ou des expressions) qui s'interfaceront avec les sources de données ou effectueront des calculs permettant la génération de code dynamique.

```
<html>
<head>
<title>Enregistrement des coordonnées</title>
</head>
<body bgcolor="orange" text="green">
<h2>Enregistrement des coordonnées effectué</h2>
<hr width="75%">
<p><b>Bonjour
  <%= request.getParameter("titre") %>
  <%= request.getParameter("prenom") %>          Expressions
  <%= request.getParameter("nom") %>
  <%
      int âge = Integer.parseInt(request.getParameter("age"));
      String message = "Vous êtes un";
      if (âge>0 && âge<12) message += " enfant.";
      if (âge>=12 && âge<18) message += " adolescent.";
      if (âge>=18 && âge<60) message += " adulte.";
      if (âge>=60) message += "e personne du troisième âge.";
  %>
<p><%= message %></b>                            Scriptlets
</body>
</html>
```

Les pages JSP s'exécutent, en fait, sous la forme de servlets. Elles disposent du même cycle de vie. Elle sont donc compilées comme les servlets et sont donc plus rapides dans leur traitement. Elles disposent du même support pour la gestion des sessions. Elles peuvent également charger des JavaBeans et appeler leurs méthodes, accéder à des sources de données se trouvant sur des serveurs distants, ou effectuer des calculs complexes. Tout ce que peut faire une servlet une page JSP est capable de le réaliser.

Les pages JSP simplifient la création de pages générées dynamiquement sur un serveur HTTP, en utilisant une démarche opposée à celle des servlets. Au lieu d'écrire du code HTML dans le code Java d'une classe servlet, un développeur JSP écrit du code Java dans le code HTML d'un fichier JSP. Toutefois, dans la démarche de la plateforme J2EE, les servlets serviront plus à traiter les requêtes des clients alors que les pages jSP servirons plus à la présentation.

Dans la suite de ce chapitre, nous allons exploiter toutes les possibilités des pages JSP.

Les éléments JSP

Nous ne pouvons pas écrire du code Java n'importe où dans une page HTML. Nous avons besoin d'un moyen pour indiquer au serveur où s'arrête le code HTML et où commence le code Java. Pour cela la spécification JSP définit des balises, un peu comme pour le HTML ou le XML, qui peuvent être employées pour délimiter le code Java. Ces balises permettent de définir trois catégories d'éléments :

1. *les directives ;*
2. *les scripts ;*
3. *les actions.*

La spécification originale utilisait des éléments dont le format n'était pas compatible avec XML, c'est-à-dire qu'ils n'étaient pas conforme à la spécification de ce langage. La spécification JSP 1.2 a introduit une nouvelle syntaxe compatible XML. Nous exploiterons les deux écritures.

Les directives

Les directives sont des éléments fournissant au conteneur des informations relatives à la page. Il existe trois directives :

1. *page : <%@ page attributs %> ou en format XML <jsp:directive.page attributs />*
2. *include : <%@ include file = "..." %> ou en format XML <jsp:directive.include file = "..." />*

3. **taglib** : *(étudié au prochain chapitre).*

Voici ci-dessous la liste des attributs les plus fréquemment utilisés :

Directive	Attribut	Description
page	import	Liste les paquetages qui doivent être importés. De la même façon que pour les fichiers sources Java, il est nécessaire d'importer dans las pages JSP les paquetages de classes référencées dans le code. Si plusieurs paquetages sont importés, ils doivent être séparés par des virgules, par exemple : <%@ page import = "java.io.*, java.util.*" %> ou bien <jsp:directive.page import = "java.io.*, java.util.*" />
	session	Cet attribut peut prendre les valeurs true ou false. La valeur par défaut est true. Elle indique que la page fait partie d'une session. Si la valeur est false, la page n'aura accès à aucune information de session.
	isThreadSafe	Indique si la page peut être employée pour des accès simultanés. La valeur par défaut est true.
	info	La valeur de cet attribut peut être une chaîne quelconque décrivant le contenu de la page, ou sa fonction, son auteur, etc.
	errorPage	Indique l'URL de la page qui doit être renvoyée au client en cas d'erreur.
	isErrorPage	Cet attribut indique si la page est une page d'erreur. Sa valeur par défaut est false.
	contentType	Définit le type de contenu de la page. Il peut s'agier d'une simple indication du type, ou d'un type et d'un jeu de caractères. La valeur par défaut est text/html pour les pages contenant des balises de style JSP et text/xml pour celles contenant des balises de style XML.
	pageEncoding	Le jeu de caractères utilisés pour la page. La valeur par défaut est ISO-8859-1 (latin script) pour les pages de style JSP et UTF8 (encodage Unicode sur 8 bits) pour les pages de style XML.
include	file	Le nom du fichier à inclure à l'emplacement de la balise. Il peut s'agir d'une page HTML ou JSP, ou du fragment de page. La valeur doit être l'URI d'un fichier appartenant à la même application Web.

Une page JSP peut contenir plusieurs directives page. Comme indiqué dans le tableau, la directive include est employée pour inclure une autre page, ou un fragment de page, dans la page JSP. Il peut s'agir d'un en-tête ou d'un pied de page, ou de tout autre élément. Cette directive est utile chaque fois qu'un contenu standard doit être réutilisé dans plusieurs pages. L'inclusion a lieu avant la traduction de la page en code Java.

Nous verrons plus loin un moyen d'inclure du contenu de manière dynamique, au moment du traitement d'une requête.
.

Les scripts

Les éléments de script permettent de placer du code java dans les pages JSP. Il en existe trois formes :

1. les **déclarations** : *<%! déclaration %> ou en format XML <jsp:declaration>déclaration</jsp:declaration>*
2. les **scriptlets** : *<% fragment de code %> ou en format XML <jsp:scriptlet>fragment de code</jsp:scriptlet>*
3. les **expressions** : *<%= expression %> ou en format XML <jsp:expression>expression</jsp:expression>*

Les déclarations

La page JSP une fois compilée est traduite sous forme de servlet. Les servlets sont des classes comme les autres, et à ce titre, elles comportent des méthodes et des attributs. Il est également possible pour les pages JSP de posséder de tels attributs et de telles méthodes. Il suffit pour cela d'utiliser les déclarations.

Une déclaration doit être employée pour déclarer, et éventuellement pour initialiser un attribut ou une méthode Java. Par exemple, pour déclarer un vecteur, nous pouvous utiliser une des syntaxes suivantes :

```
<%! Vector v = new Vector( ) ; %>
ou
<jsp:declaration>Vector v = new Vector( ) ;</jsp:declaration>
```

Ce fragment de code déclare une variable v de type Vector et l'initialise en appelant le constructeur de cette classe. Toute les variables ainsi déclarées deviennent des attributs et sont donc accessibles dans toute la page.

Nous pouvons également définir des méthodes qui seront utilisées ensuite dans l'ensemble de la page, et ainsi éviter les répétitions :

```
<%!
        public int nombreMots(String chaîne) {
                return new StringTokenizer(chaîne).countTokens();
        }
%>
```

ou

```
<jsp:declaration>
        public int nombreMots(String chaîne) {
                return new StringTokenizer(chaîne).countTokens();
        }
</jsp:declaration>
```

Les attributs ou les méthodes ainsi déclarées peuvent être appelées par n'importe quel code (scriptlets) présent dans toute la page.
.

Les scriptlets

Les scriptlets contiennent des instructions Java. Ces instructions apparaissent dans le code Java produit lors de la traduction des pages JSP (sous forme de servlet), mais pas dans les réponses envoyées au client. Les scriptlets peuvent contenir n'importe quel code Java valide. Par exemple, pour répéter dix fois le mot "Bonjour !", nous pouvons utiliser la scriptlet suivante :

```
<%
        for (int i=0; i<10; i++) {
%>
Bonjour !
<%
        }
%>
```

Nous pouvons librement imbriquer le code Java et du contenu HTML. Tout ce qui se trouve entre les délimiteurs de scriptlets (<% et %>) est du code Java. Ce qui se trouve à l'extérieur est le contenu renvoyé au client.

Notez, dans cet exemple, que le code ne doit pas obligatoirement commencer et se terminer dans la même scriptlet. Cela permet de mélanger librement le code Java et l'écriture HTML.

Puisque les scriptlets peuvent contenir n'importe quelle instruction java, le code suivant est une scriptlet valide :

```
<%
        Vector v = new Vector( ) ;
        for (int i=0; i<10; i++)
                v[i] = i;
%>
```

La première ligne de code ressemble à la déclaration que nous avons présentée précédemment. Nous pouvons alors nous demander quel peut être la différence entre les scriptlets et les déclarations. En dépit de leur similitude, elles diffèrent sur les points suivants :

1. *Les scriptlets ne peuvent être employées pour définir des méthodes. Seules les déclarations permettent cela.*
2. *Les variables déclarées dans une déclaration sont des attributs, accessible dans toutes les scriptlets de la page.*
3. *Les variables déclarées dans une scriptlet sont des variables locales et ne sont donc visibles qu'à l'intérieur de la scriptlet dans laquelle elles sont définies.*

Les expressions

Les expressions sont utilisées pour renvoyer directement au client la valeur d'une variable, la valeur retour d'une méthode, ou même tout autre type d'expression Java. L'exemple suivant affiche le texte : Le nombre d'éléments dans cette phrase est 10 dans le navigateur :

```
Le nombre d'éléments dans cette phrase est <%= nombreMots("Le nombre d'éléments dans cette phrase est n") %>
```

ou

```
Le nombre d'éléments dans cette phrase est
<jsp:expression>
        nombreMots("Le nombre d'éléments de cette phrase est n")
</jsp:expression>
```

Toutes les expressions Java valides peuvent être employées. Une expression peut contenir un appel de méthodes, comme ci-dessus, le contenu d'une variable, une expression littérale, telle 2+2, ou encore une construction utilisant des mots clés Java, comme v instanceof Vector, ou une combinaison de ces quatres possibilités.

Notez également que les déclarations et les scriptlets contiennent des lignes de code Java et doivent donc être systématiquement terminées par des points-virgules. En revanche, les expressions ne doivent pas en comporter.

Les commentaires

Il est possible d'utiliser des commentaires HTML dans les pages JSP. Ces commentaires apparaissent dans la page renvoyée au client. Ils sont de la forme suivante :

```
<!-- Ce commentaire sera transmis au client. -->
```

Il existe également des commentaires JSP :

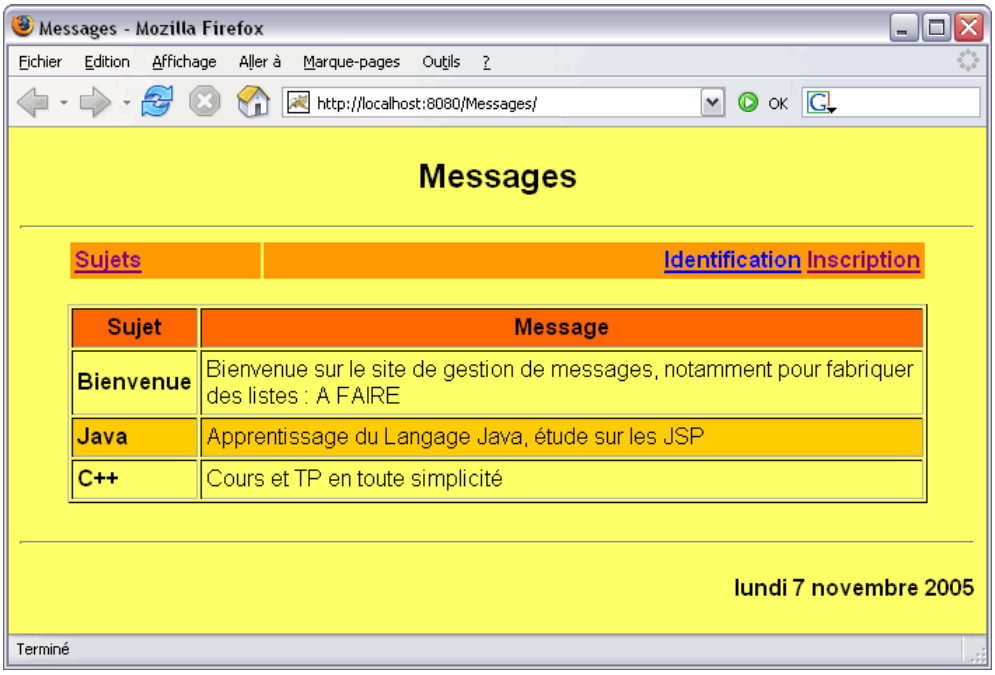
```
<%-- Ce commentaire Ne sera PAS transmis au navigateur client. --%>
```

Les données

Tout ce qui n'appartient pas à une directive, une déclaration, une scriptlet, une expression ou un commentaire JSP fait partie des données. C'est donc toute la partie HTML. Les données sont transmises au client comme si elles avaient été placées dans une page Web statique.

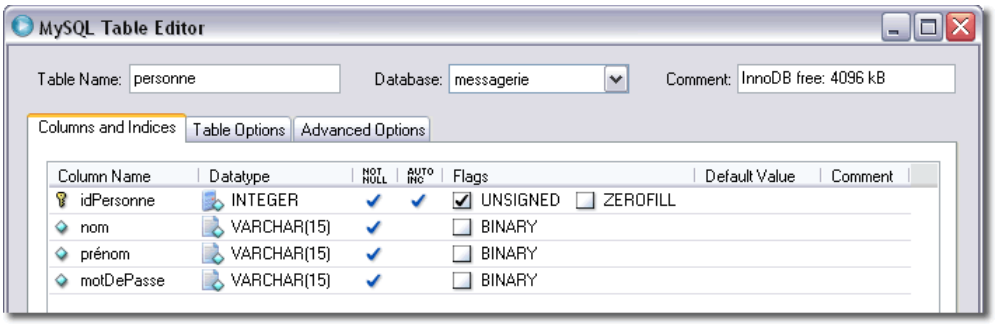
Exemple sur les directives et les scripts avec utilisation d'une base de données

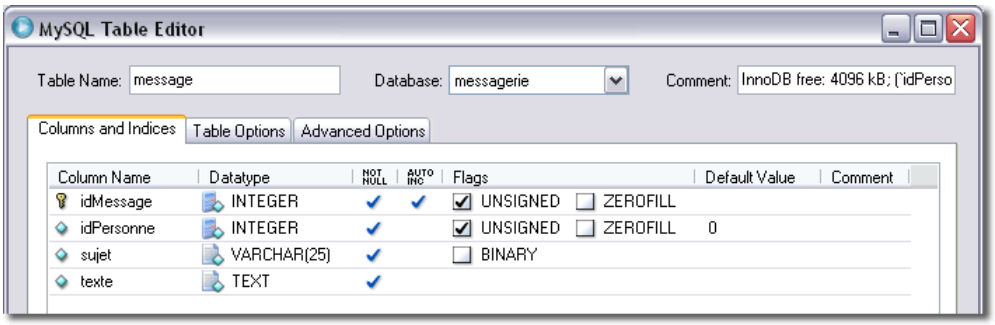
Nous allons mettre en oeuvre une application Web qui permet de gérer une petite messagerie rudimentaire interne, qui servira de liste de messages - A FAIRE. Chaque personne disposera de sa propre liste de messages. Toutefois, cette messagerie permet de délivrer des messages pour tout le monde, même pour ceux qui ne sont pas encore inscrit. Voici, ci-dessous la page d'accueil d'un tel site :



Base de données

*La base de données relative à cette application Web s'appelle **Messagerie**. Elle comporte deux tables en relations l'une avec l'autre. La première est la table **Personne** qui permet de recencer les personnes habilités à concevoir ou modifier leurs propres messages. La deuxième table est la table **Message** qui stocke l'ensemble des messages de l'application Web.*





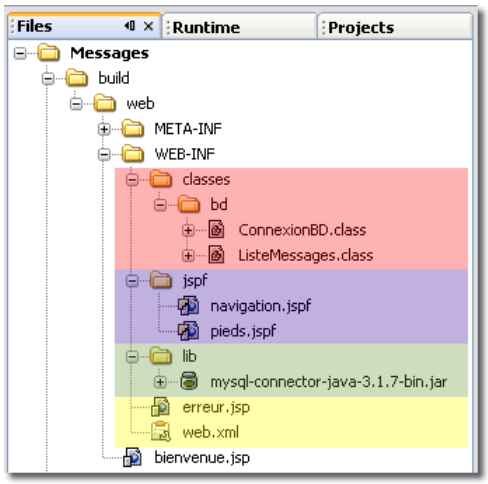
Voici le contenu de la table Message en relation avec notre page d'accueil :

idPersonne = 1 : Cette personne représente tout le monde.
.

