Servlet & pages JSP

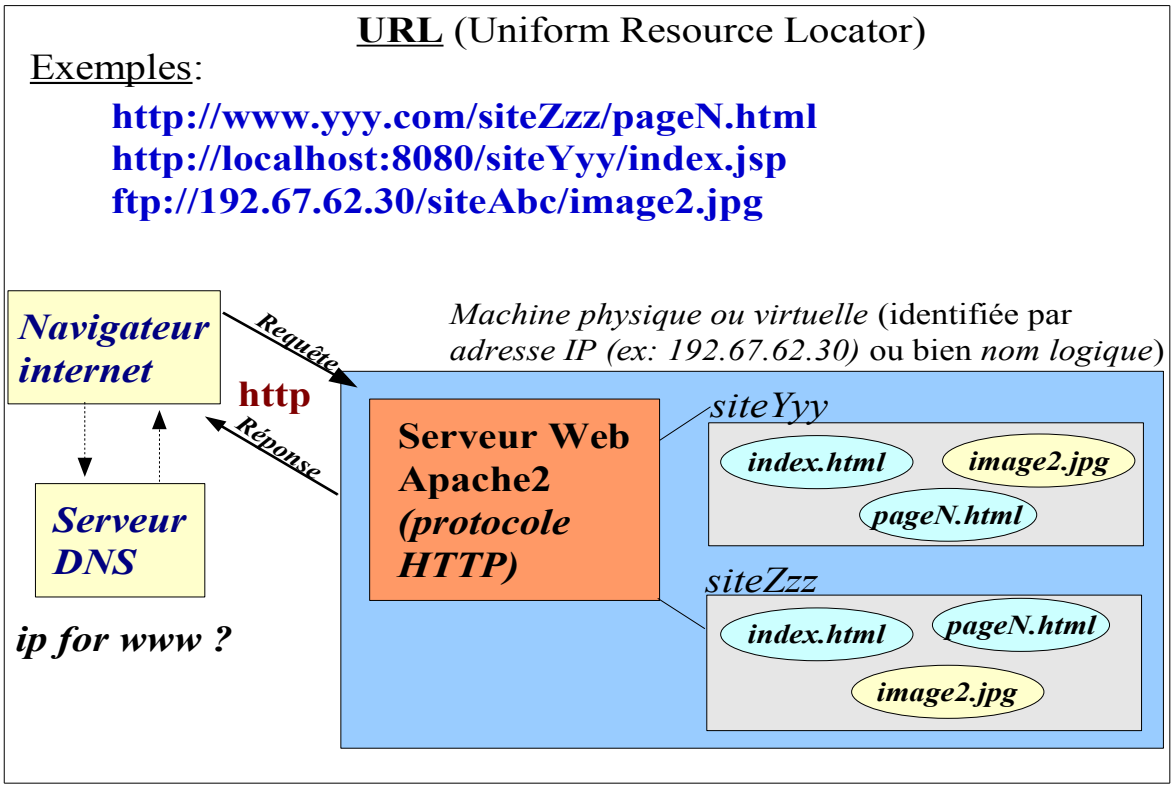
(java web-app)

