

Les Servlets

Servlets et pages Web dynamiques

- Les **Servlets** représentent une solution technologie Java de la programmation avec CGI. Il s'agit de programmes exécutés sur un serveur Web, qui servent de couche intermédiaire entre une requête provenant d'un navigateur Web et un autre service HTTP, comme des bases de données ou des applications du serveur HTTP. Leur tâche est de :
 1. **Lire toutes les données envoyées par l'utilisateur** : Ces données sont typiquement saisies dans un formulaire sur une page Web, mais elles peuvent également provenir d'une applet Java ou d'un programme client HTTP particulier.
 2. **Chercher d'autres informations sur la requête, à l'intérieur de cette requête HTTP** : Ces informations contiennent des détails sur les capacités du navigateur, sur les cookies, sur le nom de l'hôte du programme envoyant la requête, etc.
 3. **Générer des résultats** : Ce processus peut nécessiter une communication avec la base de données, ou en invoquant une ancienne application, ou encore en calculant directement la réponse.

Servlets et pages Web dynamiques

4. **Formater le résultat dans un document** : Dans la plupart des cas, cela impliquera l'incorporation des informations dans une page HTML.
5. **Définir les paramètres de la réponse HTTP appropriés** : Cela signifie qu'il faut indiquer au navigateur le type de document renvoyé (c'est à dire HTML), définir les cookies, mémoriser les paramètres, ainsi que d'autres tâches.
6. **Renvoyer le document au client** : Ce document peut être envoyé au format texte (HTML), au format binaire (comme pour des images GIF), ou même dans un format compressé comme gzip, qui en fait un e couche venant recouvrir un autre format sous-jacent.

HTTP et les serveurs

- Bien que les servlets aient été conçues originellement pour travailler avec tous les types de serveurs, elles ne sont employées en pratique qu'avec les serveurs Web.
- L'API **Servlet** contient une classe nommée **HttpServlet** spécialisée dans le traitement du protocole.
- **Protocole HTTP**
 - Définit la structure des requêtes qu'un client peut envoyer à un serveur Web, le format des paramètres pouvant accompagner ces requêtes et la façon dont le serveur doit y répondre.
 - Les servlets HTTP emploient le même protocole pour gérer les requêtes de service et envoyer les réponses aux clients. Il est donc important de bien comprendre les éléments fondamentaux du protocole HTTP pour maîtriser l'utilisation de servlets.

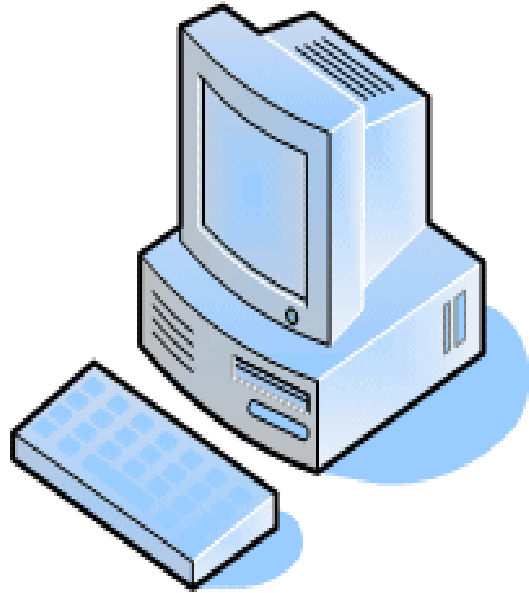
Le modèle Servlet et les servlets HTTP

- Lorsqu'un client envoie une requête au serveur Web et que celui-ci détermine que la requête est destinée à une servlet, il la passe au conteneur de servlets.
- Le conteneur de servlets est le programme responsable du chargement, de l'initialisation, de l'appel et de la destruction des instances de servlets.

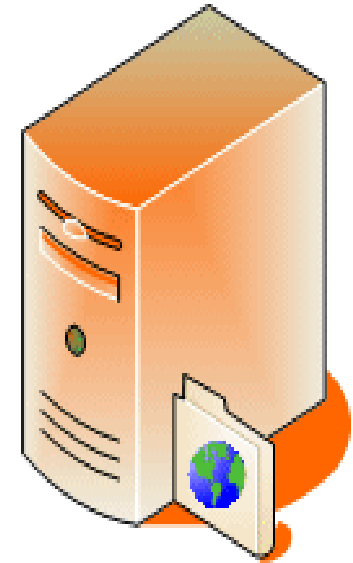
Le modèle Servlet et les servlets HTTP

- Lorsque le conteneur de servlets reçoit la requête, il en analyse l'URI, les entêtes et le corps et stocke toutes les données dans un objet implémentant l'interface `javax.servlet.ServletRequest`.
- Il crée également une instance d'un objet implémentant l'interface `javax.servlet.ServletResponse`. Cet objet encapsule la réponse qui sera envoyée au client.
- Le conteneur appelle ensuite une méthode de la classe de la servlet, en lui passant les objets requête et réponse.
- La servlet traite la requête et renvoie la réponse au client.