Architecture de l'application Web

Chaque application Web doit être structurée d'une certaine façon. Dans le tableau ci-dessous vous trouverez le principe de cette architecture. Tout d'abord, vous devez fabriquez un répertoire qui contiendra tous les composants de l'application, ici <web>. Ce répertoire est ensuite découpé en deux zones : une partie publique qui correspond à la racine de ce répertoire, et une partie privée, donc non accessible directement de l'extérieur, qui se situe dans le sous-répertoire <WEB-INF>.

Dossiers	Type de fichier
webapp	*ressources Web publiques (pages accessibles directement)* **Pages statiques : <*.html>** **Pages dynamiques JSP : <*.jsp>** **Applets : <*.class>** **ici bienvenue.jsp**



	WEB-INF		*ressources Web privée (pages non accessibles directement)* Fichier de configuration de l'application WEB : <web.xml> Pages d'erreur : <*.jsp> Autres pages JSP : <*.jsp> ici web.xml et erreur.jsp
		classes	Emplacement des classes utilitaires et des JavaBeans (compilées): <*.class>. ici bd.ConnexionBD.class et bd.ListeMessages.class
		jspf	Fragment de pages : <*.jspf> ici navigation.jspf et pieds.jspf
		lib	Bibliothèques <*.jar> non standards comme les drivers JBDC. ici mysql-connector_java_3.1.7-bin.jar
		tlds	Bibliothèques de balises

Dans la partie privée, doit se trouver impérativement, le fichier <web.xml> qui donne toute l'organisation de l'application Web, et aussi dans quel cadre les documents privés peuvent être utilisés. Voici justement le descripteur de déploiement de notre application Web.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Liste des messages personnels</display-name>
  <welcome-file-list>
    <welcome-file>bienvenue.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

Pour l'instant, dans notre descripteur de déploiement que la page d'accueil indique que la page d'accueil s'appelle <bienvenue.jsp>. Il faut dire que sans cette précision, la page d'accueil devrait s'appeler <index.jsp>. Le descripteur de déploiement est justement utile pour configurer à notre convenance l'architecture de tous nos fichiers. Nous avons également précisé le nom d'affichage de l'application.

Classes utilitaires

Pour simplifier l'écriture des pages JSP, tout le code un petit peu plus compliqué doit être placé dans des classes utilitaires qui se trouveront, après compilation dans le sous-répertoire <classes> du répertoire <WEB-INF>. En effet, il est souvent préférable qu'il y ait le minimum de code Java dans les pages JSP, juste de quoi déclarer des classes déjà construites afin de faire directement appel à leurs méthodes. Le webmaster s'occupe essentiellement de la présentation, donc des pages JSP, alors que le développeur Java s'occupe de la mise en oeuvre des classes nécessaires à la bonne marche de l'application Web. Le développeur Java libére ainsi le webmaster de tout traveaux connexes et complexes.

Ici, nous développons deux classes qui sont en relation directement avec la base de données. La première, ConnexionBD, est la classe de base qui s'occupe des instructions relatives à la connexion avec la base de données messagerie, et met en place le résultat issue de la requête passée en argument de la méthode lire(). Une fois que l'objet de cette classe est créé et que la requête a été précisée au moyen de la méthode lire() (essentiellement des requêtes de type SELECT), il est possible de consulter successivement chacune des lignes de la table choisie par la requête grâce à la méthode suivant(). Après avoir consulté toutes les lignes de la table, vous pouvez interrompre la connexion au moyen de la méthode arrêt(). La méthode miseAJour(), quand à elle, sera utilisée lorsque nous désirerons modifier le contenu de la base de données, comme par exemple, lorsque nous voudrons inscrire un nouvel utilisateur. Il faudra proposer alors des requêtes adaptées à cette situation, comme par exemple la requête INSERT INTO.

ConnexionBD.java

```
package bd;

import java.sql.*;

public class ConnexionBD {
   private Connection connexion;
   private Statement instruction;
   protected ResultSet résultat;

   public ConnexionBD() {
      try {
         Class.forName("com.mysql.jdbc.Driver");
         connexion = DriverManager.getConnection("jdbc:mysql://localhost/messagerie", "root", "manu");
         instruction = connexion.createStatement();
      }
      catch (ClassNotFoundException ex) {
         System.err.println("Problème de pilote");
      }
      catch (SQLException ex) {
         System.err.println("Base de données non trouvée ou requête incorrecte");
      }
   }

   public void lire(String requête) {
       try {
          résultat = instruction.executeQuery(requête);
       }
       catch (SQLException ex) {
          System.err.println("Requête incorrecte "+requête);
       }
   }

   public void miseAJour(String requête) {
       try {
```

```
            instruction.executeUpdate(requête);
        }
        catch (SQLException ex) {
            System.err.println("Requête incorrecte "+requête);
        }
    }

    public boolean suivant() {
        try {
            return résultat.next();
        } catch (SQLException ex) {
            return false;
        }
    }

    public void arrêt() {
       try {
          connexion.close();
       }
       catch (SQLException ex) {
          System.err.println("Erreur sur l"arrêt de la connexion à la base de données");
       }
    }
}
```

La classe ListeMessages est plus spécialisée. Elle hérite d'ailleurs de la classe ConnexionBD. Elle s'intéresse plus particulièrement de la table message. Le constructeur fabrique la requête nécessaire afin de récupérer la liste des messages dédiée à une personne en particulier. Il est alors nécessaire d'avoir récupéré l'identification de cette personne avant d'utiliser cette classe. Deux méthodes supplémentaires ont été créée afin de récupérer les champs utiles à la présentation de ces messages, sujet() et texte().

ListeMessages.java

```
package bd;

import java.sql.SQLException;

public class ListeMessages extends ConnexionBD {
   public ListeMessages(int idPersonne) {
      lire("SELECT * FROM message WHERE idPersonne=\""+idPersonne+"\"");
   }

   public String sujet() {
       try {
          return résultat.getString("sujet");
       } catch (SQLException ex) {
          return "";
       }
   }

   public String texte() {
       try {
          return résultat.getString("texte");
       } catch (SQLException ex) {
          return "";
       }
   }
}
```

Ici, l'intérêt de mettre en oeuvre l'héritage permet de simplifier le travail. Chaque classe s'occupe essentiellement de ses propres problèmes. Nous pouvons même d'ailleurs travailler en équipe, chaque développeur s'intéresse alors qu'à une seule classe. Comme d'habitude, le mécanisme d'héritage permet d'avoir une maintenance plus facile.

Nous aurions pu construire des classes plus étoffées, notamment avec la dernière afin qu'elle puisse, par exemple, introduire de nouveau messages, effacer des messages, etc. Le but ici, est de montrer le fonctionnement des pages JSP sans avoir des classes longues et compliquées. La lecture de cette étude doit être la plus rapide possible.

Les pages JSP

Rentrons maintenant dans le sujet qui nous intéresse, savoir les pages JSP. Nous avons deux pages principales : bienvenue.jsp et erreur.jsp. Ces pages font appel, par l'intermédiaire de la directive include, à des fragments de page dont l'extention d'ailleurs et <.jspf>. Nous avons ici deux fragments de pages qui correspondent respectivement à l'en-tête (navigation.jspf) et au pieds de page (pieds.jspf).*

Il est d'usage, bien que cela ne soit pas obligatoire de placer les fragments de page dans le répertoire spécifique <jspf> de la zone privée <WEB-INF>.
.

---

bienvenue.jsp

```
1  <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
2  <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
3
4  <font face="Arial">
5  <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
6
7      <tr bgcolor="#FF6600">
8          <th>Sujet</th>
9          <th>Message</th>
10     </tr>
11     <%
12         ListeMessages listeMessages = new ListeMessages(1);
13         int ligne = 0;
```

```
14          while (listeMessages.suivant()) {
15      %>
16      <tr bgcolor="<%= ligne++ % 2 == 0 ? "#FFFF66" : "#FFCC00" %>">
17          <td><b><%= listeMessages.sujet() %></b></td>
18          <td><%= listeMessages.texte() %></td>
19      </tr>
20      <%
21          }
22          listeMessages.arrêt();
23      %>
24  </table></p>
25  </font>
26
27  <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

La page bienvenue.jsp est la page d'accueil du site (précision donnée, nous l'avons vu, par le descripteur de déploiement).

Ligne 1:

Elle commence par la directive **page**. Cette directive prend deux attributs :

1. *errorPage : le premier définit la page d'erreur (erreur.jsp), vers laquelle la requête sera redirigée en cas de problème lors du traitement de la page.*
2. *import : le second permet d'importer les classes Java nécessaires, ici les deux classes ConnexionBD et ListeMessages stockées dans le paquetage bd.*

Ligne 2:

La ligne suivante fait appel à la directive **include**. Cette directive insère, à partir de cette ligne, tout le code source implémenté dans la page **navigation.jspf**. Il s'agit juste d'un petit traitement de texte (**copier-coller**). Le compilateur prends tout le code qui se trouve dans la page désignée et le met à la place de la ligne de commande correspondant à l'appel de la directive. La ligne 25 effectue également ce type d'opération pour inclure le pied de page : **pieds.jspf**. Cette technique permet d'éviter la réécriture de code similaire. C'est très fréquent sur les sites où la partie haute et la partie basse de chaque page sont identiques.

Lignes 4 à 10 :

Nous trouvons maintenant tout simplement des lignes de code HTML ordinaires.

Lignes 11 à 15 et 20 à 23:

Dans ces lignes sont insérées des **scriptlets**. Dans ces zones, nous écrivons toutes les lignes codes Java nécessaires à la bonne présentation de la page. C'est souvent utile pour mettre en place des itératives comme c'est le cas ici. Dans la première sriptlet, nous commençons par déclarer l'objet **listeMessages** en faisant appel au constructeur correspondant. Nous passons en paramètre de ce constructeur la valeur **1** qui indique que nous désirons connaître la liste des messages prévues pour tous les utilisateurs. Dans la première **scriptlet** se trouve également le début de l'itérative. L'itérative se termine ensuite dans la deuxième **scriptlet**. En fait, c'est comme si nous avions une seule **scriptlet** (même bloc) à l'intérieur de laquelle se trouve du code HTML ordinaire (**Lignes 15 à 18**). Ainsi la variable **ligne** ainsi que l'objet **listeMessages** qui sont déclarés à l'intérieur de cette **scriptlet** sont accessibles à toute la zone, mais uniquement à cette zone. Il s'agit donc de variables locales dont la durée de vie correspont à la zone.

Dans la mesure du possible, il est préférable de déclarer des objets dans des zones les plus limitées possibles. Dans le cas de l'objet listeMessages, nous aurions pu déclarer cet objet comme un attribut de la classe représentant la page JSP, c'est-à-dire, en utilisant la déclaration <%! %>. Si effectivement vous faites cette expérience et si vous cliquez plusieurs fois sur le lien sujet, vous remarquerez que la liste des messages ne s'affiche au complet que la première fois. Le problème, c'est que les navigateurs possèdent un proxy interne qui empêche de refabriquer systématiquement la page JSP. Ce qui est très bien d'ailleurs. Mais le problème, c'est qu'avec une déclaration <%! %>, l'objet listeMessages a une durée de vie qui correspond à la page entière, et comme la page n'est pas détruite, l'objet non plus n'est pas détruit et donc lorsque nous faisons appel à la méthode suivant() de cet objet, nous sommes déjà sur la dernière ligne de la base de données. En conclusion, utiliser au maximum les variables locales.

Lignes 16 à 19:

Ces lignes comportent à la fois du code HTML ordinaire et des **expressions** (**<%= %>**). Ces dernières permettent, d'une part d'afficher le texte respectivement du **sujet** et du **texte** de chaque message, d'autre part de choisir la couleur de fond de la ligne du tableau avec la même couleur une fois sur deux.

Attention, dans une expression le point-virgule n'est pas nécessaire.
.

---

navigation.jspf

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head><title>Messages</title></head>
  <body bgcolor="#FFFF66">
    <font face="Arial">
        <h2 align="center">Messages</h2>
        <hr>
        <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
            <tr bgcolor="#FF9900">
                <th align="left"><a href="bienvenue.jsp">Sujets</th>
                <th align="right">
                    <a href="#">Identification</a>
                    <a href="nouvelutilisateur.jsp">Inscription</a>
                </th>
            </th>
        </table>
```

Cette page navigation.jspf est un fragment de page (<*.jspf>). Elle s'intègre donc dans d'autres pages. En conséquences, nous pouvons avoir du code HTML classique qui semble ne pas être terminé, comme c'est le cas ici avec, par exemple, la balise <html> qui ne comporte pas de balise de fin </html>. C'est la page qui copie ce fragment de code qui doit s'en occuper, où éventuellement, un autre fragment de page.

Les fragments de pages permettent d'éviter de réécrire systématiquement les mêmes choses. L'intérêt également, c'est que chaque page dispose du coup de très peu de lignes de code, ce qui facilite grandement la maintenance. Par exemple, si un jour vous devez modifier l'en-tête des pages, vous allez directement et uniquement sur la page représentant cette en-tête.

La page navigation.jspf comporte uniquement du code HTML ordinaire. Cette page sera ultérieurement modifiée afin qu'elle prenne en compte l'utilisateur identifié.

---

pieds.jspf

```
<%@page import="java.util.Date, java.text.DateFormat" %>

<%!
    DateFormat formatDate = DateFormat.getDateInstance(DateFormat.FULL);
 %>
        <br><hr>
        <h4 align="right"><%= formatDate.format(new Date()) %></h4>
     </font>
   </body>
</html>
```

Ce fragment est le pied de page qui spécifie la date en format complet français sur chacune des pages du site. Cette fois-ci, nous utilisons une déclaration. L'objet formatDate aura donc une durée de vie qui correspond à la page. Ici, l'écriture s'en trouve simplifiée. Par ailleurs, la date du jour dure toute une journée donc l'objet formatDate peut avoir une durée de vie relativement longue.