Table des matières

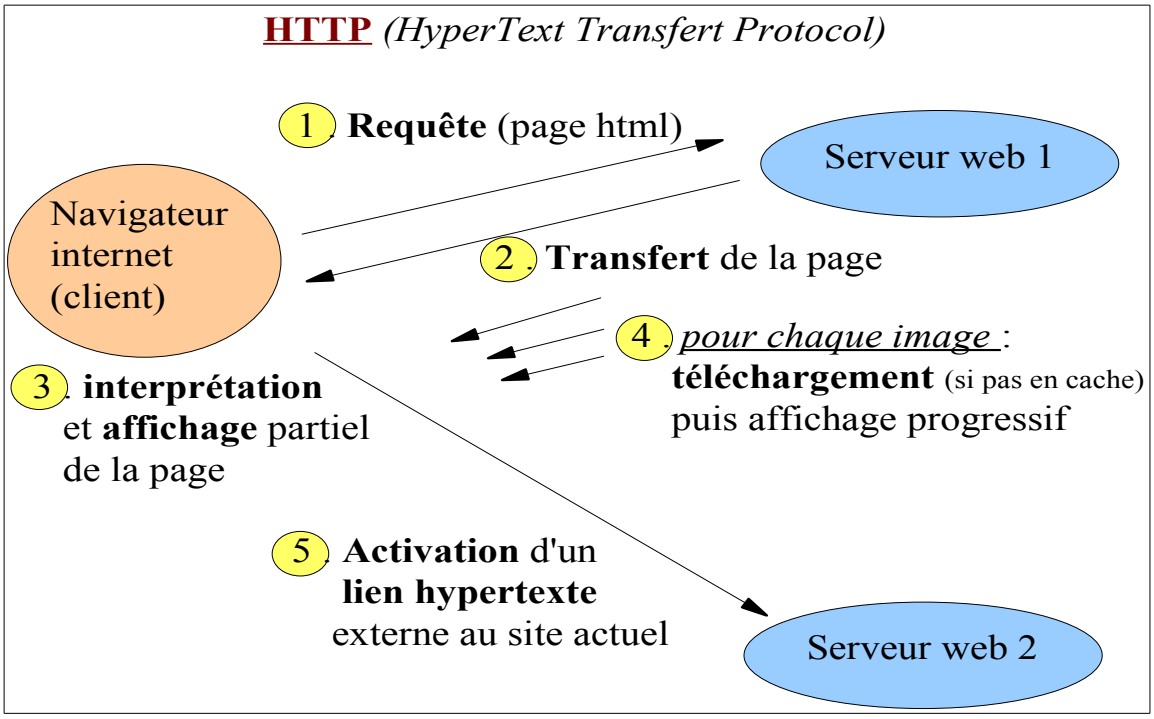
I - Application Java/Web (présentation)

1. URL et Protocole HTTP

1.1. URL (Uniform Resource Locator)



1.2. Présentation de HTTP



HTTP (HyperText Transfert Protocol)

HTTP est un protocole sans état

(*pas de session longue* ,
connexion ,
requêtes/réponses ,
déconnexion *(c'est tout)*)

Requête http

```
GET /page2.htm HTTP/1.0
If-Modified-Since: Monday, 29-Jan-96 15:56:20 GMT
User-Agent: .....
Accept: image/gif, image/jpeg, */*
[CRLF]
```

Les types **MIME** (Multipurpose Internet Mail Extension) permettent d'identifier les formats des données véhiculées :

Type MIME	*extension*	*Contenu / interprétation*
text/html	.html , .htm	page html
image/png	.png	image "png"
image/jpeg	.jpg , .jpeg	image "jpeg"
...	.zip	archive à télécharger