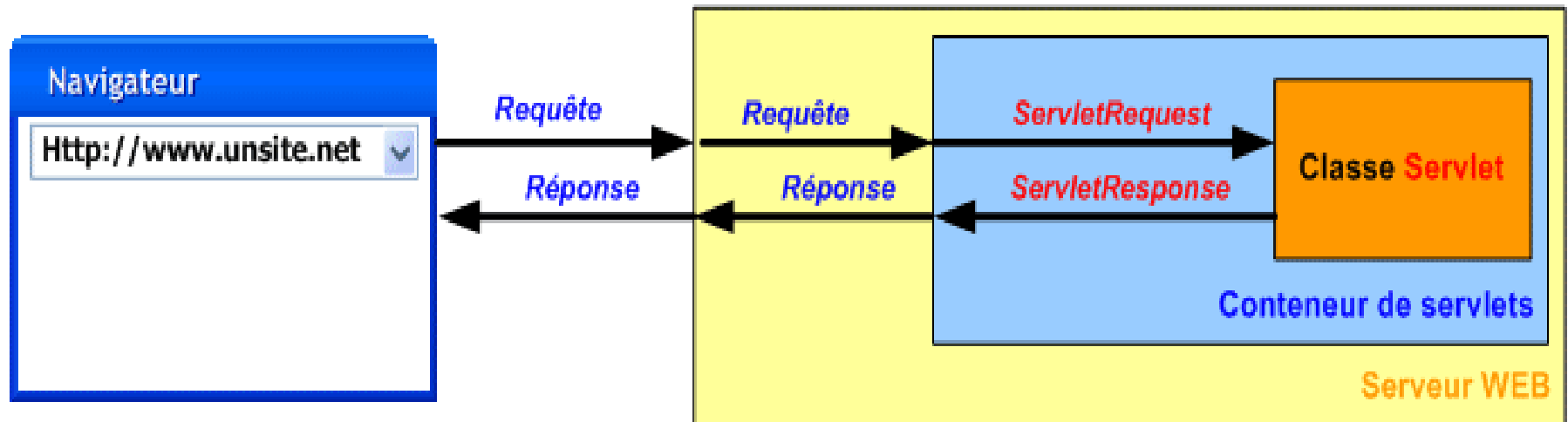
Le modèle Servlet et les servlets HTTP



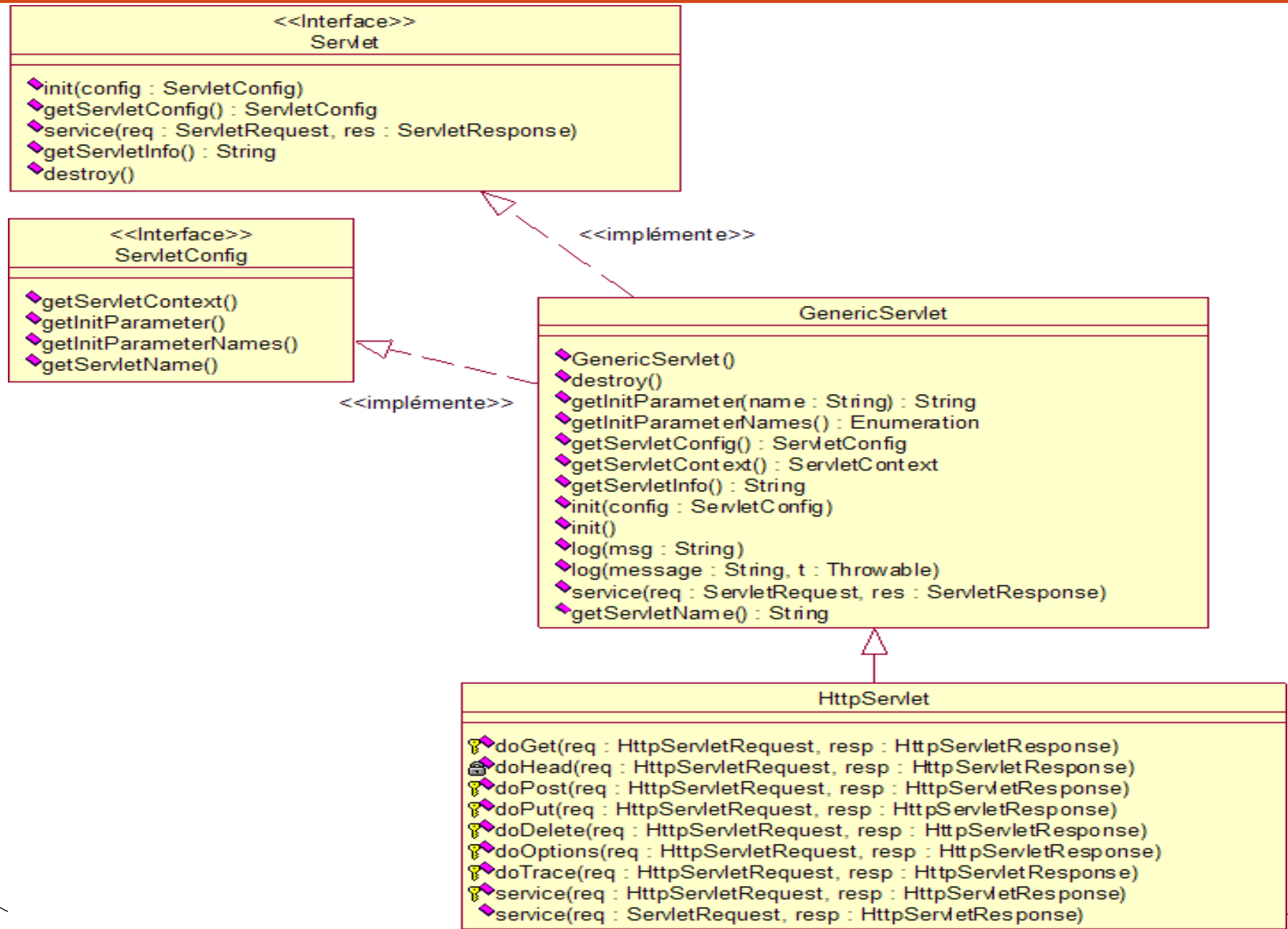
Client



Serveur Web



servlet



La méthode `service()`

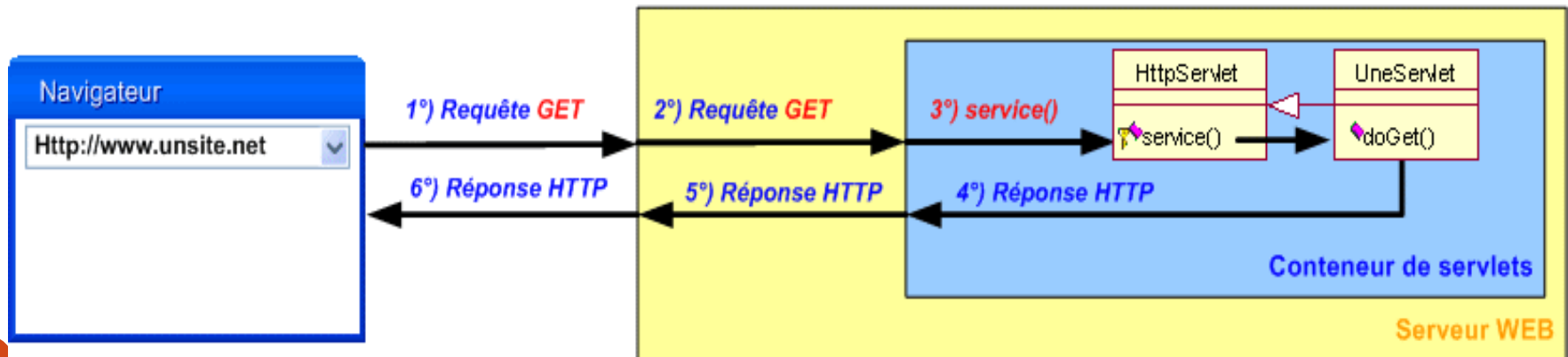
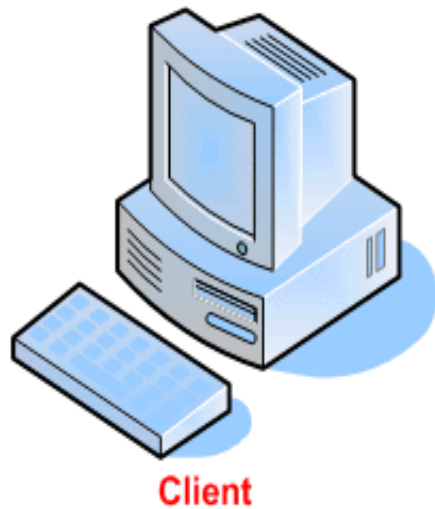
- Dans l'interface `Servlet`, la seule méthode qui gère les requêtes est la méthode **`service()`**. Lorsqu'un conteneur reçoit une requête pour une servlet, il appelle systématiquement sa méthode **`service()`**. Comme pour toutes les interfaces, une servlet implémentant l'interface `Servlet` doit obligatoirement fournir une implémentation de toutes les méthodes déclarées, et à fortiori redéfinir la méthode `service()`.

Les méthode doGet() et doPost()

- Les **HttpServlet** sont conçues pour répondre aux requêtes HTTP. Elles doivent donc traiter les requêtes **GET, POST, HEAD, etc.** La classe `HttpServlet` définit donc des méthodes supplémentaires. La méthode **doGet()** traite les requêtes GET et la méthode **doPost()** les requêtes POST. Il existe ainsi autant de méthode **doXXX()** qu'il y a de type de requêtes HTTP.
- En tant que programmeur, votre rôle consiste à développer une nouvelle servlet adaptée à la situation qui hérite de la classe `HttpServlet`, et de redéfinir uniquement les méthodes dont vous avez besoin. Le plus souvent, il s'agira de **doGet()** et de **doPost()**.

Les méthode doGet() et doPost()

- Sur cette figure, deux classes ont été représentées : HttpServlet et UneServlet. En fait, il s'agit d'un seul et même objet instance de UneServlet. Cette dernière récupère par héritage la méthode service() issue de HttpServlet.



Les objets **request** et **response**

- La signature des méthodes doXXX() est la suivante :

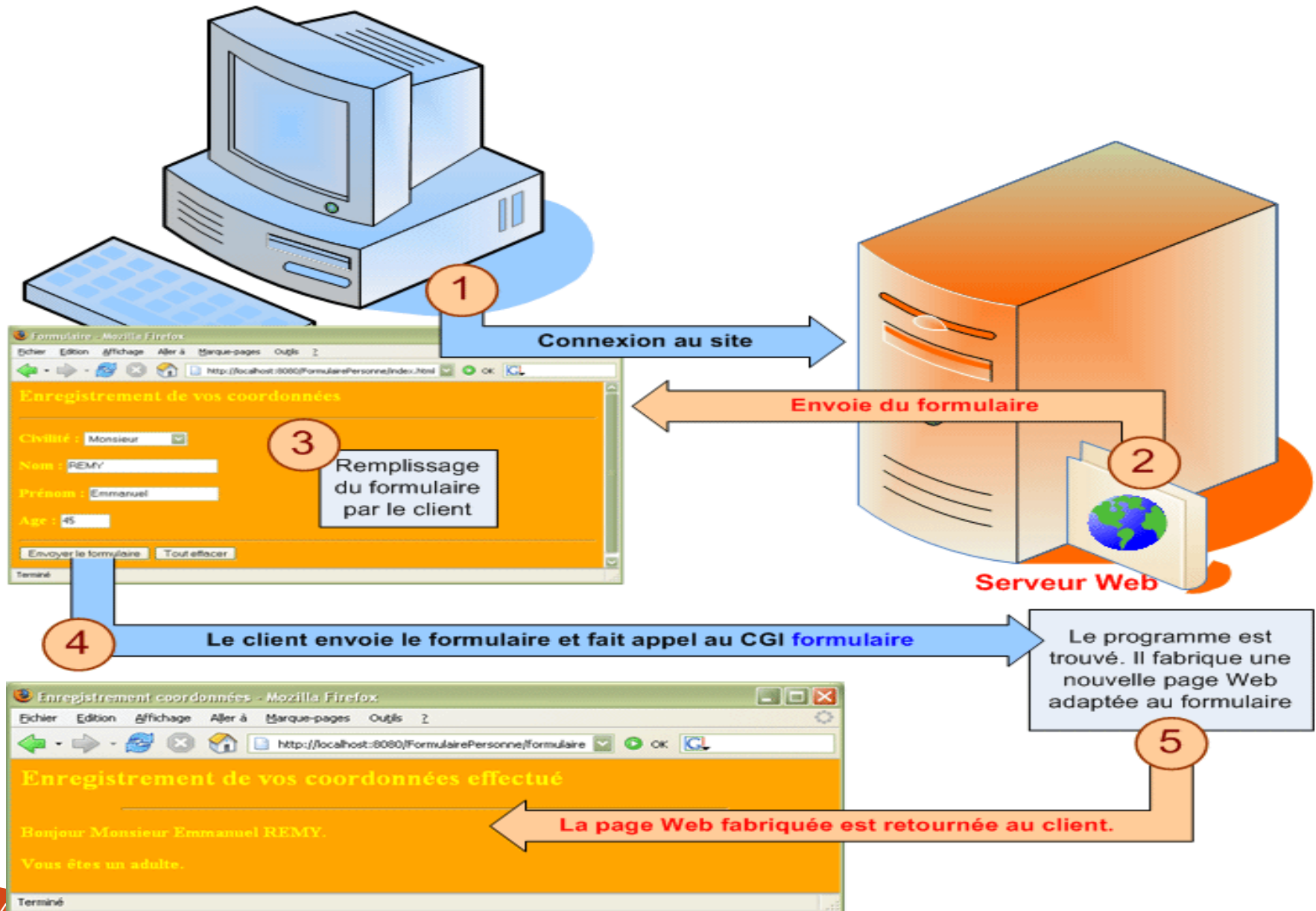
```
public void doXXX(HttpServletRequest  
request,           HttpServletResponse  
response)
```