Attention : Dans une déclaration, la syntaxe java doit être respectée au complet, c'est-à-dire qu'il faut également le point-virgule.
.

erreur.jsp

```
<%@page isErrorPage="true"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<%@include file="/WEB-INF/jspf/navigation.jspf"%>

<center>
<h1><font color="red">Erreur...</font></h1>
<p>Votre demande n"a pu aboutir.</p>
<p>Merci de signaler les circonstances de cet incident au webmaster
<br>de ce site en lui transmettant le texte d"erreur qui suit :</p>
<p><b><%= exception %></b></p>
</center>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

C'est la page d'erreur du site. A ce titre, elle doit comporter une directive page avec l'attribut isErrorPage assignée à la valeur true. La page erreur.jsp doit être affichée chaque fois qu'une exception non interceptée se produira dans la page d'accueil bienvenue.jsp. Elle n'est pas conçue pour être accessible directement aux utilisateurs du site et se trouve donc dans la zone privée <WEB-INF>.

Les actions

Le dernier groupe d'éléments est celui des actions, également appelées actions standards. Les actions standards sont définies par la spécifiaction JSP. (C'est pour cette raison qu'elles sont appelées standards).

Dans la prochaine étude, nous verrons qu'il est possible de définir de nouvelles actions et les utiliser dans nos pages JSP.
.

La spécification JSP 2.0 définit les actions standards suivantes :

1. *<jsp:useBean>*
2. *<jsp:setProperty>*
3. *<jsp:getProperty>*
4. *<jsp:param>*
5. *<jsp:include>*
6. *<jsp:forward>*
7. *<jsp:plugin>, <jsp:params>, <jsp:fallback>*
8. *<jsp:attribute>*
9. *<jsp:body>*
10. *<jsp:invoke>*
11. *<jsp:dobody>*

Dans la suite de cette étude, nous verrons certaines de ces actions suivant le besoin et dans l'ordre qu'il conviendra. Nous commencerons d'ailleurs par les JavaBeans. D'autres actions seront étudiées dans la prochaine étude.

Les JavaBeans

Dans les pages JSP, il est toujours très difficile de lire ce mélange à la fois de code HTML et de code Java. Il serait plus judicieux, dans la mesure du possible, d'utiliser une écriture plus proche du HTML en utilisant la syntaxe du XML tout en faisant référence, malgré tout, à des classes Java. Le webmaster s'occuperait alors des balises à mettre en place sur ces différentes pages Web dynamiques, alors que le développeur s'occuperait essentiellement de mettre en place toutes les classes nécessaires à l'application Web. C'est d'ailleurs déjà une approche que nous avons eu. Toutefois, actuellement, dans la page Web bienvenue.jsp, nous retrouvons quand même un peu de code Java pour pouvoir utiliser ces classes. Les JavaBeans permettent de composer une structure particulière sur ces classes respectant un canevas standard afin qu'ils puissent être utilsés par le webmaster au moyen de balises spécifiques et donc sans code Java.

L'action <jsp:useBean>

Cet élément permet de rendre un JavaBean accessible dans la page. Un JavaBean (ce qui n'est pas la même chose qu'un Entreprise JavaBean) est simplement une classe Java respectant un certain nombre de conventions. Les deux plus impotantes sont :

1. *La classe d'un JavaBean doit posséder un constructeur sans arguments.*
2. *La classe d'un JavaBean doit posséder un ensemble de propriétés.*

Une propriété est composée de trois éléments : d'abord un attribut privé suivi de deux méthodes publiques associées. Chaque propriété doit donc être accessible au client par l'intermédiaire de deux méthodes spécialisées, appelées accesseurs : une méthode get pour lire la valeur de la propriété et une méthode set pour la modifier.

Le nom de chaque accesseur, appelés communément getter et setter est construit avec get ou set suivi du nom de la propriété (attribut) avec la première lettre transformée en majuscule. Dans le cas des propriétés booléennes, on utilise les forme isXxx( ) et getXxx( ).

Ainsi en prenant comme exemple la propriété nom :

```
private String nom ;
public String getNom( ) { return nom; }
public void setNom(String nom) { this.nom = nom; ...(reste du code)... }
```

D'une façon générale, nous avons :

```
private type unePropriété ;
public type getUnePropriété( ) { return unePropriété; }
public boolean isUnePropriété( ) { return unePropriétéBooléenne; }
public void setNom(type unePropriété) { this.unePropriété = unePropriété; ...(reste du code)... }
```

L'action <jsp:useBean> prend le paramètres suivants :

1. *id : Le nom utilisé pour accéder au bean dans le reste de la page. Il doit être unique. Il s'agit en fait du nom de l'objet référençant l'instance de la classe du bean donné par le paramètre class.*
2. *scope : La portée du bean. Les valeurs possibles sont page, request, session et application. La valeur par défaut est page.*
3. *class : Le nom de la classe bean.*
4. *beanName : Le nom du bean, tel qu'il est requis par la méthode instanciate() de la classe java.beans.Beans. Le plus souvent, vous utiliserez class plutôt que beanName.*
5. *type : Le type de la variable référençant le bean. Conformément aux règles de Java, il peut s'agir de la classe du bean, d'une classe parente, ou d'une interface implémentée par le bean ou une classe parente.*

Lorsqu'il rencontre l'action <jsp:useBean>, le conteneur de l'application Web recherche dans la portée indiquée s'il existe un objet avec l'id correspondante. S'il n'en trouve pas, et si une classe a été spécifiée, il tente de créer une instance (un objet). Il est possible d'utiliser les attributs class, beanName, et type dans les conbinaisons suivantes :

1. *class - Crée une instance de la classe qui sera référencée par la valeur de l'id.*
2. *class, type - Crée une instance de la classe qui sera référencée par la valeur de l'id, avec le type indiqué.*
3. *beanName, type - Crée une instance du bean indiqué. La référence aura le type indiqué.*
4. *type - Si un objet du type indiqué existe dans la session, il sera référencé par la valeur de l'id.*

Il est indispensable de créer une référence à un JavaBean à l'aide de <jsp:useBean> avant de pouvoir utiliser les actions <jsp:setProperty> et <jsp:getProperty>.

L'action <jsp:setProperty>

Cette action permet de modifier la valeur d'une propriété d'un JavaBean. Elle prend les attributs suivant :

1. *name* - *l'id du bean*.
2. *property* - *le nom de la propriété à modifier* - *Le valeur peut nommer explicitement une propriété du bean. Dans ce cas, la méthode setXxx( ) de cette propriété sera appelée. La valeur peut également être (*). Dans ce cas, le conteneur lit tous les paramètres de la requête envoyée par le client et modifie les valeurs des propriétés correspondantes.*
3. *value* - *contient la nouvelle valeur à affecter à la propriété*.
4. *param* - *le nom du paramètre de la requête contenant la valeur à affecter à la propriété*. *Cet attribut permet également de changer la valeur de la propriété comme l'attribut value. Toutefois, cet attribut param va plus loin puisqu'il demande au conteneur de JSP de chercher un paramètre dans la requête envoyée à la page JSP portant le nom mentionné puis d'écrire directement la valeur trouvée dans la propriété désignée.*

Les attributs name et property sont toujours requis. Les attributs param et value sont mutuellement exclusifs. Si aucun d'eux n'est présent, l'action <jsp:setProperty> tente d'utiliser le paramètre de la requête portant le même nom que la propriété.

Supposons que nous ayons un JavaBean contenant les référence d'une personne qui serviront ensuite à identifier un utilisateur de l'application Web :

```
public class Personne {
  private String nom;
  private String prénom;
  private String motDePasse;

  public Personne() { }

  public String getNom() { return this.nom; }
  public void setNom(String nom) { this.nom = nom; }

  public String getPrénom() { return this.prénom;}
  public void setPrénom(String prénom) { this.prénom = prénom; }

  public String getMotDePasse() { return this.motDePasse; }
  public void setMotDePasse(String motDePasse) { this.motDePasse = motDePasse;  }
}
```

Voici un exemple d'utilisation de <jsp:setProperty> avec une valeur littérale et une expression :

```
<jsp:useBean id = "utilisateur" class = "Personne" />
<jsp:setProperty name = "utilisateur" property = "nom" value = "REMY" />
<jsp:setProperty name = "utilisateur" property = "prénom" value = "<%= request.getParameter("prénom") %>" />
```

Après que la page JSP contenant ce code a été exécuté, la propriété nom du bean utilisateur a la valeur "REMY" et la propriété prénom a la valeur retournée par la requête du client. Lors de l'exécution, les actions sont transformées en code Java qui crée une instance de Personne et appelle ses méthodes setNom() et setPrénom().

Pour la dernière ligne de code, nous venons de le voir, il s'agit de changer la propriété prénom du bean Personne. Toutefois, la nouvelle valeur de cette propriété est délivrée par la requête envoyée à la page JSP. Du coup, il est souvent préférable d'utiliser l'attribut param en lieu et place de value.

```
<jsp:useBean id = "utilisateur" class = "Personne" />
...
<jsp:setProperty name = "utilisateur" property = "prénom" param = "prénom" />
```

Le paramètre de la requête peut avoir un nom différent de celui de la propriété, puisque à l'aide de l'attribut param, nous pouvons spécifier la connexion désirée. Ici, vu que le nom du paramètre de la requête porte effectivement le même nom que la propriété, nous pouvons nous passer de l'attribut param :

```
<jsp:useBean id = "utilisateur" class = "Personne" />
...
<jsp:setProperty name = "utilisateur" property = "prénom" />
```

L'action <jsp:getProperty>

Cette action permet de lire la valeur d'une propriété d'un JavaBean. Elle possède les attributs suivant :

1. *name* - *l'id du bean.*
2. *property* - *le nom de la propriété à lire.*

Les attributs name et property sont toujours requis. Lorsque cette action est présente dans une JSP, la valeur de la propriété est incluse dans la réponse à la requête et donc, au cas où, la valeur retournée est transformée en chaîne de caractères même si le type de la propriété n'est pas de type String. Dans l'exemple précédent, nous pourrions utiliser cette action de la manière suivante.

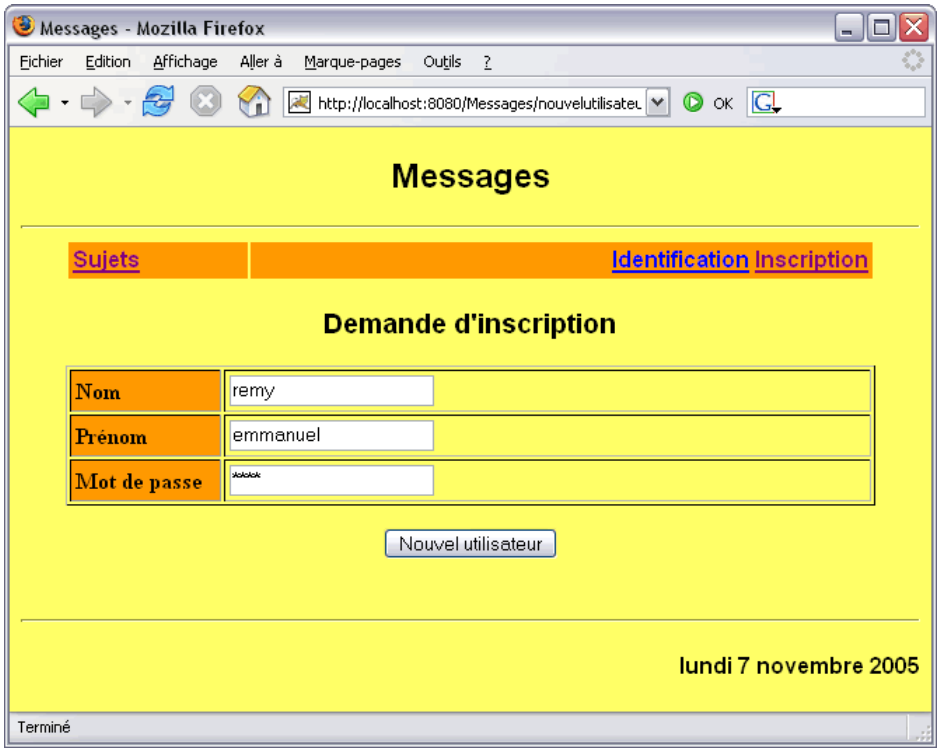
```
L'utilisateur a pour nom
<jsp:getProperty name = "utilisateur" property = "nom" />
et pour prénom
<jsp:getProperty name= "utilisateur" property = "prénom" />
```

Lorsque la page JSP est traduite en code Java, cette action est remplacée par un appel aux méthodes getNom() et getPrénom(). Les valeurs retournées sont placées dans le texte de la réponse qui est renvoyée au client sous la forme :

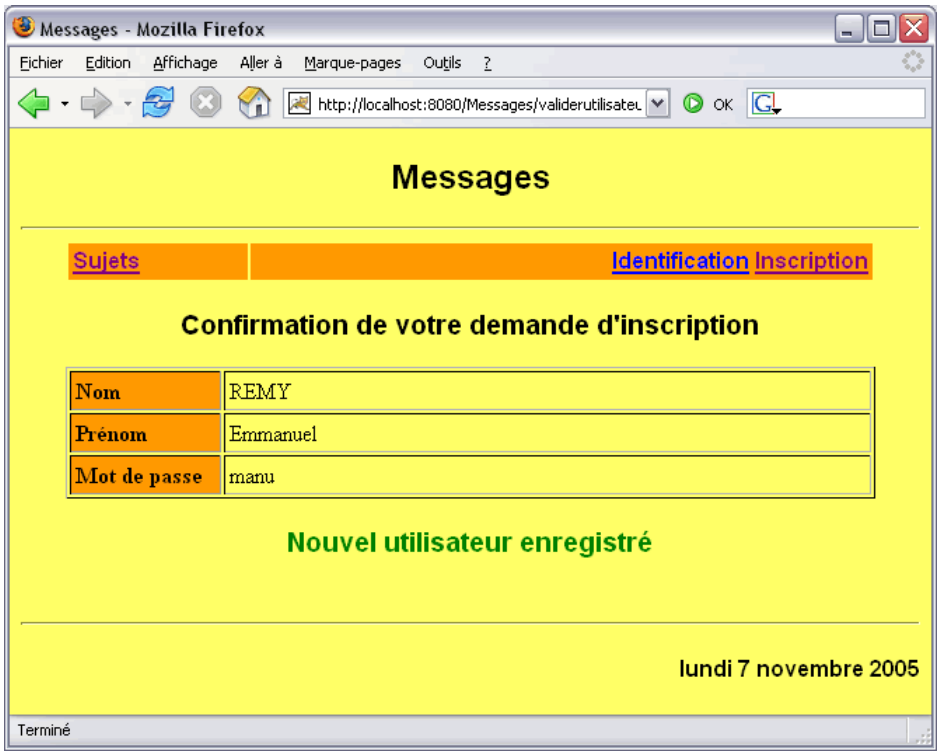
```
L'utilisateur a pour nom REMY et pour prénom Emmanuel.
```

Exemple de JavaBean sur le site de gestion des messages

Nous allons reprendre le site de la messagerie auquel nous rajouterons l'inscription d'un nouvel utilisateur. Voici d'ailleurs ci-dessous la page d'inscription <nouvelutilisateur.jsp> :



L'opérateur devra confirmer sa saisie par l'appui sur le bouton "Nouvel utilisateur". Si effectivement, c'est la première fois que cet opérateur s'enregistre, il devrait alors voir la page suivante <validerutilisateur.jsp> qui sert de confirmation de l'enregistrement réel dans la base de données :



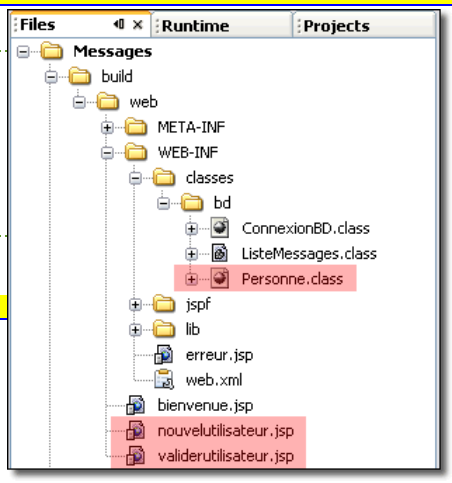
Dans le cas contraire, les données ne seront effectivement pas enregistrées dans la base de données, et la même pages jSP <validerutilisateur.jsp> devrait plutôt produire la page HTML suivante :

Remarquez au passage que le nom de l'utilisateur a automatiquement été mis en majuscule. Pour le prénom, seule la première lettre est en majuscule, par contre, les autres lettres du mot, même si cela n'apparaît pas ici, doivent être mises en minuscule.

Nouvelle architecture de l'application Web

Pour réaliser l'ensemble de l'inscription, nous avons besoin de développer trois nouveaux composants Web :

1. *Le javaBean Personne qui récupère les informations saisies par l'utilisateur afin de les enregistrer dans la base de données avec une mise en majuscule adaptée.*
2. *La page JSP <nouvelutilisateur.jsp> qui s'occupe du formulaire de saisie.*
3. *La page JSP <validerutilisateur.jsp> qui récupère les informations issues du formulaire, se met ensuite en relation avec le JavaBean Personne et affiche le résultat suivant le comportement du JavaBean, c'est-à-dire, suivant si la personne est déjà enregistrée ou pas.*



JavaBean Personne

Nous allons continuer le codage du JavaBean Personne déjà vu précédemment dans la partie cours. Nous allons faire en sorte qu'il soit en connexion avec la base de données, notamment avec la table personne. Du coup, vu que nous sommes avec la même base de données, il est plus judicieux, comme pour la classe ListeMessages, que ce JavaBean hérite de la classe de base ConnexionBD. Ainsi, toute la problématique de la connexion avec la base de données est déjà résolue, il suffit juste de mettre en oeuvre la requête adaptée à l'insertion. Nous prendrons donc la méthode miseAJour() de la classe ConnexionBD. Nous allons profiter de la construction de ce JavaBean Personne pour traiter les majuscules.

Nous voyons ici, l'intérêt de scinder notre travail en plusieurs classes spécialisées. Chaque classe s'occupe d'un problème particulier.
.

Personne.java