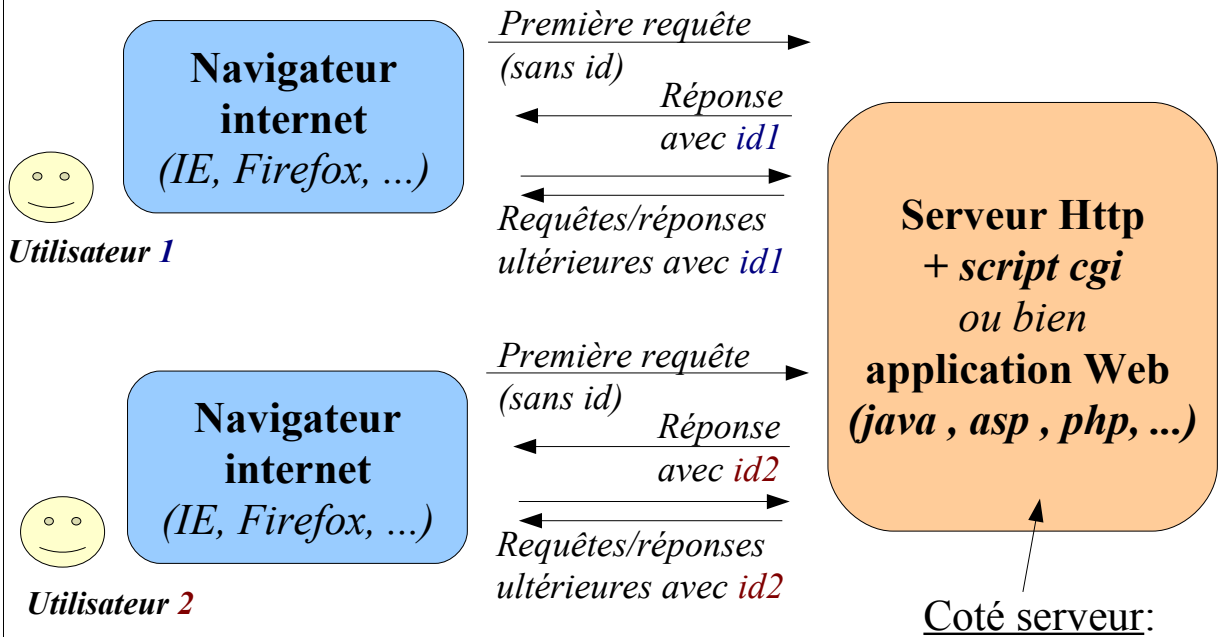
Réponse http

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 226
[CRLF]
<HTML> <HEAD> ... </HEAD> <BODY> ... </BODY> </HTML>
```

HTTP ***(protocole de + haut niveau)*** → *Gère des transferts de fichiers avec des liens hypertextes*

TCP/IP ***(protocoles réseaux de bas niveau)*** → *Gère les communications réseau à travers la toile internet (www=world wide web)*

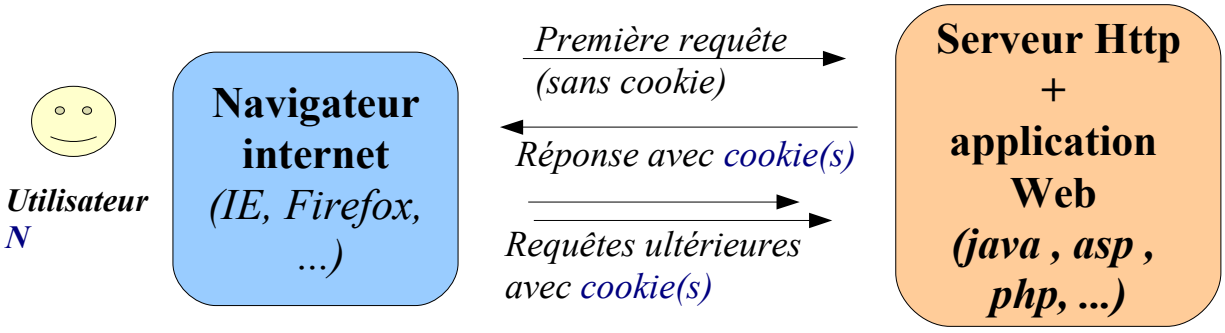
HTTP (lien à moyen/long terme avec l'utilisateur)



Techniques pour véhiculer les "id" des sessions:
* ***cookies http***
* ***infos en fin d'URL*** *ou* ***champs cachés***

Génération de nouveaux id et ***associations***
(id_1 , données_session_utilisateur_1)
(id_2 , données_session_utilisateur_2)

Cookies HTTP *(id session ou préférences de l'utilisateur)*



Un ***cookie Http*** est une ***information*** de type
(nom, valeur, éventuelle_date_expiration, url_site_web)
qui est :
* ***générée dynamiquement coté serveur (selon code ou ...)***
* ***automatiquement stockée coté navigateur en mémoire et dans un fichier si une date d'expiration a été précisée.***
* ***systématiquement/automatiquement renvoyée au serveur à chaque émission d'une nouvelle requête http vers le site web impacté***

Exemples:
***sport_préféré**=football,expires=2010,...*
***jSessionId**=A2B3C567C45677B3445A445*

HTTP *(modes "GET" et "POST")*

Page html dans navigateur

```
<html> <head>...</head>
<body> ...
  <form action="s1"
    method="GET"
          ou"POST" >
    Prenom : alain
    Nom:     Therieur
    Age :    40
  </form>
</body>
</html>
```

Requête HTTP avec valeurs saisies → **Serveur Http + application Web *(java , asp , php, ...)***

prenom=*alain*&nom=*therieur*&age=*40*

en fin d'URL (après ?) en mode "GET" :
(ex: "http://.../s1?prenom=alain&...=...")

ou bien

Dans la partie "corps interne" de la requête HTTP en mode "POST"

Mode "GET" ---> dans historique (peu confidentiel) , utilisable dans lien hypertexte
Mode "POST" --> pas de limite de taille dans les données saisies

2. Pages HTML

HTML *(HyperText Markup Language)*

Page html interprétée (pour affichage) dans un navigateur internet