- Chaque méthode (doPost(), doGet(), etc.) prend deux paramètres. L'objet **HttpServletRequest** encapsule la requête envoyée au serveur. Il contient toutes les données de la requête, ainsi que certains en-têtes.
- Les méthodes de l'objet request permettent d'accéder à ces données.
- Les méthodes de l'objet **HttpServletResponse** encapsule la réponse au client.

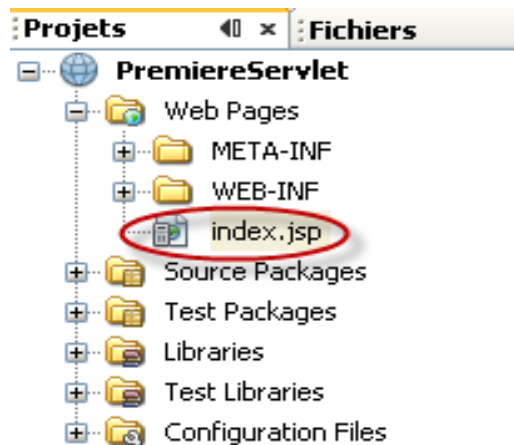
Mise en œuvre d'une servlet

1. Dans un premier, le client se connecte au site désiré en donnant la bonne URL dans la zone d'adresse du navigateur.
 - Le fait de valider cette URL, une requête est envoyée au serveur Web présent dans ce site.
2. Ce dernier envoie une page Web statique au format html pour que le client puisse s'enregistrer.
3. Le client remplit son formulaire d'enregistrement.
4. Lorsque le client a fini de remplir son formulaire, il clique sur le bouton "Envoyer le formulaire".
 - Une nouvelle requête est envoyée en demandant à une (servlet) de traiter l'ensemble des informations données par le formulaire.
5. Le serveur Web exécute la servlet demandée.
 - La servlet produit une page Web dynamique (elle n'existait pas auparavant) en correspondance des informations délivrées par le client

Mise en œuvre d'une servlet



Mise en œuvre d'une servlet



```
<html>

<head><title>Formulaire</title></head>

<body bgcolor="orange" text="yellow">

<h2>Enregistrement de vos coordonnées</h2><hr>

<form method="get" action="formulaire">

  <h3>Civilit&ecute; :

  <select name="civilite">

    <option>Monsieur</option>

    <option>Madame</option>

    <option>Mademoiselle</option>

  </select></h3>

  <h3>Nom : <input type="text" name="nom" size="24"></h3>

  <h3>Pr&ecute;nom : <input type="text" name="prenom"></h3>

  <h3>Age : <input type="text" name="age" size="5"></h3>

  <hr /><input type="submit" value="Envoyer le formulaire">

    <input type="reset" value="Tout effacer">

</form>

</body>

</html>
```

Mise en œuvre d'une servlet



http://localhost:8084/PremiereServlet/

Enregistrement de vos coordonnées

Civilité : ▼

Nom :

Prénom :

Age :

Mise en œuvre d'une servlet

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

public class Formulaire extends HttpServlet {

//Traiter la requête HTTP Get

public void doGet(HttpServletRequest request, HttpServletResponse response) throws
    ServletException, IOException {
    response.setContentType("text/html"); // type MIME pour http →Page HTML
    PrintWriter Sortie = response.getWriter(); Sortie.println("<html>");
    Sortie.println("<head><title>Enregistrement coordonnées</title></head>");
    Sortie.println("<body bgcolor=orange text=yellow>");
    Sortie.println("<h2>Enregistrement de vos coordonnées effectué</h2>");
    Sortie.println("<hr width=75%>");

    Sortie.print("<p><b>Bonjour " + request.getParameter("civilite") + " ");
    Sortie.print(request.getParameter("prenom") + " ");
    Sortie.println(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 += "une personne du troisième âge.";
    Sortie.println("<p>" + message + "</b></body></html>");
} //Traiter la requête HTTP post