```
package bd;

public class Personne extends ConnexionBD {
   private String nom;
   private String prénom;
   private String motDePasse;

   public Personne() { }

   public String getNom() {
      return this.nom;
   }
   public void setNom(String nom) {
      this.nom = nom.toUpperCase();
   }

   public String getPrénom() {
      return this.prénom;
   }
   public void setPrénom(String prénom) {
      this.prénom  = prénom.substring(0, 1).toUpperCase() + prénom.substring(1, prénom.length()).toLowerCase();
   }

   public String getMotDePasse() {
      return this.motDePasse;
   }
   public void setMotDePasse(String motDePasse) {
      this.motDePasse = motDePasse;
   }
```

```
   public boolean enregistrer() {
      if (existeDéjà())
         return false;
      else {
         miseAJour("INSERT INTO personne (nom, prénom, motDePasse) VALUES (\""+nom+"\",\""+prénom+"\",\""+motDePasse+"\")");
         return true;
      }
   }
   private boolean existeDéjà() {
      lire("SELECT * FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\"");
      return suivant();
   }
}
```

La première fois que nous avons mis en oeuvre ce JavaBean, les méthodes dites accesseurs ne faisaient que du copier-coller entre le paramètre de la méthode et l'attribut de la classe. Nous pouvions nous demander a quoi cela pouvait servir. Pourquoi ne pas manipuler directement les attributs. Dans la philosophie Objet, ce que nous appelons encapsulation nous impose effectivement d'avoir des attributs privés. C'est normal et logique, par exemple, lorsque nous disposons d'une voiture bleu, elle ne peut pas devenir rouge comme cela, il faut bien passer par une certaine procédure ; décaper, poncer, peindre, etc. Ainsi, pour accéder aux attributs, il faut passer par deux méthodes. L'une est prévue pour retourner la valeur de l'attribut correspondant, sans modification de l'attribut lui-même. La deuxième est prévue pour faire évoluer l'état de l'objet, c'est-à-dire, de changer la valeur de l'attribut correspondant. Généralement, avant que l'attribut soit changé, il est souvent nécessaire d'avoir une mis en forme adaptée comme c'est le cas ici avec la gestion des majuscules, notamment pour les méthodes setNom() et setPrénom(). Nous voyons bien ici, l'intérêt de passer systématiquement par une méthode. Si nous accedions directement avec l'attribut nom, ce n'est pas sûr que nous aurions penser à le mettre en majuscule, alors que là, c'est déjà traité par la méthode, et de plus nous n'avons pas en nous préoccuper. Surtout, grâce à cette démarche, nous sommes sûr que l'attribut est parfaitement conditionné comme le développeur l'a prévu. Ainsi, l'encapsulation permet de se protéger contre toute mauvaise utilisation.

Vous avez remarquez que nous avons rajouté deux autres méthodes qui vont particulièrement utiliser les compétences de la classe de base et vont donc servir à décrire les requêtes adaptées :

1. ***enregistrer()*** *: Cette méthode permet d'enregistrer dans la base de données l'ensemble des attributs déjà présents dans l'objet relatif à la classe* ***Personne****. Toutefois, cet enregistrement ne se réalise que si la personne n'a jamais été enregistrée. Si l'enregistrement c'est réalisé correctement, la méthode renvoie la valeur* ***true****, ce qui signifie que c'est la première fois que cette personne s'enregistre. La valeur* ***false*** *est renvoyée dans le cas contraire.*
2. ***existeDéja()*** *: Cette méthode est privée, donc inaccessible de l'extérieur. Elle est utilisée par la méthode* ***enregistrer()*** *afin de tester si la personne actuelle représentée par l'ensemble des attributs de la classe est déja présente dans la base de données.*

La page JSP <nouvelutilisateur.jsp>

Cette page permet à l'opérateur, à l'aide d'un formulaire, de réaliser la saisie des références d'un nouvel utilisateur de la messagerie.

nouvelutilisateur.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp"%>
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<h3 align="center">Demande d"inscription</h3>

<form action="validerutilisateur.jsp" method="post">
    <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
            <td><input type="text" name="nom"></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
            <td><input type="text" name="prénom"></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
            <td><input type="password" name="motDePasse"></td>
        </tr>
    </table></p>
    <p align="center"><input type="submit" value="Nouvel utilisateur"></p>
</form>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Cette page, mise à part les directives, est essentiellement composée de code HTML. C'est elle qui propose le formulaire. Nous aurions pu presque utiliser directement une page HTML plutôt qu'une page JSP. L'intérêt ici de prendre malgré tout une page JSP, c'est de pouvoir faire appel à l'en-tête et au pied de page déjà créées et aussi de spécifier la page d'erreur à solliciter en cas de problème.

Attention, les noms donnés aux balises <input> est très important. Ces noms doivent correspondre parfaitement aux noms des attributs du JavaBean Personne, donc respectivement : nom, prénom, motDePasse.

La page JSP <validerutilisateur.jsp>

Cette page récupère les informations issues du formulaire, se met ensuite en relation avec le JavaBean Personne et affiche le résultat suivant le comportement du JavaBean, c'est-à-dire, suivant si la personne est déjà enregistrée ou pas.

validerutilisateur.jsp

```
 1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
 2 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 3
 4 <h3 align="center">Confirmation de votre demande d"inscription</h3>
 5
 6 <jsp:useBean id="utilisateur" class="bd.Personne">
 7     <jsp:setProperty name="utilisateur" property="*" />
 8
 9     <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
10         <tr>
11             <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
12             <td><jsp:getProperty name="utilisateur" property="nom" /></td>
```

```
13          </tr>
14          <tr>
15              <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
16              <td><jsp:getProperty name="utilisateur" property="prénom" /></td>
17          </tr>
18          <tr>
19              <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
20              <td><jsp:getProperty name="utilisateur" property="motDePasse" /></td>
21          </tr>
22      </table></p>
23      <h3 align="center">
24      <% if (!utilisateur.enregistrer()) { %>
25          <font color="red">ATTENTION : Utilisateur déja enregistré</font>
26      <%
27         }
28         else {
29      %>
30          <font color="green">Nouvel utilisateur enregistré</font>
31      <%
32         }
33         utilisateur.arrêt();
34      %>
35      </h3>
36  </jsp:useBean>
37
38  <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Comme beaucoup de pages JSP, cette page est composée de balisage HTML avec également un mélange de codage Java. Ici, toutefois, grâce à l'écriture spécifique des JavaBeans, nous n'avons pas d'écriture dans tous les sens avec un code fouillu. Elle est au contraire très concise et claire. Nous avons presque essentiellement du balisage, ce qui est intéressant pour le webmaster. En effet, les seules scriptlets que nous avons, concernent l'enregistrement éventuel dans la base de données.

Ligne 6 :

Création de l'objet **utilisateur** correspondant au JavaBean **Personne**.

Ligne 7 :

Cette balise **<jsp:setProperty>** permet de récupérer toutes les valeurs des paramètres délivrées par la requête issue de la page **<nouvelutilisateur.jsp>** et de positionner, en une seule fois, tous les attributs de l'objet **utilisateur**. Nous voyons là, particulièrement, l'intérêt de travailler avec les JavaBeans.

Lignes 9 à 22 :

Présentation d'un tableau récapitulant la saisie faite par l'opérateur. Chaque information stockée actuellement dans l'objet **utilisateur** (grâce à la balise **<jsp:setProperty>** de tout à l'heure) est délivrée au moyen de la balise **<jsp:getProperty>**.

Lignes 24 à 34 :

Appel à la méthode **enregistrer()** de l'objet **utilisateur** afin que ses attributs actuels soient placés définitivement dans la base de données. Cette méthode retourne **true** si effectivement les attributs ont été sauvegardés dans la base de données. Dans ce cas là, un affichage en couleur verte est proposé pour indiquer que tout s'est bien passé. Dans le cas contraire, la méthode **enregistrer()** renvoie **false** et un message d'avertissement en rouge est alors proposé. Après avoir fait toutes ces opérations, nous faisons appel à la méthode **arrêt()** afin que la connexion à la base de données soit interrompue.

Au travers de cet exemple, et grâce à la technique des JavaBean, nous séparons bien le rôle du développeur de celui du webmaster. Tous les éléments sont concis est clair. Globalement, pour chaque composant, nous avons très peu de lignes de code. N'hésitez pas à utiliser la philosophie objet en mettant en oeuvre notamment l'héritage. Cela simplifie le codage et favorise le travail en équipe.

Les objets implicites

Avant de continuer sur la suite des actions, nous allons passer par la connaissance des objets implicites. Nous venons de voir que les propriétés d'un JavaBean pouvaient être modifiées à l'aide des valeurs des paramètres de la requête envoyée par le client. Une page JSP peut accéder directement à la requête, par l'intermédiare d'un objet implicite nommé request. Ce nom nous est familier, nous l'avons déjà rencontré dans les servlets. Comme les pages JSP sont finalement des servlets, il est normal de retrouver les mêmes objets. La particularité, c'est qu'ils existent implicitement.

Le modèle JSP définit un certain nombre d'objets implicites. Ces objets sont appelés ainsi car nous pouvons y accéder sans jamais avoir à les déclarer ou à les initialiser. Les objets implicites sont accessibles dans les scriptlets et dans les expressions. Voici la liste des objets implicites :

1. *request*
2. *response*
3. *out*
4. *session*
5. *config*
6. *exception*
7. *application*

L'objet request

Les pages JSP sont des composants Web dont le rôle est de répondre à des requêtes HTTP. L'objet implicite request représente la requête que doit traiter la page. Grâce à cet objet, il est possible de lire les en-têtes de la requête, ses paramètres, ainsi que de nombreuses autres informations. Le plus souvent toutefois, l'objet request est utilisé pour connaître les paramètres de la requête.

```
String request.getParameter(String nom) ;
```

La méthode getParameter(String) retourne la valeur du paramètre dont le nom correspond à la chaîne utilisée comme argument. Le nom utilisé comme argument de la méthode getParameter(String) doit être le même que celui employé dans le formulaire. Ainsi, en se servant de l'exemple du JavaBean de la section précédente, et si nous désirons récupérer la valeur du prénom saisie par l'opérateur et l'afficher en grand :

```
<h3><%= request.getParameter("prénom") %></h3>
```

L'objet request dispose de beaucoup d'autres méthodes que nous avons déjà découvert. *Revoyer ces méthodes dans l'étude sur les servlets*

L'objet out

L'objet implicite out est une référence au stream de sortie utilisée par la réponse. Il peut être employé dans une scriptlet pour écrire des données dans la réponse envoyée au client.

Ainsi, le code suivant :

```
<h3><%= request.getParameter("prénom") %></h3>
```

Peut être remplacé par :

```
<% out.println("<h3>"+request.getParameter("prénom")+"</h3>") ; %>
```

Dans cet exemple, l'utilisation de l'objet implicite out ne procure pas un grand intérêt. Là où il peut être utile, c'est lorsque nous sommes dans une grande scriptlet, et que nous ne désirons pas placer du code HTML au milieu de cet scriptlet. Personnellement, je l'utilise assez rarement.

L'objet session

Rappelons que HTTP est un protocole sans état. Du point de vue d'un serveur Web, chaque requête envoyée par un client est une nouvelle requête, sans aucune relation avec les précédentes. Toutefois, dans le cas des applications Web, l'interaction d'un client avec l'application s'étend souvent sur plusieurs requêtes et réponses. Pour rassembler ses différentes interactions en une conversation cohérente entre le client et l'application, il est nécessaire de faire appel au concept de session. Une session est l'ensemble d'une conversation entre un client et le serveur.

Les composants JSP d'une application Web participent automatiquement à une session, sans nécessité aucune intervention. En revanche, si une page JSP utilise la directive page pour donner à l'attribut session la valeur false, cette page n'aura plus accès à l'objet session et ne pourra donc pas participer à la session.

Grâce à l'objet session, la page peut stocker des informations à propos du client ou de l'état de la conversation avec celui-ci. Par contre, nous ne pouvons placer dans une session que des objets, et non des primitives Java. Pour conserver des primitives, il faut les envelopper dans une classe prévue à cet effet, comme Integer, Double ou Boolean. Les méthodes permettant de placer des objets dans la session et de les y retrouver sont les suivantes :

```
Object setAttribute(String nom, Object valeur) ;
Object getAttribute(String nom) ;
Enumeration getAttributeNames ( ) ;
void removeAttribute(String nom) ;
```

Lorsque d'autres composants de l'application Web reçoivent une requête, ils peuvent accéder aux données qui ont été placées dans la session par les premiers composants. Ils peuvent modifiers ces données ou en ajouter.

Vous pouvez placer das JavaBean dans la session. Par contre là, il est nécessaire de le préciser au moyen de l'attribut scope, puisque par défaut, la portée d'un JavaBean est la page JSP en cours. Ainsi, vous pouvez, par exemple, conserver le nom de l'utilisateur durant toute la session de l'application Web messagerie. Du coup, dans la page d'accueil du site, nous pouvons faire apparaître son identité et donner la liste des messages le concernant.

Si nous désirons que l'objet utilisateur du JavaBean Personne soit stocké dans la session, nous devons apporter la modification suivante :

```
<jsp:useBean id = "utilisateur" class = "bd.Personne" scope = "session" />
```

Lorsqu'une autre page désire retrouver cet objet stockée dans la session, nous devons écrire :

```
Personne opérateur = (Personne) session.getAttribute("utilisateur") ;
```

L'objet exception

Cet objet implicite est accessible dans les pages d'erreur. Nous l'avons d'ailleurs déjà utilisé dans notre application Web messagerie. Il s'agit d'une référence à l'objet java.lang.Throwable qui a causé l'utilisation de la page d'erreur.