```
<html>
  <head>
    <title>titrePage</title> ...
  </head>
  <body>
     <b>texte_en_gras (bold)</b>
     <i>texte_en_italique</i>
     <u>texte souligné <underline)</u>
     <img src='image1.jpg' />
     <a href='page2.html'> vers page 2 </a>
  </body>
</html>
```

Entête invisible

Partie visible de la page

Mise en forme du texte encadré (balisé)

Insertion d'image

Lien hypertexte vers une autre page

HTML *(formulaires et tableaux)*

Page html dans navigateur

```
<html> <head>...</head>
 <body> ...
  <form action="s1"
    method="GET" ou"POST" >
     couleur : rouge
     Nom: Therieur
     soumettre
  </form>
  <table border='2' >
      <tr><th>années</th><th>valeurs</th></tr>
      <tr><td>2008</td><td>100</td></tr>
      <tr><td>2009</td><td>120</td></tr>
  </table>
 </body>
</html>
```

Formulaire de saisie

Liste déroulante

```
<select name="couleur">
    <option>rouge</option>
    <option>bleu</option>
</select>
```

Zone de saisie

```
<input type=text" .../>
```

Bouton poussoir

```
<input type="submit" .../>
```

tableau

années	*valeurs*
2008	100
2009	120

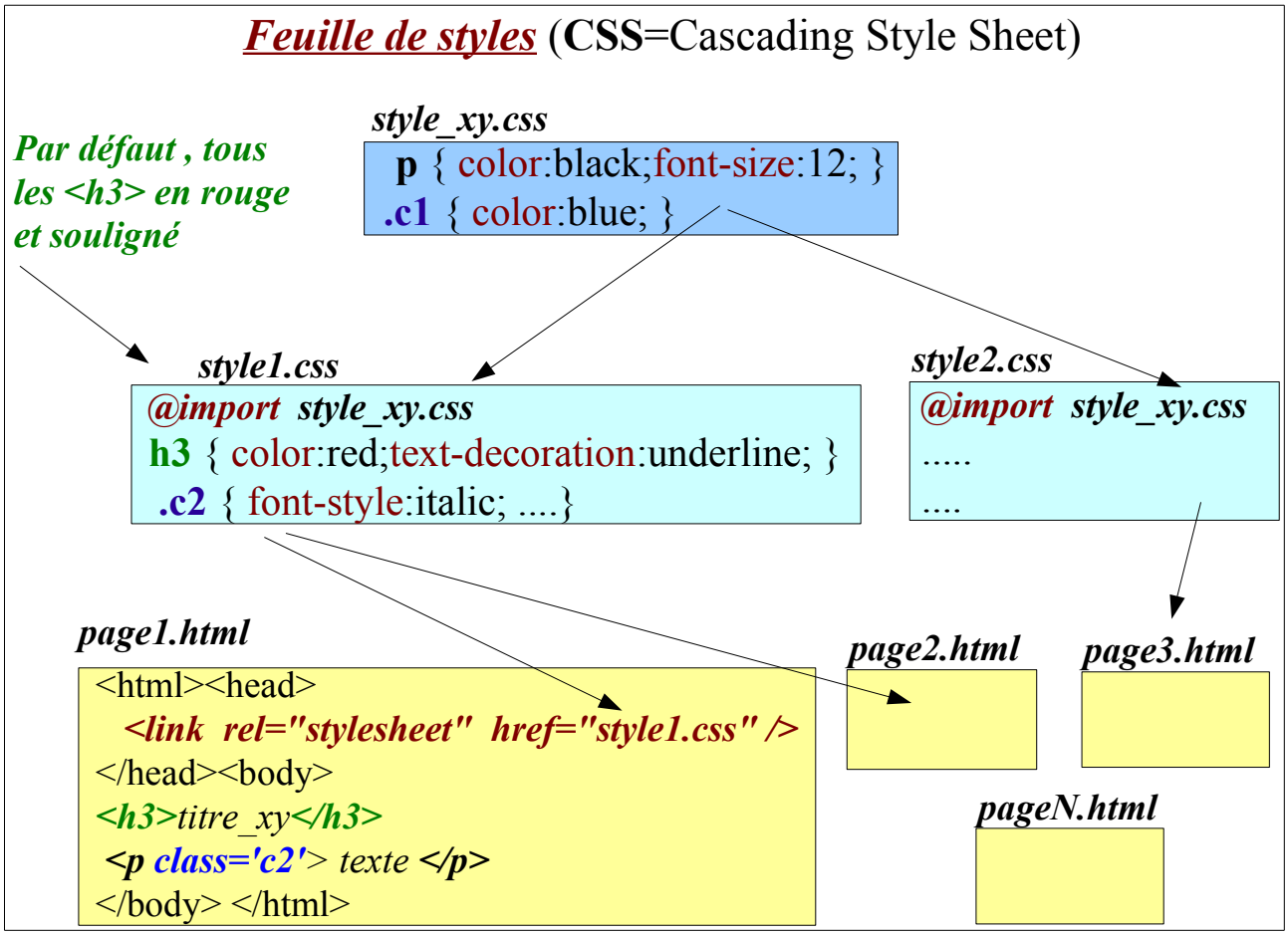
3. Feuilles de styles (CSS, ...)

3.1. Intérêts des feuilles de styles

- Clairement **découpler (séparer)** le **contenu** de la **mise en forme** .
- Permettre différentes représentations d'un même contenu (via l'application de différents styles)