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

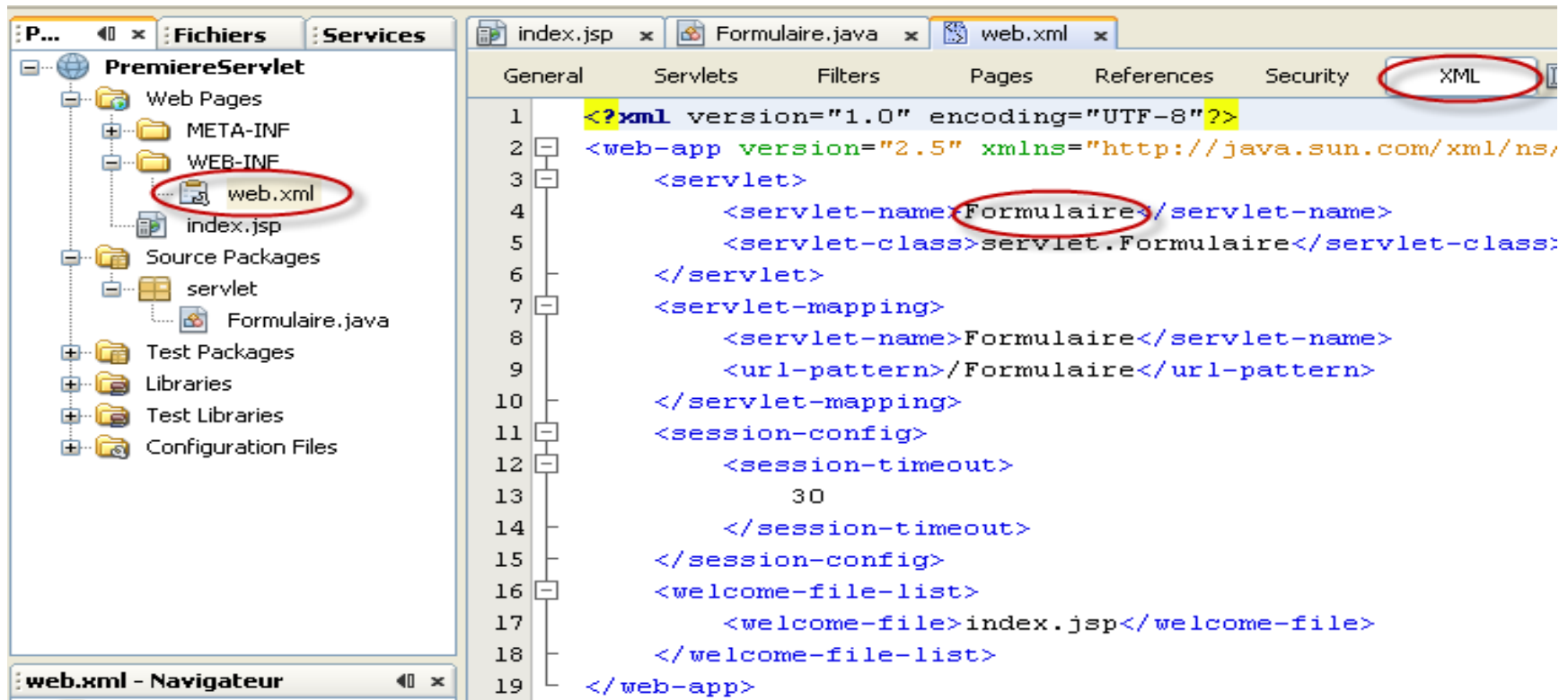
Mise en œuvre d'une servlet

Enregistrement de vos coordonnées effectué

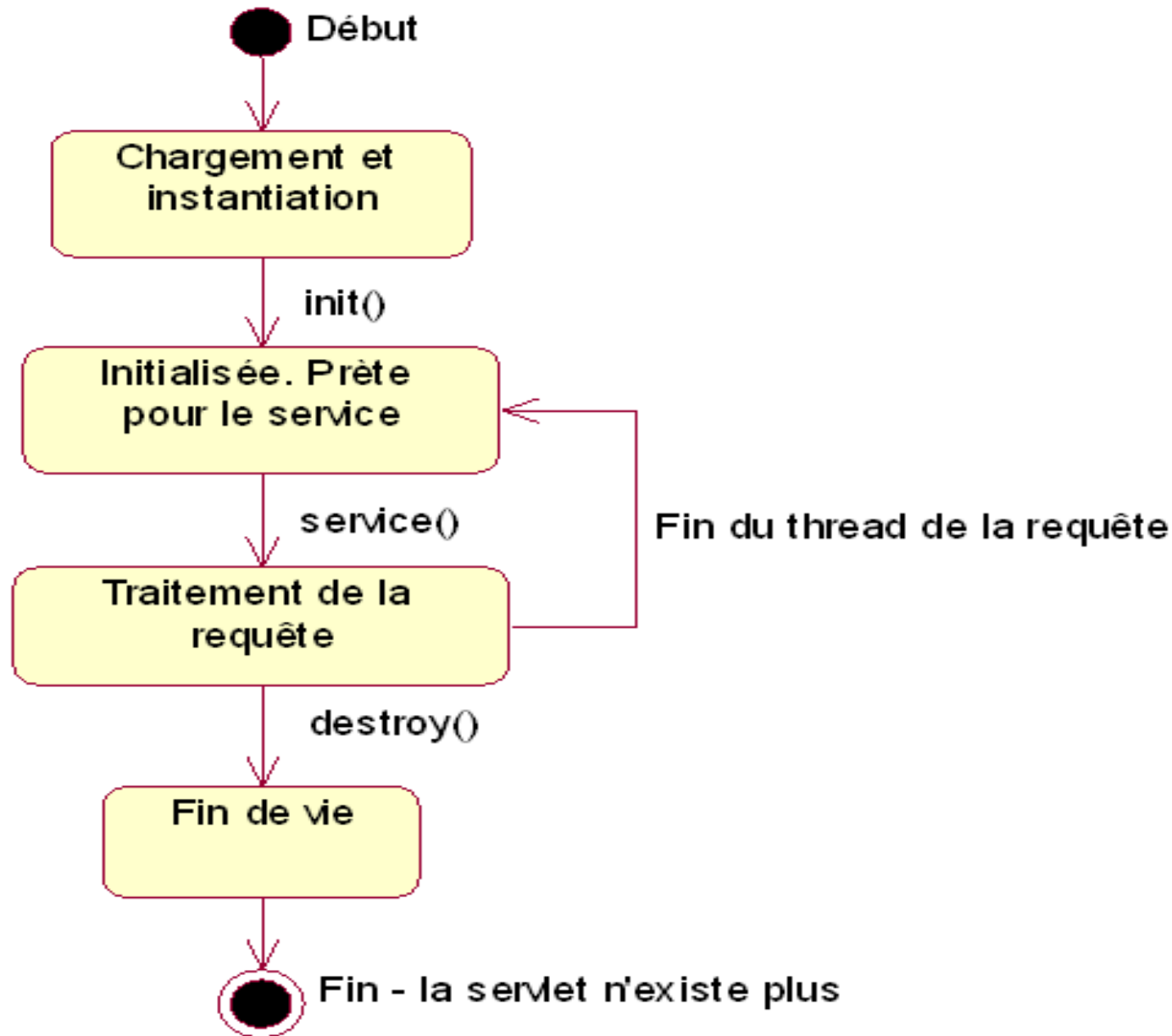
Bonjour Monsieur Youssef BERAICH.

Vous êtes un adulte.

Résultat



Cycle de vie de la servlet



Chargement et instanciation

- Comment le conteneur de servlets sait-il quelle servlet charger ?
- Tout simplement en lisant le descripteur de déploiement dont il connaît l'emplacement. Chaque application Web possède son propre répertoire à l'intérieur duquel se situe le sous-répertoire **<WEB-INF>**.
- Ce sous-répertoire constitue la partie privée de l'application Web qui n'est donc accessible que par le conteneur de servlets.
- **<WEB-INF>** contient le fichier **<web.xml>** qui n'est autre que le descripteur de déploiement. Le conteneur de servlets lit ce fichier et charge les classes des servlets identifiées, puis, il fabrique l'objet de chaque servlet en appelant son constructeur par défaut.

Chargement et instantiation

```
web.xml  Formulaire.html  Formulaire.java

<?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>Gestion du Personnel</display-name>
  <welcome-file-list>
    <welcome-file>Formulaire.html</welcome-file>
  </welcome-file-list>
  <servlet>
    <display-name>Enregistrement du Personnel</display-name>
    <servlet-name>FormulairePersonne</servlet-name>
    <servlet-class>Formulaire</servlet-class>
    <init-param>
      <param-name>jdbc.Driver</param-name>
      <param-value>com.mysql.jdbc.Driver</param-value>
    </init-param>
    <init-param>
      <param-name>localisation</param-name>
      <param-value>jdbc:mysql://localhost/gestion</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>FormulairePersonne</servlet-name>
    <url-pattern>/formulaire</url-pattern>
  </servlet-mapping>
</web-app>
```

Initialisation

- Vu que nous ne redéfinissons pas le constructeur par défaut, il faut tout de même prévoir l'initialisation de la servlet lorsque cette dernière a été chargée et instanciée. Cette phase d'initialisation a lieu lorsque le conteneur appelle la méthode `init(ServletConfig)`.
- Si votre servlet n'a aucune initialisation particulière à effectuer, il n'est pas nécessaire d'implémenter cette méthode.
- Cette méthode permet à la servlet de lire les paramètres d'initialisation ou les données de configuration, d'initialiser des ressources externes telles des connexions à une base de données ou d'effectuer diverses autres tâches qui seront accomplies une seule fois juste après la création de la servlet.

`public void init() throws ServletException`

`public void init(ServletConfig) throws ServletException`

Récupération des paramètres

- Pour récupérer ces paramètres, il suffit d'utiliser la méthode :
getInitParameter(String).
- L'intérêt des paramètres d'initialisation est de permettre de changer de configuration sans avoir besoin de recompiler la servlet..
- `<init-param>` est composé de deux sous-éléments qui correspondent respectivement au nom du paramètre suivi de sa valeur :
 - `<param-name>` : nom du paramètre
 - `<param-value>` : valeur du paramètre

Application

Enregistrement de vos coordonnées

Civilité :

Nom :

Prénom :

Age :

Choisissez un Continent :

```
<hr />
```

```
<form name="Globe" action="ListePays"  
      method="POST">
```

Choisissez un Continent :

```
<select name="Continent">
```

```
  <option >Afrique</option>
```

```
  <option >Amerique</option>
```

```
  <option> Asie</option>
```

```
  <option >Australie</option>
```

```
  <option >Europe</option>
```

```
</select>
```

```
<input type="submit" value="Rechercher"  
      name="rechercher" />
```

```
</form>
```


Servlet : ListePays.java

ListePays -> /ListePays

Servlet Name:

ListePays

Startup Order:

Description:

☒ Servlet Class:

servlet.ListePays

Browse...

[Go to Source](#)

☐ JSP File:

Browse...

[Go to Source](#)

URL Pattern(s):

/ListePays

Use comma (,) to separate multiple patterns.

Initialization Parameters:

Parameter Name	Parameter Value	Description
JDBC Driver	sun.jdbc.odbc.JdbcOdbcDriver	
Localisation	jdbc:odbc:Globe	

Ajouter...

Éditer...