L'objet application

Cet objet représente l'environnement de l'application Web. Il peut être utilisé pour lire les paramètres de configuration de l'application. Ces paramètres sont définis dans le descripteur de déploiement, dans l'élément <webapp> :

```
<webapp>
        <context-param>
                <param-name>nom</param-name>
                <param-value>valeur</param-value>
        </context-param>
</webapp>
```

Dans la page JSP, le paramètre de configuration peut être récupéré au moyen de la méthode getInitParameter(String) de l'objet implicite application :

```
application.getInitParameter(String nom) ;
```

L'objet config

Cet objet est utilisé pour lire les paramètres d'initialisation spécifiques aux pages JSP. Ces paramètres sont définis dans le descripteur de déploiement, mais concernant une page particulière et non plus toute l'application Web comme pour l'objet application. Ces paramètres figurent dans l'élément <servlet> car la page une fois compilée est en fait une servlet. L'élément <servlet> peut contenir un ou plusieurs éléments <init-param>, comme dans l'exemple suivant :

```
<servlet>
        <servlet-name>NouveauMessage</servlet-name>
        <servlet-class>NouveauMessage.class</servlet-class>
        <init-param>
                <param-name>nom</param-name>
                <param-value>valeur</param-value>
        </init-param>
</servlet>
```

Les paramètres définis dans le descripteur de déploiement sont accessibles grâce à la méthode getInitParameter(String) de l'objet implicite config :

```
config.getInitParameter(String nom) ;
```

Les différentes portées

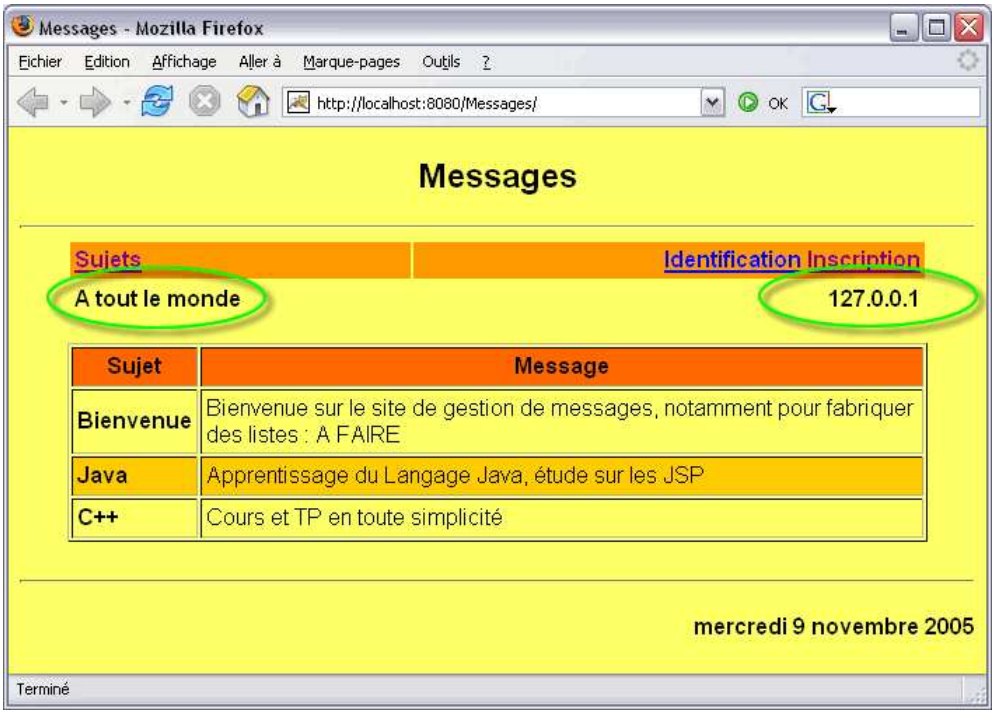
Les objets créés dans les pages JSP ont une certaine portée, qui correspond en quelque sorte à leur durée de vie. Dans certain cas, cette portée est déterminée et ne peut être modifiée. Il en ainsi des objets implicites. Pour d'autre objets (par exemple les JavaBeans), le développeur peut choisir la portée. Les portées valides sont page, request, session, application.

1. *page - C'est la portée la plus réduite. Les objets ayant cette portée ne sont accessibles que dans la page qui les définit. Les JavaBeans créés avec la portée page, ainsi que les objets créés dans les scriptlets, sont protégés contre les accès concurrents.*

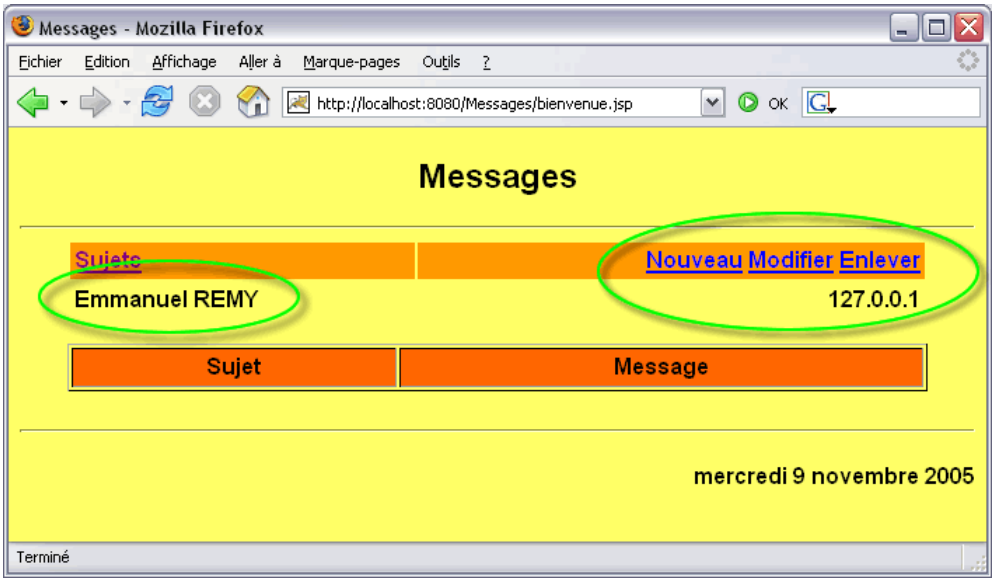
2. *request* - *Avec cette portée, les objets sont accessibles pendant toute la durée de la requête. Cela signifie qu'un objet est disponible dans la page qui l'a créé, ainsi que dans toutes les pages auxquelles la requête est transmise et dans celles qui sont incluses. Ces objets sont protégés contre les accès concurrents. Seul les thread exécutant la requête peut y accéder.*
3. *session* - *Les objets ayant cette portée sont accessibles pour tous les composants participant à la session. Ils ne sont pas protégés contre les accès concurrents.*
4. *application* - *Il s'agit de la portée la moins restrictive. Les objets concernés sont accessibles à toute l'application Web, pendant toute sa durée de vie. Ils ne sont pas protégés contre les accès concurrents.*

Exemple sur les objets implicites

Nous allons reprendre l'application Web précédente sur laquelle nous allons mettre en oeuvre un certain nombre d'objets implicites. Ce qui serait intéressant, c'est de connaitre durant toute la durée de la session, quel est l'utilisateur inscrit et éventuellement sur quel poste du réseau il travaille. Sinon, tant qu'il n'y a pas eu d'inscription, l'application Web doit s'adresser à tout le monde.



Par contre, une fois que l'inscription a eu lieu, le nom de l'opérateur doit apparaître. Le menu doit également changer afin de permettre la gestion des messages, comme par exemple, donner la possibilité de composer un nouveau message. Pour terminer, la liste des messages doit correspondre à l'utilisateur inscrit. D'ailleurs, la première fois, la liste des messages est vide.



Par ailleurs, la couleur de fond de toutes les pages du site doivent toujours être identique et configurable sans avoir à recompiler l'application.

Descripteur de déploiement <web.xml>

La seule possibilité de respecter le dernier critère est d'utiliser le descripteur de déploiement. Il suffit pour cela de créer un paramètre valide pour toute l'application Web que nous appelerons couleurFond.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <display-name>Liste des messages personnels</display-name>
  <welcome-file-list>
    <welcome-file>bienvenue.jsp</welcome-file>
  </welcome-file-list>
```

```
  <context-param>
      <param-name>couleurFond</param-name>
      <param-value>FFFF66</param-value>
  </context-param>
</web-app>
```

Pour cela, nous devons utiliser la balise <context-param>. Ici, la valeur du paramètre couleurFond est égale à FFFF66.
.

Modification de la page JSP : validerutilisateur.jsp

Il faut que notre application Web conserve l'utilisateur actuellement en connexion, quelque soit sa navigation. Il faut donc prendre en compte la gestion de session. Quand, l'opérateur s'inscrit, il est alors nécessaire qu'au moment de la création du JavaBean l'objet utilisateur soit placer dans la session, ce que ne fait pas par défaut un JavaBean.

validerutilisateur.jsp

```
 1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
 2 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 3
 4 <h3 align="center">Confirmation de votre demande d"inscription</h3>
 5
 6 <jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
 7     <jsp:setProperty name="utilisateur" property="*" />
 8
 9     <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
10         <tr>
11             <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
12             <td><jsp:getProperty name="utilisateur" property="nom" /></td>
13         </tr>
14         <tr>
15             <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
16             <td><jsp:getProperty name="utilisateur" property="prénom" /></td>
17         </tr>
18         <tr>
19             <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
20             <td><jsp:getProperty name="utilisateur" property="motDePasse" /></td>
21         </tr>
22     </table></p>
23     <h3 align="center">
24     <% if (!utilisateur.enregistrer()) { %>
25         <font color="red">ATTENTION : Utilisateur déja enregistré</font>
26     <%
27        }
28        else {
29     %>
30         <font color="green">Nouvel utilisateur enregistré</font>
31     <%
32        }
33 //       utilisateur.arrêt();
34     %>
35     </h3>
36 </jsp:useBean>
37
38 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Ligne 6 :

Il faut donc rajouter l'attribut **scope** avec la valeur **"session"** dans la balise **<jsp:javaBean>**.

Ligne 33 :

Attention, vu que l'utilisateur doit être conservé durant toute la session, il ne faut plus interrompre la connexion avec la base de données.

JavaBean Personne

Il faut également rajouter une méthode dans le JavaBean Personne pour qui nous procurera l'identificateur de la personne recherchée.

Personne.java

```
 1 package bd;
 2
 3 import java.sql.SQLException;
 4
 5 public class Personne extends ConnexionBD {
 6    private String nom;
 7    private String prénom;
 8    private String motDePasse;
 9
10    public Personne() { }
11
12    public String getNom() {
13       return this.nom;
14    }
15    public void setNom(String nom) {
16       this.nom = nom.toUpperCase();
17    }
18
19    public String getPrénom() {
20       return this.prénom;
21    }
22    public void setPrénom(String prénom) {
23       this.prénom  = prénom.substring(0, 1).toUpperCase() + prénom.substring(1, prénom.length()).toLowerCase();
24    }
```

```
25
26    public String getMotDePasse() {
27       return this.motDePasse;
28    }
29    public void setMotDePasse(String motDePasse) {
30       this.motDePasse = motDePasse;
31    }
32
33    public boolean enregistrer() {
34       if (existeDéjà())
35          return false;
36       else {
37          miseAJour("INSERT INTO personne (nom, prénom, motDePasse) VALUES
(\""+nom+"\",\""+prénom+"\",\""+motDePasse+"\")");
38          return true;
39       }
40    }
41    private boolean existeDéjà() {
42       lire("SELECT * FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\"");
43       return suivant();
44    }
45
46    public int identificateur(){
47       lire("SELECT idPersonne FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\"");
48       suivant();
49        try {
50          return résultat.getInt("idPersonne");
51        }
52        catch (SQLException ex) {
53          return 1;
54        }
55    }
56 }
57
```

Lignes 46 à 54:

Pour cela, nous créons donc une méthode qui s'appelle **identificateur()** qui renvoie la valeur entière représentant le numéro de la personne donné par la clé primaire de la table **personne** correspondant au nom et au prénom de l'utilisateur.

Le fragment de page : navigation.jspf

Le plus gros changement se situe sur le fragment de page navigation.jspf. C'est en effet sur ce fragment de page que nous devons identifier l'utilisateur et prévoir le menu en conséquence. C'est également à ce niveau que nous devons régler la couleur de fond. C'est enfin toujours sur ce fragment que nous allons identifier l'ordinateur hôte du client.

navigation.jspf

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
2     "http://www.w3.org/TR/html4/loose.dtd">
3
4  <%! int idPersonne = 1; %>
5  <%! String identité = "A tout le monde"; %>
6  <%
7      Personne opérateur = (Personne) session.getAttribute("utilisateur");
8      if (opérateur!=null) {
9          idPersonne = opérateur.identificateur();
10         identité = opérateur.getPrénom()+" "+opérateur.getNom();
11     }
12 %>
13
14 <html>
15   <head><title>Messages</title></head>
16   <body bgcolor="#<%= application.getInitParameter("couleurFond") %>">
17     <font face="Arial">
18         <h2 align="center">Messages</h2>
19         <hr>
20         <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
21             <tr bgcolor="#FF9900">
22                 <th align="left"><a href="bienvenue.jsp">Sujets</th>
23                 <th align="right">
24                     <% if (idPersonne == 1) { %>
25                         <a href="#">Identification</a>
26                         <a href="nouvelutilisateur.jsp">Inscription</a>
27                     <% }
28                        else { %>
29                          <a href="#">Nouveau</a>
30                          <a href="#">Modifier</a>
31                          <a href="#">Enlever</a>
32                     <% } %>
33                 </th>
34             </tr>
35             <tr>
36                 <th align="left"><%= identité %></th>
37                 <th align="right"><%= request.getRemoteHost() %></th>
38             </tr>
39         </table>
40
```

Lignes 4 et 5:

Déclaration de deux variables **idPersonne** et **identité** qui sont initialisée et qui seront utiles par la suite.

Lignes 6 à 12:

Sciptlet qui permet de récupérer l'objet **utilisateur** si il a été placé dans la **session**. Si c'est le cas, il faut modifier les deux variables précédemment créées afin de tenir compte des références de cet utilisateur, c'est-à-dire, à la fois son identificateur ainsi que son identité.

Ligne 16:

Récupération de la valeur du paramètre **couleurFond** placé dans le descripteur de déploiement afin de préciser la couleur de fond de la page. **(Vu qu'il s'agit d'un fragment de page, cela concerne en fait toutes les pages du site)**.

Lignes 24 à 32:

Fabrication du menu en prenant en compte si l'utilisateur est connu ou pas.

Ligne 36:

Affichage du prénom et du nom de l'utilisateur s'il est connu, sinon affichage du texte : **"A tout le monde"**.

Ligne 37:

Récupération au moyen de la requête du nom de l'ordinateur hôte du client (ou son adresse IP).

La page bienvenue.jsp

Ici, très peu de modification. Il faut juste afficher la liste des messages correspondant à la personne inscrite.

bienvenue.jsp

```
1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
2 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
3
4 <font face="Arial">
5
6 <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
7     <tr bgcolor="#FF6600">
8         <th>Sujet</th>
9         <th>Message</th>
10    </tr>
11    <%
12        ListeMessages listeMessages = new ListeMessages(idPersonne);
13        int ligne = 0;
14        while (listeMessages.suivant()) {
15    %>
16    <tr bgcolor="<%= ligne++ % 2 == 0 ? "#FFFF66" : "#FFCC00" %>">
17        <td><b><%= listeMessages.sujet() %></b></td>
18        <td><%= listeMessages.texte() %></td>
19    </tr>
20    <%
21        }
22        listeMessages.arrêt();
23    %>
24 </table></p>
25 </font>
26
27 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Ligne 12:

Seule cette ligne subit une modification. Cette fois-ci, nous tenons compte de l'identificateur déterminé dans le fragment **navigation.jspf** pour proposer la liste des messages. **(Nous voyons ici l'intérêt des déclarations de variables au sein de la page entière)**.

Inclusion de pages et transmission de requêtes

Les pages JSP offrent la possibilité d'inclure d'autres pages ou servlets dans la sortie renvoyée au client, ou de transmettre la requête à une autre page ou à une servlet, grâce aux actions standards <jsp:include> et <jsp:forward>.