- Basculer très rapidement d'un look à l'autre (en changeant les attributs d'une feuille globale).
- Maintenance du site simplifiée , évolutivité garantie .

3.2. CSS (Cascading Style Sheet)

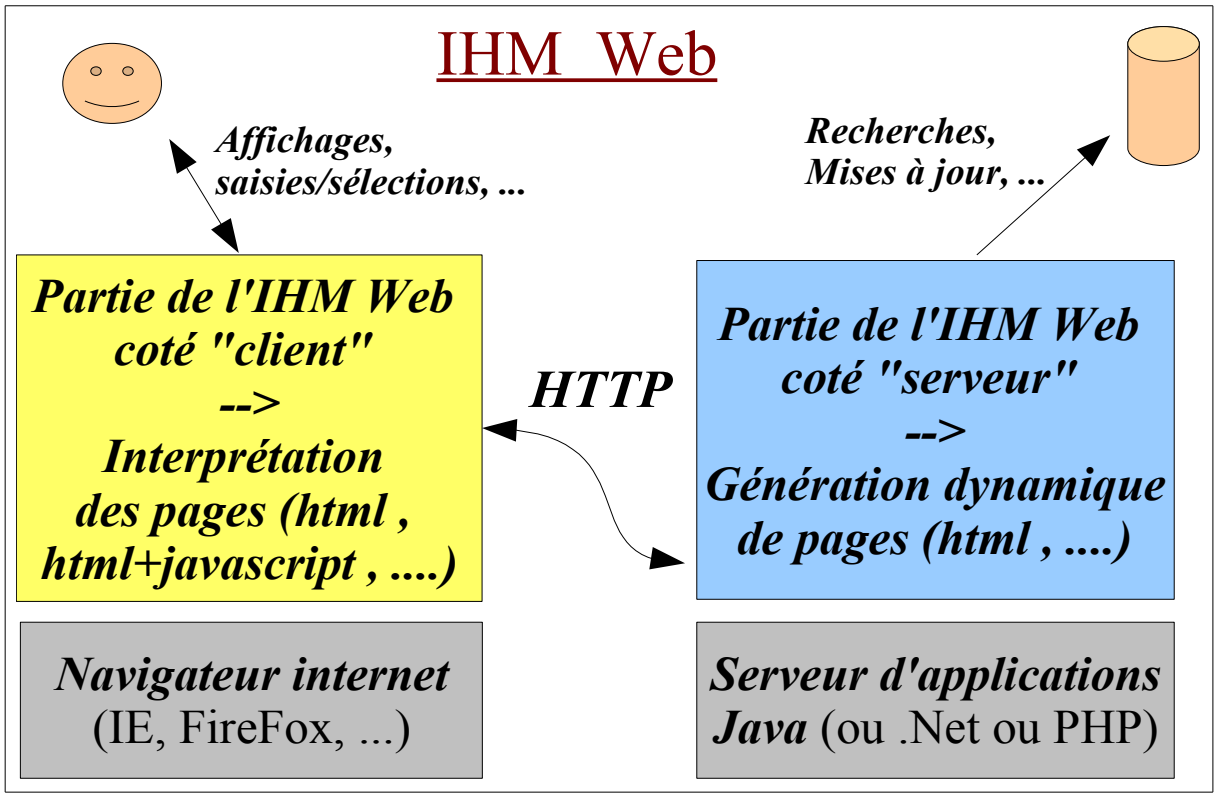
Ces **feuilles de styles** peuvent être organisées **en cascade** car on peut établir tout un tas d'inclusions entre différentes expressions complémentaires des styles :

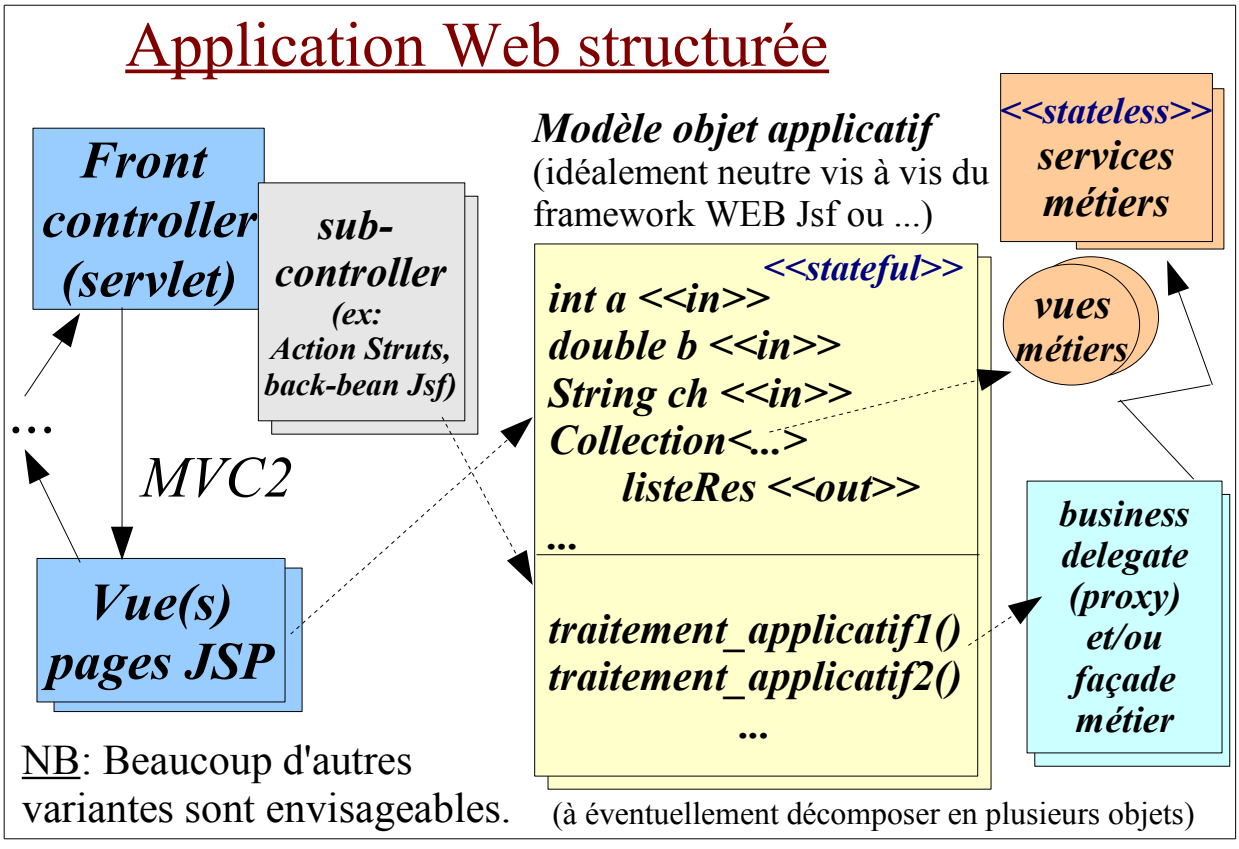


Les **styles CSS** permettent d'appliquer automatiquement (via des règles) certains **attributs de mise en forme** *(couleurs, polices , dispositions, ...)* à des éléments du document source.