Supprimer

Servlet : ListePays.java

```
public class ListePays extends HttpServlet {  
    private Connection Connexion = null;    private Statement  
    Instruction = null;  
    ResultSet RS = null;    private String Requete = null;  
  
    public void init() throws ServletException{  
        String Pilote = getInitParameter("JDBC Driver");  
        String BD = getInitParameter("Localisation");  
        try{  
            Class.forName(Pilote);  
            Connexion =  
            DriverManager.getConnection(BD, "root", "");  
        }catch(ClassNotFoundException cnfe){  
            log("Driver BD: "+ Pilote+" non trouvé !!"); throw new  
            ServletException();  
        }catch(SQLException sqle){  
            log("BD: "+BD+" non trouvée !!");  
        }  
    }  
}
```

Servlet : ListePays.java

```
String ContinentRech = request.getParameter("Continent") ;
try {
    Requete = "SELECT * FROM Pays WHERE
ContinentPays='"+ContinentRech+"'";
    Instruction = Connexion.createStatement();
    RS = Instruction.executeQuery(Requete);
    out.println("<html>"); out.println("<head>");
    out.println("<title>Servlet ListePays</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Liste des Pays Dans le Continent :
"+ContinentRech+"</h1>");

    ResultSetMetaData RSMD = RS.getMetaData();
    int NbColonne = RSMD.getColumnCount();

    out.println("<Table border = 2>");
    for(int i=1;i<=NbColonne;i++){
        out.println("<th>" + RSMD.getColumnLabel(i) + "</th>");
    }
}
```

Servlet : ListePays.java

```
while(RS.next()){
    out.println("<tr>");
    for(int i=1;i<=NbColonne;i++)
        out.println("<td align=center>" + RS.getString(i)
+ "</td>");
    out.println("</tr>");
}
    out.println("</table>");

}catch(Exception e){
    e.printStackTrace();
}
finally {
    out.close();
}
}
```

Servlet : ListePays.java

- }catch(Exception e){
- e.printStackTrace();
- }
- finally {
- out.close();
- }
- }

Choisissez un Continent : Europe  Rechercher



Liste des Pays Dans le Continent : Europe

NuméroPays	NomPays	CapitalePays	PopulationPays	ContinentPays
5	France	Paris	0	Europe
6	Espagne	Madride	0	Europe
7	Portugale	Lisbone	0	Europe
8	Belgique	Bruxelles	0	Europe
9	Suede	Oslo	0	Europe
10	Roumanie	Bukharest	0	Europe
11	Italie	Rome	0	Europe

Les JSP

Java Server Page

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.
- 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.

Exemple de JSP

```
<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") %>
  <%= 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>
</p>
</body>
</html>
```

Expressions

Scriptlets

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.**

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 XML `<jsp:directive.include file = "... " />`
 3. **taglib** : (étudié au prochain chapitre).

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 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 pouvons utiliser une des syntaxes suivantes :

<%! Vector v = new Vector() ; %>

Ou

**<jsp:declaration>Vector v=new Vector()
;</jsp:declaration>**

Les déclarations

<%!

```
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 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 !
```

```
<% } %>
```

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 dans le navigateur le texte :

Le nombre d'éléments dans cette phrase est 10

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>

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.

--%>









- 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 recenser 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.

Base de données

MySQL Table Editor

Table Name: Database: Comment:









Columns and Indices Table Options Advanced Options

Column Name	Datatype	NOT NULL	AUTO INC	Flags	Default Value	Comment
 idPersonne	 INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL		
 nom	 VARCHAR(15)	<input checked="" type="checkbox"/>		<input type="checkbox"/> BINARY		
 prénom	 VARCHAR(15)	<input checked="" type="checkbox"/>		<input type="checkbox"/> BINARY		
 motDePasse	 VARCHAR(15)	<input checked="" type="checkbox"/>		<input type="checkbox"/> BINARY		

MySQL Table Editor

Table Name: Database: Comment:

Columns and Indices Table Options Advanced Options

Column Name	Datatype	NOT NULL	AUTO INC	Flags	Default Value	Comment
 idMessage	 INTEGER	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL		
 idPersonne	 INTEGER	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/> UNSIGNED <input type="checkbox"/> ZEROFILL	0	
 sujet	 VARCHAR(25)	<input checked="" type="checkbox"/>		<input type="checkbox"/> BINARY		
 texte	 TEXT	<input checked="" type="checkbox"/>				

descripteur de déploiement 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>
```

ConnexionBD.java : Déclaration

```
package bd;  
import java.sql.*;  
public class ConnexionBD {  
    private Connection connexion;  
    private Statement instruction;  
    protected ResultSet résultat;
```

ConnexionBD.java : Constructeur

```
public ConnexionBD() {  
    try {  
        Class.forName("com.mysql.jdbc.Driver");  
        connexion =  
            DriverManager.getConnection("jdbc:mysql://localhost/messagerie", "root", "");  
        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"); } }  
}
```

• ConnexionBD.java

```
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); } }
```

ConnexionBD.java

```
public boolean suivant() {  
    try {  
        return resultat.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");  
    } } }
```


ListeMessages.java

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 resultat.getString("sujet");
        } catch (SQLException ex) {
            return "";
        }
    }

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

```

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" %>

```

```
<!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>
```

```
<%@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>
```

```
<%@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" %>
```

Les actions

- Le dernier groupe d'éléments (des JSP) est celui des **actions**, également appelées **actions standards**.
- Les actions standards sont **définies** par la **spécification JSP**. (C'est pour cette raison qu'elles sont appelées standards).
- Nous verrons qu'il est possible de **définir de nouvelles actions** et les utiliser dans nos pages JSP.

Les actions

- **<jsp:useBean>**
- **<jsp:setProperty>**
- **<jsp:getProperty>**
- **<jsp:param>**
- **<jsp:include>**
- **<jsp:forward>**
- **<jsp:plugin>, <jsp:params>, <jsp:fallback>**
- **<jsp:attribute>**
- **<jsp:body>**
- **<jsp:invoke>**
- **<jsp:dobody>**

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, 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 les pages Web dynamiques, alors que le **développeur** s'occuperait essentiellement de mettre en place **toutes les classes** nécessaires à l'application Web.
- Les **JavaBeans** permettent de composer une structure particulière sur ces classes respectant un **canevas standard** afin qu'ils puissent être utilisés 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 (pas un Entreprise JavaBean) est une classe Java respectant un certain nombre de conventions. Les deux plus importantes 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()**.

L'action <jsp:useBean>

- 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>`

- L'action `<jsp:useBean>` prend les 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 `instantiate()` 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.

L'action `<jsp:useBean>`

- 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 combinaisons 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.

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.

Exemple de JavaBean

- 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; }  
}
```

Exemple d'utilisation de <jsp:setProperty>

- 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 = "BERAICH"
```

```
/>
```

```
<jsp:setProperty  
    name = "utilisateur"  
    property = "prénom" value =  
        "<%= Request.getParameter("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.
 3. 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.

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 BERAICH et pour prénom Youssef.

Application : Action JSP et JavaBean

Voici ci-dessous la page d'inscription

<**nouvelutilisateur.jsp**> :

Messages

[Sujets](#)

[Identification](#) [Inscription](#)

Demande d'inscription

Nom	<input type="text" value="beraich"/>
Prénom	<input type="text" value="youssef"/>
Mot de passe	<input type="password" value="••••"/>

dimanche 25 novembre 2012

Application : Action JSP et JavaBean

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 :

Messages

[Sujets](#)

[Identification](#) [Inscription](#)