L'action include

Comme nous l'avons déjà découvert, il peut arriver que de nombreuses pages JSP contiennent des fragments semblables, voir identiques, comme par exemple, le haut d'une page Web ou encore un des éléments de la charte graphique. Il est alors pratique d'isoler ces frament dans des fichiers séparés et de les inclures dans les différentes pages qui en ont besoin. Cette approche facilite la maintenance du site car il n'ya plus qu'un fichier à modifier. Il existe deux approches :

1. *La directive include,*
2. *L'action include.*

L'inclusion d'une page à l'aide d'une action standard diffère de celle réalisée grâce à la directive include par le moment où elle a lieu et la façon dont la ressource désignée est incluse :

1. ***Directive** - <% include file ="..." %> - La ressource est incluse au moment de la traduction de la page en code Java. Il peut s'agir d'un fragment de page. Cette inclusion a pour effet de copier le fichier dans la page au moment de sa construction. C'est un moyen partique de mettre en facteur commun des contenus statiques. En effet, il s'agit ici d'une inclusion statique, c'est-à-dire que si la ressource incluse est modifiée après l'inclusion, cette modification ne sera pas visible dans la page contenant la directive, jusqu'à ce que celle-ci soit modifiée à son tour et de nouveau traduite en code Java. Il faut vraiment considérer ce type d'inclusion comme une partie de page (fragment). C'est comme si nous avions qu'une seule page découpée en plusieurs morceaux. Ainsi, si vous déclarez des variables dans le fragment de page, ces variables seront bien entendues accessibles sur la page qui propose l'inclusion, puisque finalement, il s'agit de la même page.*
2. ***Action** - <jsp:include page = "..." /> - Contrairement aux directives qui ne servent qu'au moment de la traduction/compilation de page, les actions sont exécutées lors du traitement d'une requête. Avec l'action standard **include**, la page stoppe le traitement de la requête et la transmet à la ressource incluse. Celle-ci renvoie sa réponse, qui est incluse dans la page appelante. Cette page reprend alors le contrôle de la requête. Il doit s'agir d'une page valide ou d'une servlet. Nous avons donc ici un comportement dynamique. La page incluse est considérée comme une page en elle-même. Si vous déclarez des variables à l'intérieur de cette dernière, elles ne seront pas accessibles dans la page qui propose l'inclusion. Du coup, nous ne pouvons considérer la page incluse tout à fait comme un fragment de page, même si elle s'occupe de la gestion d'une partie de page. D'ailleurs, son extension est comme une page classique **<*.jsp>** et non plus comme un fragment de page **<*.jspf>**.*

L'action forward

Il arrive qu'une page JSP souhaite rediriger la requête vers une autre page. Avec l'action standard forward, la page arrête le traitement de la requête et transmet celle-ci à la ressource désignée. La page appelante ne reprend pas le contrôle. A la différence de l'action include, qui peut être employée n'importe où dans une page, l'action forward doit être placée avant tout envoi de données dans l'objet OutputStream de la réponse, c'est-à-dire avant tout code HTML dans la page, et avant toute scriptlet ou expression écrivant l'OutputStream. Si des données ont déja été écrites, l'action forward provoque une exception.

L'action forward est très utile pour contrôler les requêtes qui sont envoyées par le client afin de permettre des redirections et ainsi afficher la bonne page qui correspond à la situation actuelle du client.

Utilisation de include et forward

La syntaxe de l'action include est la suivante :

```
<jsp:include page = "URL" flush = "true|false">
        <jsp:param name = "nom" value = "valeur" />
</jsp:include>
```

L'attribut page est requis et sa valeur doit être l'URL de la page à inclure. L'attribut flush est optionnel. Il indique si le tampon de sortie doit être vidé avant l'inclusion. La valeur par défaut est false.

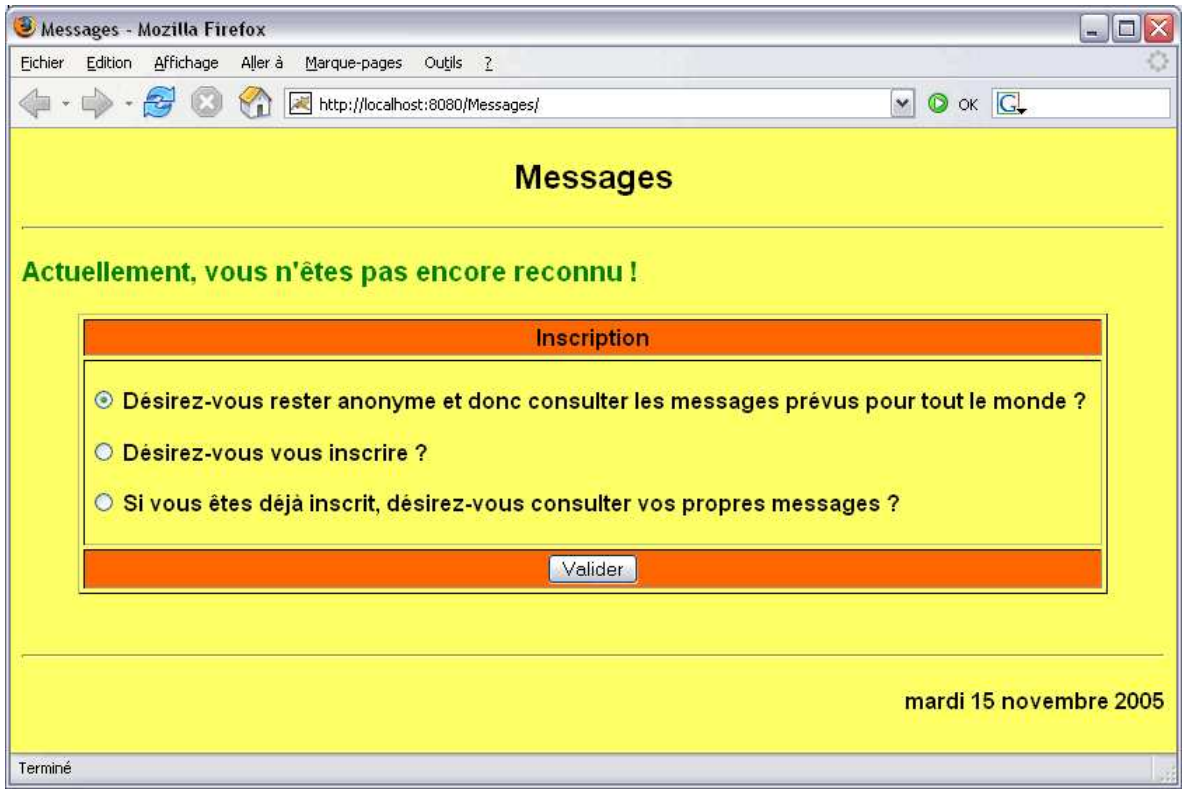
S'il est nécessaire de passer des paramètres à la ressource incluse, cela peut-être fait au moyen de l'élément <jsp:param>. Un élément doit être présent pour chaque paramètre. Cet élement est optionnel mais, s'il est présent, les deux attributs name et value sont obligatoires. Les paramètres ainsi définis sont ajoutés à la requête et sont donc accessibles grâce aux méthodes getParameter() et getParameterValues() de l'objet request.

La syntaxe de l'action forward et la signification de l'élément <jsp:param> est similaire :

```
<jsp:forward page = "URL" flush = "true|false">
        <jsp:param name = "nom" value = "valeur" />
</jsp:forward>
```

Exemple sur les transmissions de requêtes

Nous allons reprendre l'application Web précédente. La page d'accueil <bienvenue.jsp> proposera des redirections éventuelles suivant si le client se connecte pour la première fois, où au contraire, si il s'est déjà identifié. En effet, à la première connexion, alors que le client demande la page d'accueil (en réalité, c'est le descripteur de déploiement qui le fait), la page <bienvenue.jsp> ne s'affiche pas encore parce que le client est redirigée vers la page suivante :



Cette page se nomme <authentifier.jsp>. Elle permet de savoir ce que désire faire le client dans la suite des opérations, notamment s'il désire être authentifier ou au contraire rester anonyme. Voici le codage de cette page :

authentifier.jsp

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
2    "http://www.w3.org/TR/html4/loose.dtd">
3
4  <html>
5    <head><title>Messages</title></head>
6    <body bgcolor="#<%= application.getInitParameter("couleurFond") %>">
7      <font face="Arial" style="bold">
8          <h2 align="center">Messages</h2>
9          <hr>
10         <h3><font color="green">Actuellement, vous n"êtes pas encore reconnu !</font></h3>
11
12         <form action="bienvenue.jsp" method="post">
13           <table border="1" cellspacing="3" cellpadding="2" width="90%" align="center">
14             <tr bgcolor="#FF6600"><th>Inscription</th></tr>
15             <tr><td><b>
16                 <p><input type="radio" name="authentification" value="anonyme" checked>
17                  Désirez-vous rester anonyme et donc consulter les messages prévus pour tout le monde ?</p>
18                 <p><input type="radio" name="authentification" value="nouveau">
19                  Désirez-vous vous inscrire ?</p>
20                 <p><input type="radio" name="authentification" value="personnel">
```

```
21                  Si vous êtes déjà inscrit, désirez-vous consulter vos propres messages ?</p>
22              </b></td></tr>
23              <tr bgcolor="#FF6600"><td align="center"><input type="submit" value="Valider"></td></tr>
24            </table>
25          </form>
26
27          <%@include file = "/WEB-INF/jspf/pieds.jspf" %>
28      </font>
29    </body>
30  </html>
31
```

Le contenu de cette page est très classique et n'appelle aucun commentaire particulier si ce n'est que lorsque nous validons le formulaire, nous repartons de nouveau vers la page d'accueil <bienvenue.jsp> (ligne 12).

---

bienvenue.jsp

```
 1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
 2
 3 <%! int idPersonne = 1; %>
 4 <%! String identité = "A tout le monde"; %>
 5
 6 <%
 7     Personne utilisateur = (Personne) session.getAttribute("utilisateur");
 8     String authentification = request.getParameter("authentification");
 9     if (utilisateur == null && authentification == null) {
10 %>
11     <jsp:forward page="/WEB-INF/authentifier.jsp" />
12 <%
13     } else if (utilisateur == null && authentification != null && !authentification.equals("anonyme")) {
14  %>
15     <jsp:forward page="utilisateur.jsp">
16         <jsp:param name="authentification" value="<%= authentification %>" />
17     </jsp:forward>
18 <%
19     } else if (utilisateur != null) {
20         idPersonne = utilisateur.identificateur();
21         identité = utilisateur.getPrénom()+" "+utilisateur.getNom();
22     }
23 %>
24
25 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
26
27 <html>
28   <body bgcolor="#<%= application.getInitParameter("couleurFond") %>">
29     <font face="Arial">
30
31 <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
32     <tr bgcolor="#FF6600">
33         <th>Sujet</th>
34         <th>Message</th>
35     </tr>
36     <%
37         ListeMessages listeMessages = new ListeMessages(idPersonne);
38         int ligne = 0;
39         while (listeMessages.suivant()) {
40     %>
41     <tr bgcolor="<%= ligne++ % 2 == 0 ? "#FFFF66" : "#FFCC00" %>">
42         <td><b><%= listeMessages.sujet() %></b></td>
43         <td><%= listeMessages.texte() %></td>
44     </tr>
45     <%
46         }
47         listeMessages.arrêt();
48     %>
49 </table></p>
50 </font>
51
52 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Cette page <bienvenue.jsp> devient la page principale du site. Beaucoup de lignes de code ont été rajoutées. Ainsi, maintenant cette page s'occupe de l'ensemble des redirections nécessaires avant d'avoir l'affichage proprement dit.

Lignes 3 et 4:

Déclaration des deux variables **idPersonne** et **identité** que ne avons déjà traitées et qui ont été déplacées du fragment de page <navigation.jspf>. Nous avons besoin de ces variables tout de suite après. Nous utilisons les actions **forward** dans la suite du code qui doivent être impérativement placées avant tout codage HTML. Du coup, l'inclusion du fragment de page doit être placé après ces actions, ce qui nous a obligé à retirer ces variables et de les placer directement dans la page <bienvenue.jsp>.

Lignes 6 à 10:

Sciptlet qui permet de récupérer l'objet **utilisateur** si il a été placé dans la **session**, ainsi que l'**authentification** de l'utilisateur si elle a déjà été faite. D'ailleurs, la **ligne 9** permet de contrôler si ces deux éléments et de réagir en conséquence.

Ligne 11:

La première fois (uniquement dans ce cas là d'ailleurs), aucun de ces deux objets ne sont valides. Du coup, la page <bienvenue.jsp> propose effectivement une redirection vers la page <authentifier.jsp>. Cette page a pour but de spécifier authentification qui peut prendre une des trois valeurs suivantes : **"anonyme", "nouveau", "personnel"**.

Lignes 13:

Le test de cette ligne est vrai si le client désire s'authentifier, soit par création d'un nouveau compte, soit pour consulter ses messages personnels. Dans le cas contraire, cela veut dire que le client voulait rester anonyme et donc consulter les messages prévus pour tout le monde. Dans ce dernier cas, nous n'avons plus de redirection, et donc le contenu de la page <bienvenue.jsp> est cette fois-ci traité dans sa totalité. Ainsi, l'ensemble de la page est visualisé avec le tableaux des messages prévus pour tout le monde.

Ligne 15 à 17:

Cette redirection est proposée dans le cas où l'utilisateur souhaite s'identifier. Cette fois-ci, nous proposons un paramètre à la redirection. En effet, nous devons préciser pour quelle raison l'utilisateur désire s'identifier, soit par création d'un nouveau compte, soit pour spécifier les références de l'utilisateur déjà inscrit. Pour cela, nous passons en paramètre de la redirection la valeur de la variable **authentification** qui a été initialisée à partir de la page <authentifier.jsp>. Cette redirection fait appel à la page <utilisateur.jsp> qui est maintenant capable de gérer les deux situations.

Ligne 19 à 22:

En imaginant que l'utilisateur s'est effectivement identifié, soit par création d'un nouveau compte, soit par précision des références, les variables, préalablement créée, récupèrent les informations nécessaires afin que l'affichage du tableau des messages correspond à la personne désignée.