Confirmation de votre demande d'inscription

Nom	BERAICH
Prénom	Youssef
Mot de passe	1234

Nouvel utilisateur enregistré

dimanche 25 novembre 2012

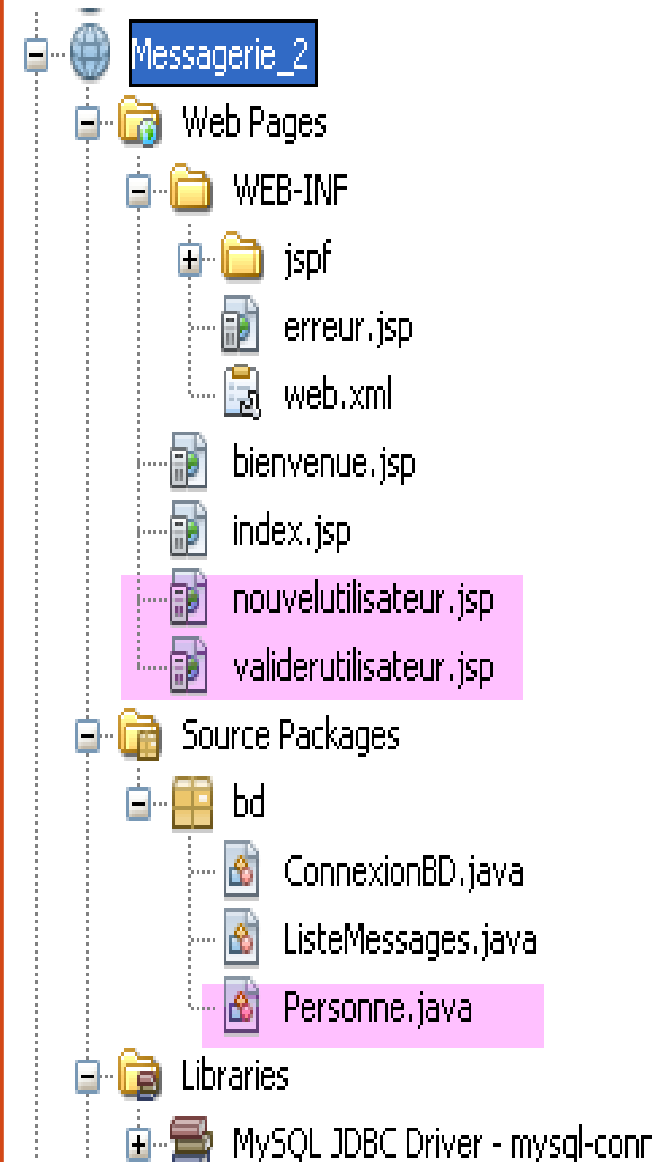
Application : Action JSP et JavaBean

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 :

Messages	
Sujets	Identification Inscription
Confirmation de votre demande d'inscription	
Nom	BERAICH
Prénom	Youssef
Mot de passe	1234
ATTENTION : Utilisateur déjà enregistré	
dimanche 25 novembre 2012	

Application : Action JSP et JavaBean

- Pour réaliser l'ensemble de l'inscription, nous avons besoin de développer trois nouveaux composants Web :
- 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.
- La page JSP **<nouvelutilisateur.jsp>** qui s'occupe du formulaire de saisie.
- 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.



Application : **JavaBean** **Personne**

```
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; }  
}
```

Application : **JavaBean** **Personne**

```
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();  
}  
}
```

Application : La page <nouvelutilisateur.jsp>

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" %>
```



Mêmes noms que dans le bean Personne

Application : La page <validerutilisateur.jsp>

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 déjà 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>
```

Tous les attributs


```

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 déjà 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>

```

Tous les attributs

Action JSP : Les objets implicites

- Une page JSP peut accéder directement à la requête, par l'intermédiaire 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 auquel on peut 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 :

request	config
response	exception
out	application
session	

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)**
;