Vu que le fragment de page a subie quelques petites modifications, je vous la propose ci-desous

navigation.jspf

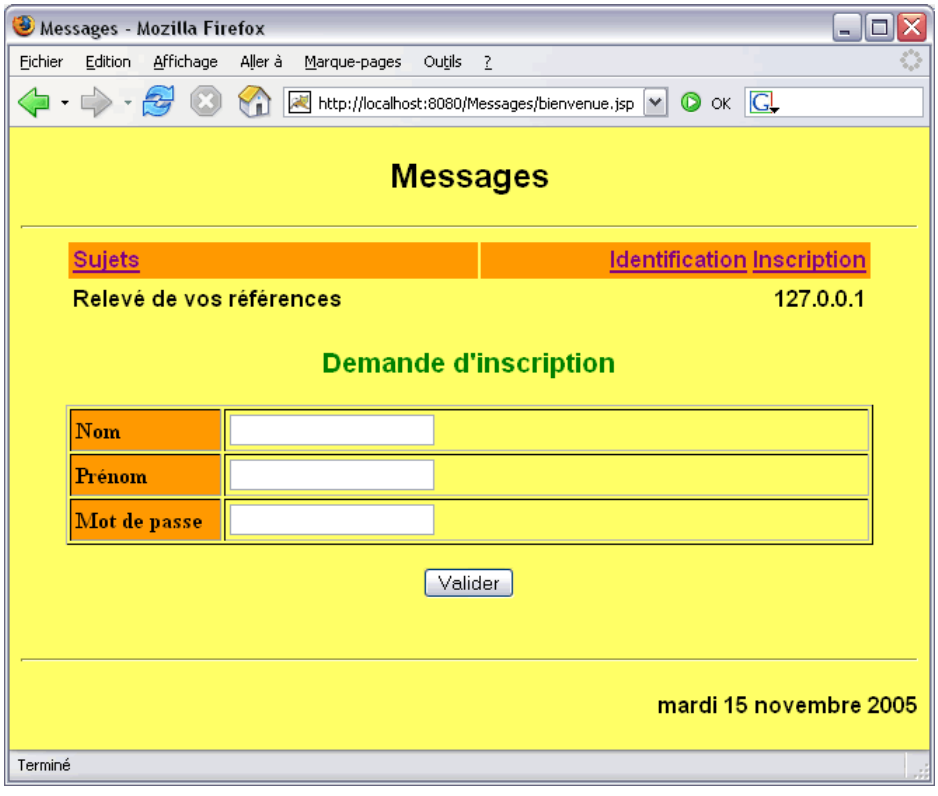
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
  <head><title>Messages</title></head>
  <body bgcolor="#<%= application.getInitParameter("couleurFond") %>">
    <font face="Arial">
        <h2 align="center">Messages</h2>
        <hr>
        <table bgcolor="1" cellpadding="3" cellspacing="2" width="90%" align="center">
            <tr bgcolor="#FF9900">
                <th align="left"><a href="bienvenue.jsp">Sujets</th>
                <th align="right">
                    <% if (idPersonne == 1) { %>
                        <a href="utilisateur.jsp?authentification=personnel">Identification</a>
                        <a href="utilisateur.jsp?authentification=nouveau">Inscription</a>
                    <% }
                       else { %>
                         <a href="#">Nouveau</a>
                         <a href="#">Modifier</a>
                         <a href="#">Enlever</a>
                    <% } %>
                </th>
            </tr>
            <tr>
                <th align="left"><%= identité %></th>
                <th align="right"><%= request.getRemoteHost() %></th>
            </tr>
        </table>

```

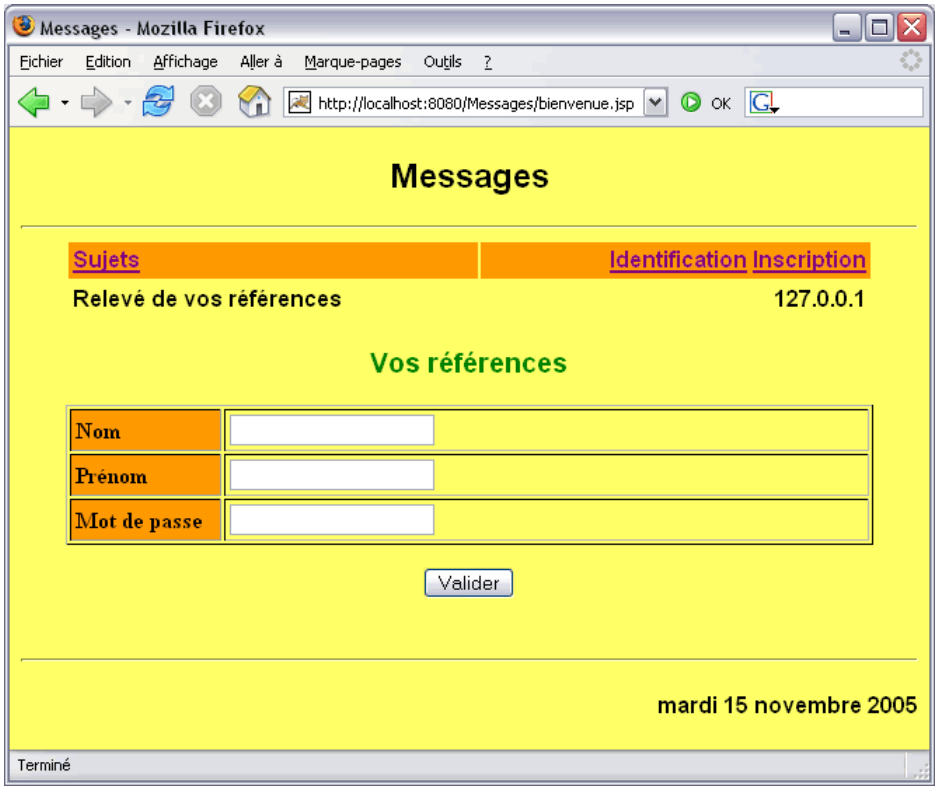
Lignes 15 et 16:

Lien vers la même page <utilisateur.jsp>. Le paramètre **authentification** permet de préciser dans quel contexte nous utilisons cette page.

Nous allons analyser maintenant la page <utilisateur.jsp> qui s'appelait au préalable <nouvelutilisateur.jsp>. Cette page est appelée soit en venant de la redirection proposée par la page d'accueil <bienvenue.jsp>, soit par le lien proposé par le fragment de page <navigation.jspf>.

Cette page joue également un rôle important puisqu'elle s'occupe de recenser l'identité du client afin de rediriger ces informations suivant le cas, soit vers la création d'un nouvel utilisateur, soit vers la vérification de l'existence de l'opérateur déjà inscrit. Cette page peut donc traiter deux cas de figure :





D'un point de vue visuel, très peu de changement, mis à part l'intitulé de la page qui peut prendre l'un des deux textes suivants : "Demande d'inscription" ou "Vos références". Voici ci-dessous le codage correspondant :

utilisateur.jsp

```
 1 <%@ page errorPage = "/WEB-INF/erreur.jsp" import = "bd.*" %>
 2 <%! int idPersonne = 1; %>
 3 <%! String identité = "Relevé de vos références"; %>
 4 <%@ include file = "/WEB-INF/jspf/navigation.jspf" %>
 5
 6 <%! boolean nouveau; %>
 7 <% nouveau = request.getParameter("authentification").equals("nouveau"); %>
 8
 9 <h3 align="center">
10     <font color="green"><%= (nouveau ? "Demande d"inscription" : "Vos références") %></font>
11 </h3>
12
13 <form action="<%= (nouveau ? "validerutilisateur.jsp": "controleidentite.jsp") %>" method="post">
14     <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
15         <tr>
16             <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
17             <td><input type="text" name="nom"></td>
18         </tr>
19         <tr>
20             <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
21             <td><input type="text" name="prénom"></td>
22         </tr>
23         <tr>
24             <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
25             <td><input type="password" name="motDePasse"></td>
26         </tr>
27     </table></p>
28     <p align="center"><input type="submit" value="Valider"></p>
29 </form>
30
31 <%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Lignes 2 à 4:

Dans cette page, nous n'avons pas de redirection au moyen de l'action **forward**. Nous pouvons donc, dès le départ, inclure le fragment de page <navigation.jspf>. Toutefois, il est nécessaire de déclarer, comme pour la page d'accueil, les deux variables **idPersonne** et **identité** puisqu'elles ne sont plus présentes dans le fragment de page. Du coup, nous pouvons en profiter pour mettre choisir une valeur d'**identité** différente. En l'occurence, ici, **identité** prend la valeur **"Relevé de vos références"**.

Lignes 6 et 7:

Déclaration suivi de l'initialisation de l'attribut **nouveau**. Cet attribut est initialisé à partir du paramètre **authentification** placé dans la requête, soit au moyen de l'action **forward** si nous venons de la page d'accueil <bienvenue.jsp>, soit au niveau du lien proposé par le fragment de page <navigation.jspf>.

Ligne 10:

Affichage de l'intitulé de la page suivant la valeur du paramètre **authentification** de la requête envoyée, c'est-à-dire suivant ce que l'utilisateur désire faire : s'inscrire ou se connecter au moyen de ses références.

Lignes 13:

Lancement et exécution de la page concernée après avoir réalisé la saisie et confirmé les informations en cliquant sur le bouton **"Valider"**. C'est la page <validerutilisateur.jsp> qui est sollicitée si c'est une demande d'inscription, ou la page <controleidentite.jsp> dans le cas contraire.

Voici, ci-dessous, avec les toutes petites modifications apportées :

validerutilisateur.jsp

```jsp
<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>
<%! int idPersonne = 1; %>
<%! String identité = "Confirmation de vos références"; %>
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<h3 align="center">Confirmation de votre demande d"inscription</h3>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />

    <p><table border="1" cellpadding="3" cellspacing="2" width="90%" align="center">
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Nom</b></td>
            <td><jsp:getProperty name="utilisateur" property="nom" /></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Prénom</b></td>
            <td><jsp:getProperty name="utilisateur" property="prénom" /></td>
        </tr>
        <tr>
            <td bgcolor="#FF9900" width="100"><b>Mot de passe</b></td>
            <td><jsp:getProperty name="utilisateur" property="motDePasse" /></td>
        </tr>
    </table></p>
</jsp:useBean>

<h3 align="center">
<% if (!utilisateur.enregistrer()) {  %>
     <font color="red">ATTENTION : Utilisateur déja enregistré</font>
<%
        session.removeAttribute("utilisateur");
    } else {
%>
     <font color="green">Nouvel utilisateur enregistré</font>
<% } %>
</h3>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Lignes 2 à 4:

Là aussi, nous avons besoin de nous préoccuper des attributs **idPersonne** et **identité** avant de solliciter le fragment de page **<navigation.jspf>** correspondant au menu.

Lignes 31:

Si la demande d'inscription porte sur un utilisateur déjà connu, il est impératif d'enlever de la **session** l'objet **utilisateur**.

Il nous manque la page qui va s'occuper de savoir si l'identité de la personne qui désire consulter ses propres messages est déjà inscrite dans la base de données.

controleidentite.jsp

```jsp
<%@ page errorPage = "/WEB-INF/erreur.jsp" import="bd.*" %>

<jsp:useBean id="utilisateur" class="bd.Personne" scope="session">
    <jsp:setProperty name="utilisateur" property="*" />
</jsp:useBean>

<%  if (!utilisateur.authentifier()) {
        session.removeAttribute("utilisateur");
    }
%>

<jsp:forward page="bienvenue.jsp" />

```

Lignes 3 à 5:

Comme pour la page **<validerutilisateur.jsp>**, récupération des paramètres de la requête issus de la page **<utilisateur.jsp>** au travers de la construction du JavaBean **utilisateur** qui est automatiquement intégré dans la **session**.

Lignes 7 à 9:

Contrôle de la présence de l'utilisateur au moyen de la méthode **authentifier()** créée à cet effet. Si l'utilisateur n'est pas connu, il faut impérativement l'enlever de la **session**.

Ligne 12:

Redirection vers la page d'accueil **<bienvenue.jsp>** au moyen de l'action **forward** afin que les messages personnels de l'utilisateur soient automatiquement affichés.

Personne.java

```java
package bd;

import java.sql.SQLException;

public class Personne extends ConnexionBD {
   private String nom;
   private String prénom;
   private String motDePasse;

   public Personne() { }

```

```
12     public String getNom() {
13        return this.nom;
14     }
15     public void setNom(String nom) {
16        this.nom = nom.toUpperCase();
17     }
18
19     public String getPrénom() {
20        return this.prénom;
21     }
22     public void setPrénom(String prénom) {
23        this.prénom  = prénom.substring(0, 1).toUpperCase() + prénom.substring(1, prénom.length()).toLowerCase();
24     }
25
26     public String getMotDePasse() {
27        return this.motDePasse;
28     }
29     public void setMotDePasse(String motDePasse) {
30        this.motDePasse = motDePasse;
31     }
32
33     public boolean enregistrer() {
34        if (existeDéjà())
35           return false;
36        else {
37           miseAJour("INSERT INTO personne (nom, prénom, motDePasse) VALUES
(\""+nom+"\",\""+prénom+"\",\""+motDePasse+"\")");
38           return true;
39        }
40     }
41     private boolean existeDéjà() {
42        lire("SELECT * FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\"");
43        return suivant();
44     }
45
46     public int identificateur(){
47        lire("SELECT idPersonne FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\"");
48        suivant();
49         try {
50           return résultat.getInt("idPersonne");
51         }
52         catch (SQLException ex) {
53           return 1;
54         }
55     }
56     public boolean authentifier() {
57        lire("SELECT * FROM personne WHERE nom=\""+nom+"\" AND prénom=\""+prénom+"\" AND motDePasse=\""+motDePasse+"\"");
58        return suivant();
59     }
60  }
61
```

Lignes 56 à 59:

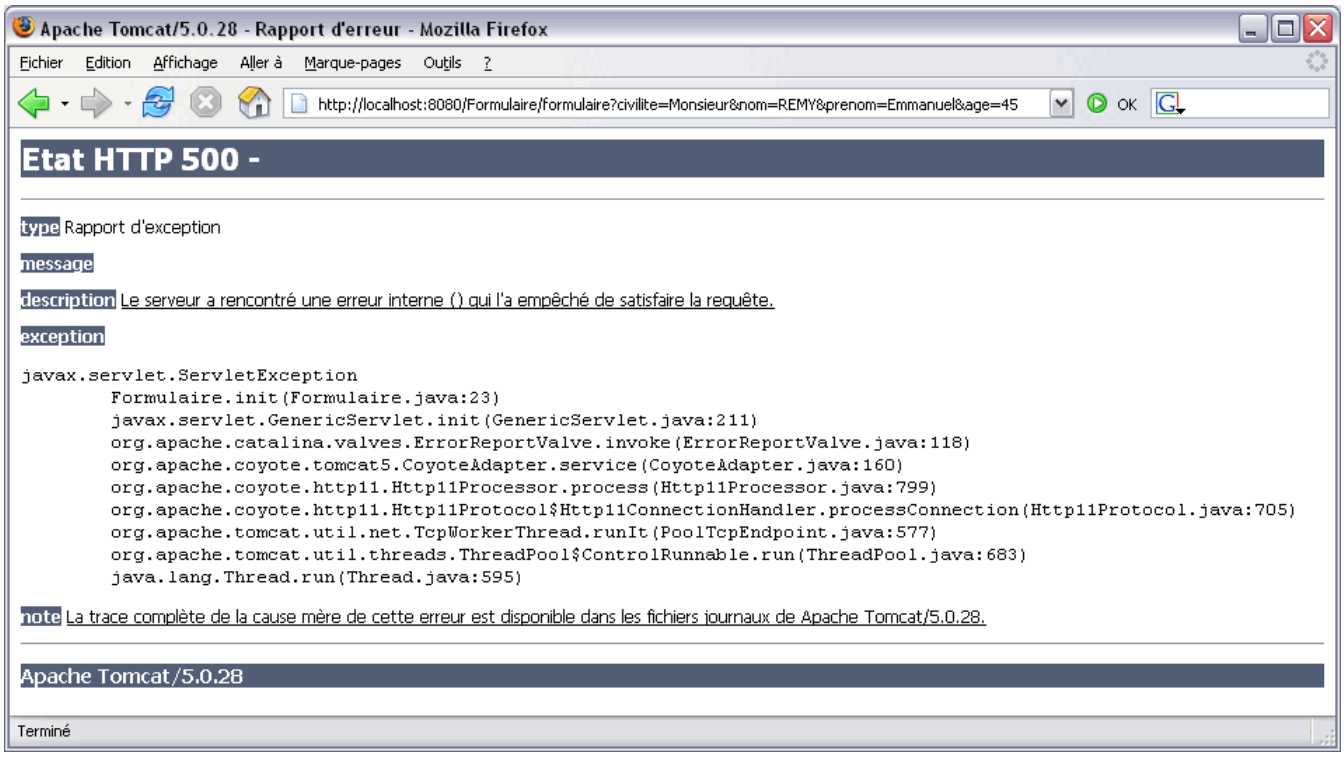
Définition de la méthode **authentifier()** de la classe **Personne**.

Conclusion

Le contrôle du déroulement de l'application est devenu compliqué puisque nous passons systématiquement par la page d'accueil <bienvenue.jsp>. Cette dernière est devenue deux fois plus importante avec un grand mélange de balisage et de code Java. La principale raison du passage par la page d'accueil est de donner plusieurs exemples de l'utilisation de <jsp:forward>. Toutefois, il existe une autre raison, il s'agit d'un exemple très simple de ce que nous appelons le Modèle 2, ou architecture Modèle-Vue-Contrôleur, ou MVC. Dans l'architecture Modèle 2, un composant agit comme contrôleur de trafic, redirigeant les requêtes vers les composants les plus aptes à les traiter. Cette technologie sera mis en oeuvre dans une prochaine étude.

Erreurs et exceptions

Il est possible que notre application Web ne fonctionnement correctement lorsque par exemple le serveur de base de données est momentanément hors connexion. Voici ce que le client peut recevoir au travers de son navigateur. Il s'agit ici d'un cas exceptionnel. La technique qui s'occupe de ce genre de cas s'appelle la gestion d'exception. Suivant les serveurs, l'utilisateur voit s'afficher la trace de l'exception comme ci-dessous ou alors il n'obtient aucune réponse du serveur.



Dans tous les cas, l'utilisateur a le sentiment que l'application est défectueuse, ce qui est le cas puisque pour l'instant, il n'y a pas de véritable gestion d'exception.

Cette situation n'est pas acceptable. Nous avertissons le client lorsque l'enregistrement c'est bien déroulé. Nous devons également l'avertir lorsqu'un problème s'est rencontré et lui proposer une solution éventuelle, notamment lors de la mauvaise saisie d'un champ. C'est la moindre des choses. Bref, nous devons gérer l'exception.

Règle de bonne conduite : A moins que l'exception soit une IOException (communication interrompue) survenant lors de l'écriture de la réponse, le client devrait toujours recevoir une réponse intelligible.

Les application Web peuvent traiter les esceptions de différentes façons. De toute évidence, certaines d'entre elles peuvent être traitées préventivement lors du développement en ajoutant des procédures de validation et des blocs try...catch. Ces techniques empêchent les exceptions de se produire. Cependant, nous avons besoin d'un moyen permettant de traiter une éventuelle exception imprévue. Enfait, nous en avons deux à notre disposition :

1. *La directive page ;*
2. *Le descripteur de déploiement.*

La directive page

Nous avons déjà vu comment inclure une directive page dans une JSP. Cette directive peut avoir un attribut errorPage. Lorsqu'une exception se produit et n'est pas interceptée, le serveur retourne la page indiquée. Il est ainsi possible d'utiliser une page d'erreur différente pour chaque composant de l'application (chaque page JSP). Dans notre application Web, pour toutes las pages JSP construites, nous avons toujours fait appel à la même page d'erreur. Voici, systématiquement,la syntaxe que nous avons utilisé :

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" %>
```

La valeur de l'attribut est tout simplement le chemin d'accès de la page d'erreur. L'inconvénient de cette méthode est que la même page d'erreur est envoyée quelle que soit l'exception.

Le descripteur de déploiement

Le descripteur de déploiement permet de désigner des gestionnaires d'erreurs pour toute l'application. Il est ainsi possible d'avoir des pages d'erreur différentes en fonction du type d'exception. Si une certaine exception se produit dans l'application Web, le descripteur de déploiement désigne la page d'erreur qui doit être renvoyée au client. Bien entendu, une page d'erreur définie dans une page a la priorité sur celle indiquée par le descripteur de déploiement.

Dans le descripteur de déploiement, l'élément qui décrit les pages d'erreur est <error-page>. Cet élément <error-page> dispose ensuite de deux types de sous-élements : <exception-type> et <code-type>

Deux scenarii sont alors possibles. Soit nous indiquons le type d'erreur exacte, soit un code d'erreur délivré par le protocole HTTP (500 par exemple) qui correspond alors à une erreur plus générique. Le conteneur de JSP prend toujours, dans le descripteur de déploiement, en premier la page qui correspond pile à l'erreur. S'il ne la trouve pas, il se rabat alors sur le code d'erreur qui englobe un ensemble d'erreurs potentielles.

```
<error-page>
                <exception-type>java.lang.NumberFormatException</exception-type>
                <location>/WEB-INF/NombreIncorrect.html</location>
</error-page>
```

ou/et les sous-éléments suivant :

```
<error-page>
                <code-type>500</exception-type>
                <location>/WEB-INF/ErreurServeur.html</location>
</error-page>
```

La première écriture indique que si une erreur de type numérique intervient, c'est la page <NombreIncorrect.html> qui va être envoyée au client. Dans le deuxième cas, c'est l'erreur 500 donnée par le protocole HTTP qui permet d'envoyer au client la page <ErreurServeur.html>.

L'erreur 500 devrait systématiquement être traitée de cette façon. Ce code indique une erreur interne du serveur que celui-ci ne peut traiter. Il peut s'agir d'un problème de compilation d'une JSP ou d'une exception dans une servlet.

En spécifiant une page d'erreur, vous vous assurez que le client recevra une page lisible et correctement mis en forme, plutôt qu'une trace cabalistique.
.

Je ne vais pas proposer d'exemple de gestion d'erreur. Nous l'avons déjà mis en oeuvre lors de l'étude sur les servlets. La technique demeure identique.

Les applets

Préalablement, nous avions vu que nous pouvions placer des applets dans des pages Web statiques HTML. Il est bien entendu également possible de les mettre au sein de pages Web dynamiques JSP. L'intérêt de placer des applets dans des pages Web, c'est qu'elles permettent d'avoir une plus grande convivialité ainsi qu'une plus grande richesse au niveau de l'interface graphique.

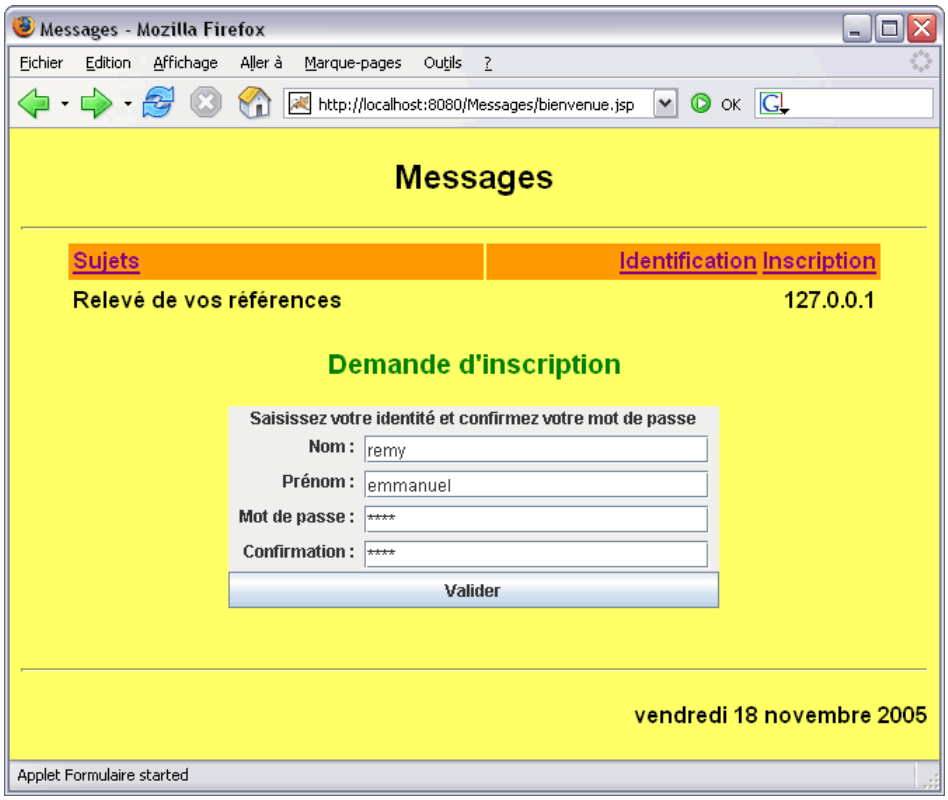
Attention toutefois, le navigateur, dans ce cas là, n'est plus un environnement léger comme dans le cas de page Web classiques.
.

Une applet est en fait une application classique (avec des menus, des boutons, etc...) qui tourne au sein d'une page Web. Cette applet est téléchargée sur l'ordinateur client juste au moment du chargement de la page Web qui sert de conteneur. Grâce à cette applet, il est possible d'effectuer un grand nombre de prétraitement avant la validation de la requête qui nous permet de passer vers la page Web suivante. Par exemple, nous pourrions avoir un traitement d'image à effectuer à l'aide de l'applet avant que cette dernière ne soit stockée dans la base de données au travers de l'application Web.

Etude de cas

Dans les pages JSP, les applet s'utilise de la même façon que pour les pages HTML classiques. Du coup, je ne vais pas en faire une étude particulière. En effet, deux études ont déjà été consacrées sur ce sujet.

Nous allons juste reprendre l'application Web précédente sur la gestion de messagerie. Dans cette application Web, nous allons modifier la page Web <utilisateur.jsp> qui s'occupe de la saisie de l'identité de l'utilisateur. Nous allons remplacer le formulaire HTML par une applet équivalente. Cette applet permettra en plus de contrôler le mot de passe saisie.



Nous aurions pu rajouter cette fonctionnalité supplémentaire avec les pages JSP déjà créées. C'est juste l'occasion de voir l'intégration d'une applet dans une application Web avec des pages Web dynamiques.

Restructuration de la page JSP <utilisateur.jsp>

Dans cette page, nous enlevons tout le balisage qui correspond à la balise <form> et nous le remplaçons par la balise <applet>. Nous en profitons pour passer en paramètre de l'applet le type d'authentification de la personne, à l'aide de la balise <param>, qui permettra ensuite d'envoyer la requête issue de l'applet sur la bonne page JSP. En effet, le paramètre authentification détermine s'il s'agit d'une création d'un nouveau compte, ou au contraire de l'identification de l'opérateur.

utilisateur.jsp

```
<%@ page errorPage = "/WEB-INF/erreur.jsp" import = "bd.*" %>
<%! int idPersonne = 1; %>
<%! String identité = "Relevé de vos références"; %>
<%@ include file = "/WEB-INF/jspf/navigation.jspf" %>

<%! boolean nouveau; %>
<%! String authentification; %>
<%
    authentification = request.getParameter("authentification");
    nouveau = authentification.equals("nouveau");
%>

<h3 align="center">
    <font color="green"><%= (nouveau ? "Demande d"inscription" : "Vos références") %></font>
</h3>

<p align="center"><applet code="Formulaire" width="350" height="144">
    <param name="authentification" value="<%= authentification %>">
</applet></p>

<%@ include file = "/WEB-INF/jspf/pieds.jspf" %>
```

Applet Formulaire.java

Formulaire.java

```
 1 import java.awt.*;
 2 import java.awt.event.*;
 3 import java.io.UnsupportedEncodingException;
 4 import java.net.MalformedURLException;
 5 import java.net.*;
 6 import javax.swing.*;
 7
 8 public class Formulaire extends JApplet implements ActionListener {
 9    private JPanel panneau = new JPanel();
10    private JPanel panneauOuest = new JPanel();
11    private JLabel message = new JLabel("Saisissez votre identité et confirmez votre mot de passe");
12    private JTextField nom = new JTextField();
13    private JTextField prénom = new JTextField();
14    private JPasswordField motDePasse = new JPasswordField();
15    private JPasswordField confirmation = new JPasswordField();
16    private JButton valider = new JButton("Valider");
17    private String authentification;
18
19    public void init() {
20       authentification = this.getParameter("authentification");
21       valider.addActionListener(this);
22
23       panneauOuest.setPreferredSize(new Dimension(90, 50));
24       panneauOuest.setLayout(new GridLayout(4, 1));
25       message.setHorizontalAlignment(SwingConstants.CENTER);
26       JLabel lnom = new JLabel("Nom :");
27       lnom.setHorizontalAlignment(SwingConstants.RIGHT);
28       panneauOuest.add(lnom);
29       JLabel lprénom = new JLabel("Prénom :");
30       lprénom.setHorizontalAlignment(SwingConstants.RIGHT);
31       panneauOuest.add(lprénom);
32       JLabel lmotDePasse = new JLabel("Mot de passe :");
33       lmotDePasse.setHorizontalAlignment(SwingConstants.RIGHT);
34       panneauOuest.add(lmotDePasse);
35       JLabel lconfirmation = new JLabel("Confirmation :");
36       lconfirmation.setHorizontalAlignment(SwingConstants.RIGHT);
37       panneauOuest.add(lconfirmation);
38
39       nom.setColumns(22);  panneau.add(nom);
40       prénom.setColumns(22);  panneau.add(prénom);
41       motDePasse.setColumns(22);  panneau.add(motDePasse);
42       confirmation.setColumns(22);  panneau.add(confirmation);
43       panneau.add(valider);
44
45       this.getContentPane().add(panneauOuest, BorderLayout.WEST);
46       this.getContentPane().add(panneau);
47       this.getContentPane().add(message, BorderLayout.NORTH);
48       this.getContentPane().add(valider, BorderLayout.SOUTH);
49    }
50
51    public void actionPerformed(ActionEvent e) {
52       if (motDePasse.getText().equals(confirmation.getText())) envoyerSaisie();
53       else message.setText("ATTENTION : mots de passe différents");
54    }

55    private void envoyerSaisie() {
56       String suiteURL = authentification.equals("nouveau") ? "validerutilisateur.jsp": "controleidentite.jsp";
57       URL url = null;
58       try {
59          suiteURL = suiteURL+"?nom="+URLEncoder.encode(nom.getText())
60             +"&"+URLEncoder.encode("prénom")+"="+URLEncoder.encode(prénom.getText())
```

```
61              +"&motDePasse="+URLEncoder.encode(motDePasse.getText());
62           url = new URL(this.getDocumentBase(), suiteURL);
63        }
64        catch (MalformedURLException ex) { System.out.println("Impossible d"atteindre la page"); }
65        this.getAppletContext().showDocument(url, "_parent");
66     }
67  }
68
```

Lignes 8:

Mise en place de la classe **Formulaire** représentant l'applet par dérivation de la classe **JApplet**. Cette classe écoute l'événement **validation** grâce à l'implémentation de **ActionListener**. Ce système d'écoute d'événement est validé par la **ligne 21** qui indique quel est le composant qui doit pris en compte par le système d'écoute, ici le bouton **valider**.

Lignes 9 à 17:

Création de tous les composants graphiques utiles à l'applet.

Ligne 19 à 49:

Initialisation des composants afin de les placer correctement sur l'applet.

Ligne 51 à 54:

Puisque nous avons demander à implémenter un écouteur d'événement de type **ActionListener**, il est du devoir de l'applet de redéfinir la méthode **actionListener()**. C'est dans cette dernière que nous allons stipuler le traitement à réaliser lorsque l'utilisateur validera sa saisie par l'action sur le bouton **valider**, soit par un click sur le bouton lui-même, soit en appuyant sur la touche **"Entrée"** du clavier si ce bouton a le focus. C'est dans cette méthode que nous contrôlons si les deux mots de passe sont identiques. Si c'est le cas, nous activons la méthode **envoyerSaisie()**. Dans le cas contraire, nous avertissons l'usager de la discordance des deux mots de passe.

Ligne 55 à 66:

C'est dans cette méthode que nous préparons la requête en récupérant les différents paramètres requis et en activant la bonne page JSP en lieu et place de la page actuelle **<utilisateur.jsp>**. Nous en profitons pour coder l'URL afin que les lettres accentuées soient correctement interprétés.