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("préno  
m") + "</h3>") ; %>
```
- Dans cet exemple, l'utilisation de l'objet implicite **out** ne procure pas un grand intérêt.

L'objet **session** (1)

- 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.
- La page peut stocker des informations à propos du client. 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) ;

L'objet **session** (2)

- Vous pouvez placer des **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**.
- 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) ;

La portée des objets implicites

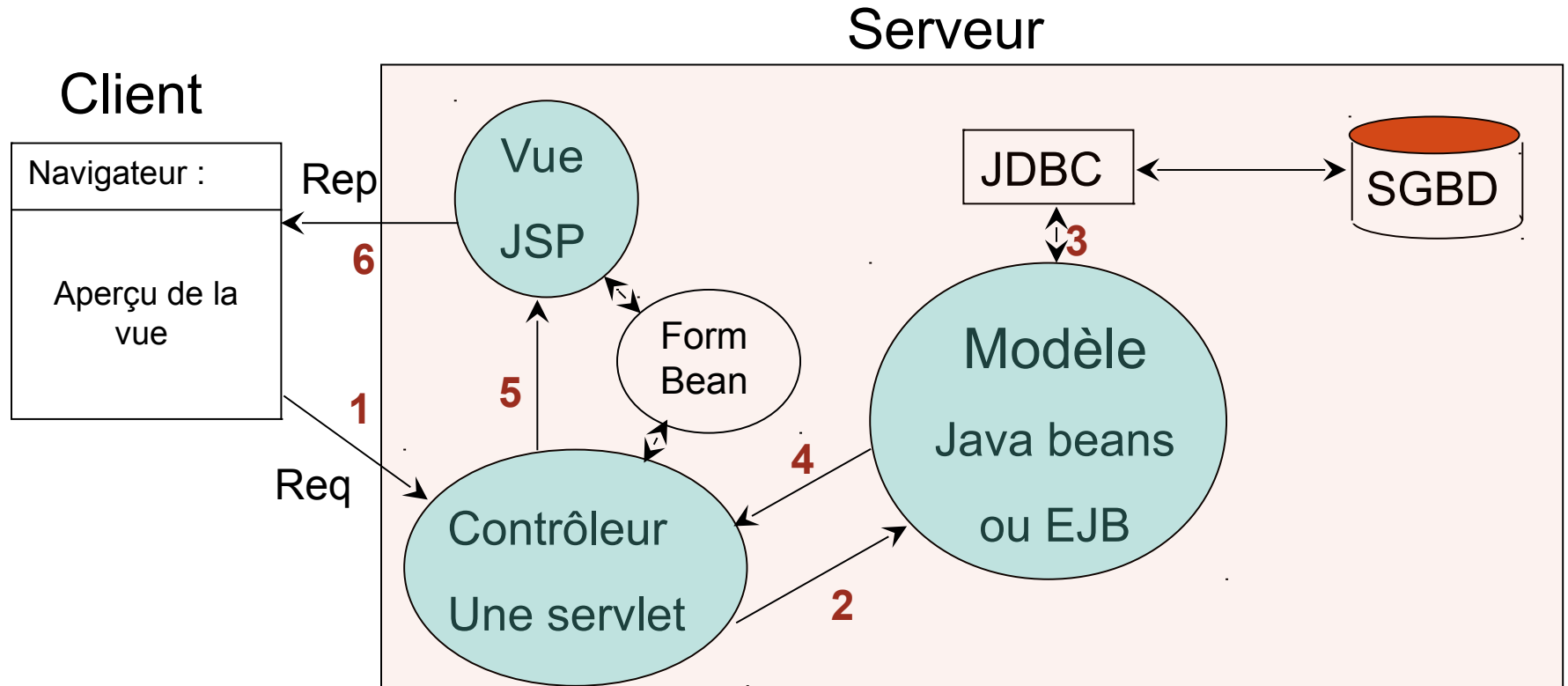
- Les objets créés dans les pages JSP ont une certaine portée, qui correspond en quelque sorte à leur durée de vie. Dans certains cas, cette portée est déterminée et ne peut être modifiée. Il en est ainsi des objets implicites. Pour d'autres objets (par exemple les JavaBeans), le développeur peut choisir la portée. Les portées valides sont page, request, session, application.
- **page** - C'est la portée la plus réduite. Les objets ayant cette portée ne sont accessibles que dans la page qui les définit.
- **request** - 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.
- **session** - Les objets ayant cette portée sont accessibles pour tous les composants participant à la session.
- **application** - Les objets concernés sont accessibles à toute l'application Web, pendant toute sa durée de vie.

Modèle MVC

MVC

- Dans la pratique, on cherche toujours à séparer la logique du métier et la logique de présentation.
- En optant pour le modèle MVC (Modèle, Vue, Contrôleur), Java met à votre disposition des différents composants qui vous permettent de répondre à ce critère.

Modèle MVC pour une application web java



MVC pour une application web

- Pour une application web, le modèle MVC est implémenté comme suit :
- **1- Modèle:**
 - Le modèle qui représente la partie la plus importante d'une application, se charge d'implémenter la logique du métier.
 - Avant son implémentation, le modèle fait l'objet d'une conception approfondie en élaborant les différents diagrammes UML.
 - Dans java, le modèle peut être implémenté soit en utilisant :
 - **Les java beans**
 - **Les EJB**

Modèle avec les java beans

- Les **java beans** sont déduits directement du diagramme de classes.
- Chaque java bean représente une classe du modèle.
- Cette classe est caractérisé par :
 - **des attributs** (propriétés et attributs d'associations avec les autres classes) qui sont souvent privés,
 - **les accesseurs** (get..), mutateurs (set..)
 - des **méthodes métiers** qui assurent les différents traitements du métier et la persistance de ces objets (souvent dans une base de données relationnelle).
- Les java beans doivent s'exécuter dans la même JVM que les autres parties de l'application, à savoir les contrôleurs et les vues.

Modèle avec les EJB

- L'autre solution pour le modèle consiste à utiliser les **EJB** (Entreprise Java Beans).
- Ce type de composants permet d'implémenter le modèle comme les java beans, sauf que les EJB ont la particularité d'être des composants distribués.
- Chaque EJB peut tourner dans un environnement différent de celui des autres composants de l'application.
- les EJB doivent être déployés dans un conteneur d'EJB qui est lui-même doit être géré par un serveur d'application J2EE (WebSphere, WebLogic, JBoss, Jonas ...).

Modèle avec les EJB

- Autre chose à mettre en considération, c'est que les EJB ont été conçu pour :
 - servir les applications distribuées,
 - de résoudre le problème de montée en charge,
 - de décharger le développeur d'un certain nombre de tâches offertes par les serveurs d'application (Gestion de la persistance, sécurité, authentification,...).
- En revanche, il faut également savoir que les EJB se basent sur la sérialisation, ce qui rend la vitesse d'accès à ces composant relativement lente.
- Il faut donc bien faire attention à l'utilisation des EJB et bien justifier leurs utilisations.

Contrôleur : Servlet

- Les servlets peuvent jouer le rôle du contrôleur. Ce dernier se charge de :
 - Recevoir les différentes requêtes http des utilisateurs,
 - Récupérer les données de ces requêtes,
 - Stocker ces données dans un objet intermédiaire associé à la requête et appelé form bean (bean de formulaire)
 - S'assurer de la validation de ces données.
 - Faire appel au modèle, qui se charge du traitement, en lui transmettant les données de la requête.
 - Récupérer les résultats éventuels retournés par le modèle.
 - Stocker ces résultats dans le form bean
 - Stocker le form bean dans la session de courante.
 - Faire une redirection vers une page JSP qui va se charger de l'affichage de la des résultats relatif à la réponse http.

Les Vues : JSP

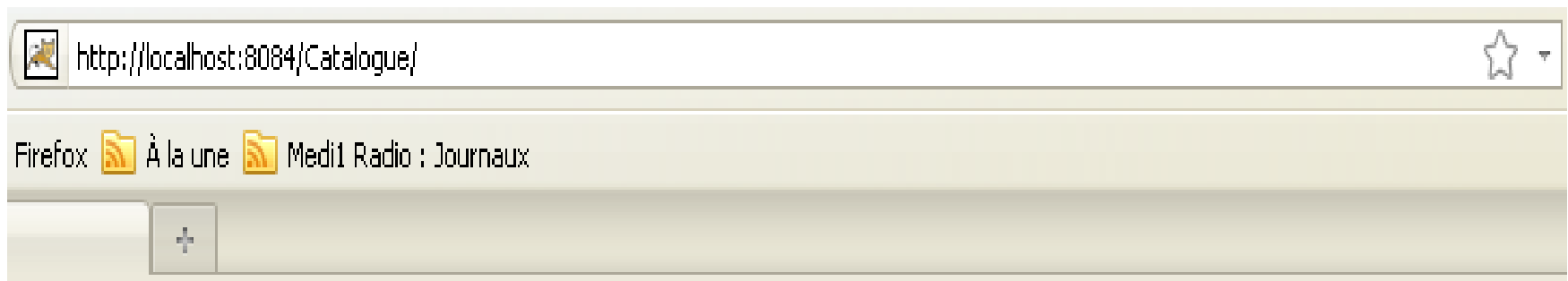
- Les vues sont implémentées par les pages JSP. Une page JSP se charge de :
 - Récupérer les données des résultats stockés, préalablement par le contrôleur, dans le form bean.
 - Afficher ces résultats dans les parties dynamiques de la page HTML (ou XML)
 - Afficher les autres parties statiques, de la vue, qui vont permettre à l'utilisateur d'envoyer d'autres requêtes vers le contrôleur via des formulaires ou des liens hypertextes ou autres.
 - Chaque vue de l'application est représentée par une page JSP. Elle peut matérialiser un ou plusieurs cas d'utilisation de l'application web.

Exemple d'application

- Supposant que l'on souhaite créer une application web qui permet de présenter le catalogue de produits d'une entreprise.
- Chaque produit appartient à une catégorie.
- Chaque catégorie est caractérisée par:
 - son identifiant,
 - le nom de la catégorie
 - et sa description.
- Un produit est caractérisé par :
 - un identifiant numérique,
 - sa désignation,
 - son prix,
 - sa quantité,
 - sa photo et une propriété selected qui indique si le produit est sélectionné ou non.

Travail demandé

- Créer une application web, respectant le modèle MVC, qui permet la saisie, l'ajout, l'affichage, la suppression et la recherche d'une catégorie en une seule vue comme le montre l'illustration suivante :

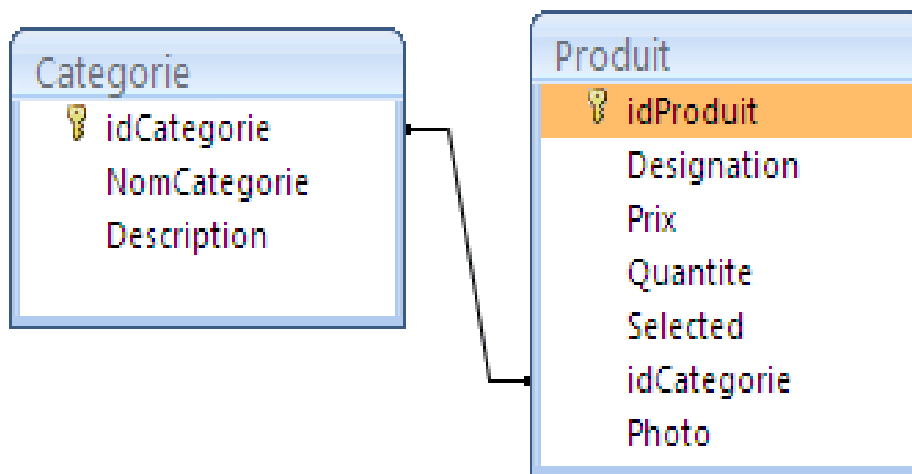
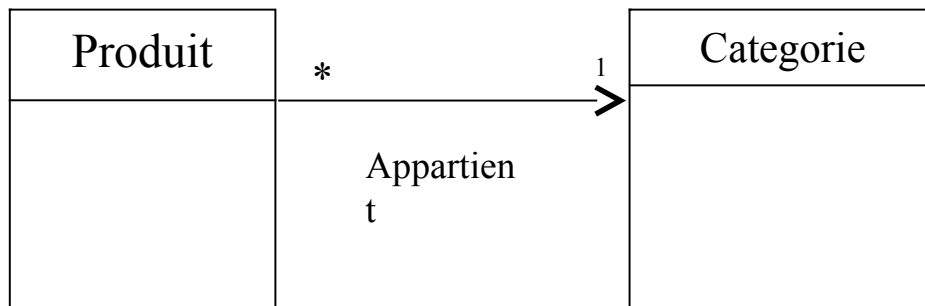


Chercher une catégorie:

Id Catégorie	Nom Catégorie	Description	
	<input type="text"/>	<input type="text"/>	<input type="button" value="Ajouter"/>
1	CPU	Microprocesseur	Supprimer
6	RAM	Mémoire Vive Pour Ordinateur de Bureau et Portable	Supprimer

Modèle : Couche métier

- Les principaux objets que va manipuler notre application sont les objets des classes **Categorie** et **Produit**.
- Ces deux classes persistantes sont liées par une association de type un à plusieurs.
- Un produit appartient à une seule catégorie et une catégorie contient plusieurs produits.



Categorie	
Nom du champ	Type de données
idCategorie	NuméroAuto
NomCategorie	Texte
Description	Texte

Produit	
Nom du champ	Type de données
idProduit	NuméroAuto
Designation	Texte
Prix	Monétaire
Quantite	Numérique
Selected	Oui/Non
idCategorie	Numérique
Photo	Texte

Modèle

- Si nous supposons qu'en connaissant un produit, on a besoin de connaître les caractéristiques de la catégorie de ce produit,
 - l'association doit être traduite dans le sens produit vers catégorie
 - ce qui signifie qu'il faut créer un attribut de type `Categorie` dans la classe `Produit`.
- Si en plus nous voudrions que si on charge une catégorie, on a besoin de connaître tous les produits de cette catégorie,
 - nous devrions traiter l'association dans le deuxième sens.
 - Ce qui va être traduit par la création, dans la classe `Categorie` d'un attribut de type collection (`Set`, `Vector`, `List`, tableau d'objets) qui permet de stocker les objets `Produit`.
- Dans notre problème nous supposant que l'association entre `Produit` et `Categorie` est unidirectionnelle dans le sens `Produit` appartient à une catégorie.

Modèle

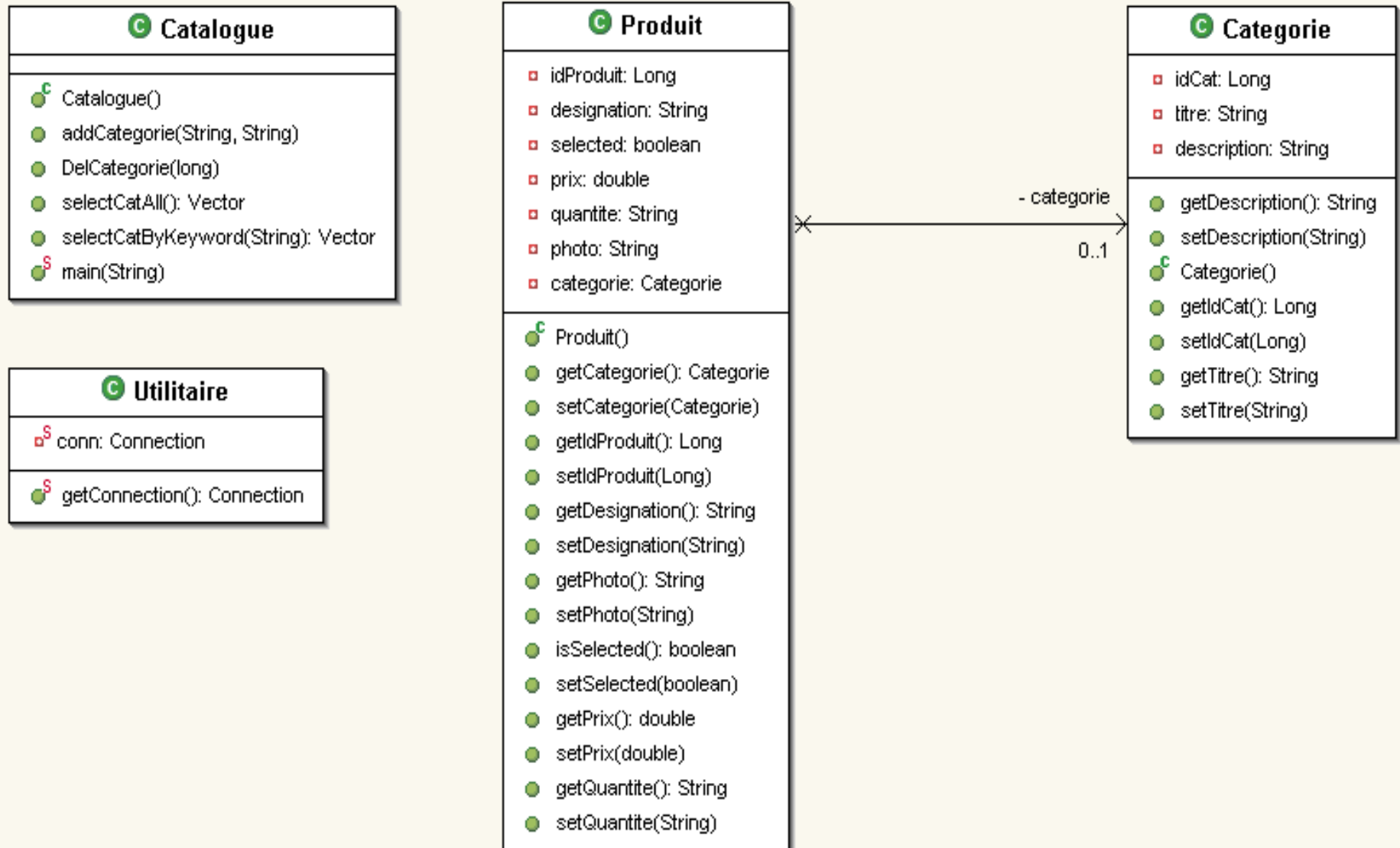
- Les deux classes Produit et Catégorie seront modélisées par de simples java beans qui seront caractérisés par :
 - des attributs privés,
 - un constructeur par défaut,
 - des getters et des setters.
- On peut ajouter dans ces classes les autres méthodes métiers qui permettent de gérer la persistance des produits et des catégories dans la base de donnée relationnelle,
- sauf qu'on préfère souvent regrouper ces différentes méthodes dans une classe à part que nous appellerons Calalogue.

Modèle

- Cette classe « Catalogue » devrait définir toutes les méthodes métiers qui permettent de répondre au besoin de l'application à savoir :
 - Une méthode qui permet d'ajouter une nouvelle catégorie
 - Une méthode qui permet de retourner toutes les catégories dans un vecteur.
 - Une méthode qui permet de retourner un vecteur catégories recherchées par mot clé.
 - Une autre méthode qui permet supprimer une catégorie.
- Dans la deuxième partie de l'application qui va manipuler les produits va nous obliger à définir d'autres méthodes qui permettent de gérer la persistance des objets de type Produit.
- En fin, pour éviter de réécrire, à chaque fois que l'on veut se connecter à la base de données, les différentes instructions qui permettent de charger le pilote JDBC et de créer une connexion, nous définissons une classe Utilitaire qui contient une méthode statique getConnection qui permet de retourner un objet Connection singleton.

Modèle: Diagramme de classes

- Notre diagramme de classe devient:



Couche Présentation

- Une fois le modèle est défini et testé en utilisant la méthode main,
- il est temps de concevoir les vues et les contrôleurs. Comme nous l'avons déjà précisé au début,
- le contrôleur est une servlet qui aura la tâche de:
 - recevoir les données des requêtes http,
 - de stocker ces données dans un objet intermédiaire appelé Form bean,
 - valider ses données,
 - faire appel au modèle pour faire le traitement,
 - stocker les résultats retournés par le modèle dans le même form bean,
 - enregistrer cet objet dans la session courante
 - et puis faire appel à la page JSP pour afficher les résultats qui se trouvent dans le form bean.

Le form bean

- **idCat** : pour stocker la valeur du paramètre idCat envoyé avec la requête si l'utilisateur clique sur le lien supprime, pour supprimer une catégorie
- **motCle** : pour stocker la valeur du paramètre motCle envoyé avec la requête lorsque l'utilisateur recherche des catégories en saisissant le mot clé.
- **nomCat** et description serviront pour stocker les valeurs du nom de la catégorie et de sa description envoyés avec la requête en utilisant le formulaire qui permet d'ajouter une nouvelle catégorie.
- **lesCat** est un vecteur qui va servir pour stocker les catégories qui seront affichées dans la vue, en réponse à la requête.

CategorieForm

- idCat: long
- motCle: String
- nomCat: String
- description: String
- lesCat: Vector

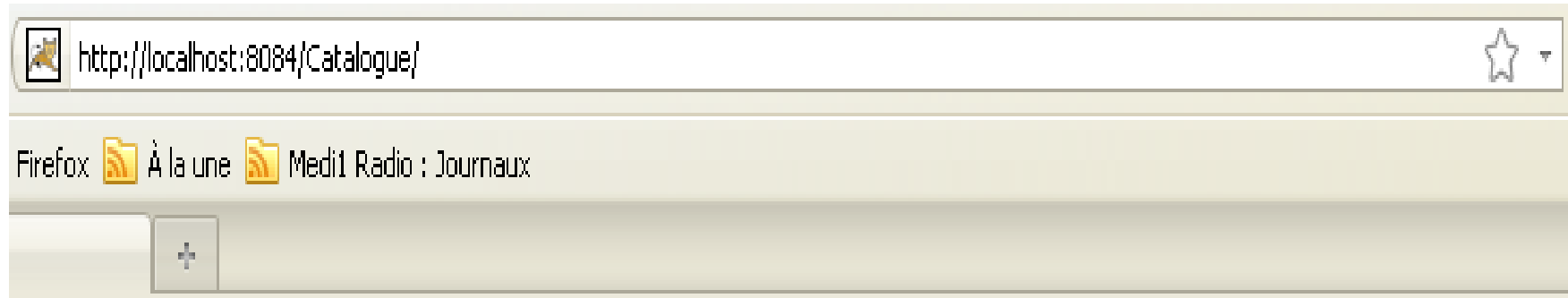
- getMotCle(): String
- setMotCle(String)
- getIdCat(): long
- setIdCat(long)
- getDescription(): String
- setDescription(String)
- getLesCat(): Vector
- setLesCat(Vector)
- getNomCat(): String
- setNomCat(String)

Contrôleur : Servlet (CatalogueServlet.java)

- **Servlet**

CatalogueServlet
<ul style="list-style-type: none">● doGet()● doPost()

Vue : une JSP (Catalogue.jsp)



Chercher une catégorie: <input type="text"/> <input type="button" value="Chercher"/>			
Id Catégorie	Nom Catégorie	Description	
	<input type="text"/>	<input type="text"/>	<input type="button" value="Ajouter"/>
1	CPU	Microprocesseur	Supprimer
6	RAM	Mémoire Vive Pour Ordinateur de Bureau et Portable	Supprimer