La **sélection d'une règle** repose principalement sur les **noms des balises** ou sur les noms des **classes de styles** (attribut ***class*** d'une balise).

4. Tiers présentation





5. Tomcat (modes "développement" & "production")

"***Apache Software Foundation***" est un **organisme "Open Source"** dont la branche "java" est appelée "jakarta-apache" .

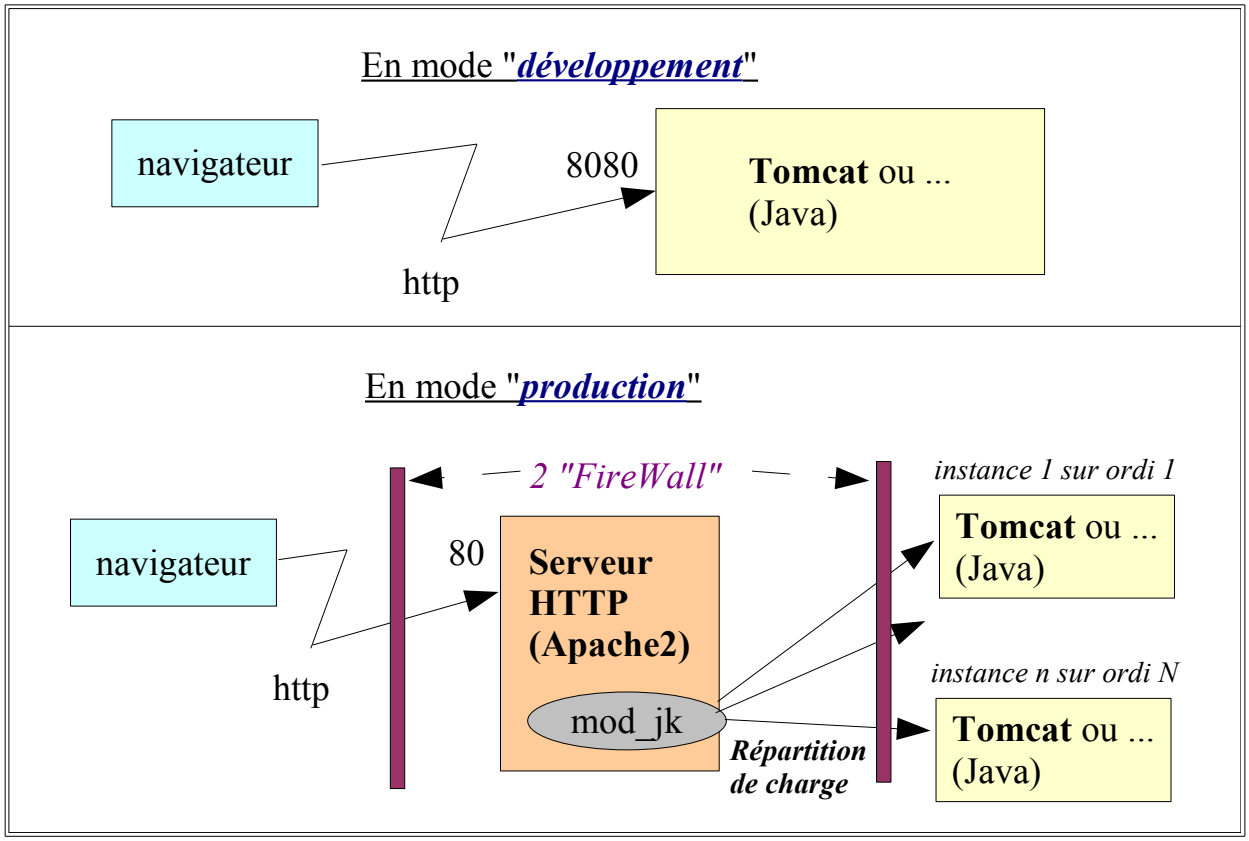
Cet organisme gère plusieurs projets (et produits logiciels) dont les principaux suivants:

- **serveur HTTP "Apache 2"** (développé en C/C++ et pas en java)
- **Geronimo** (serveur Java_EE complet avec parties "Web" et "Ejb")
- **Tomcat** qui est à la fois un serveur autonome gérant la partie "Java_Web" et un "conteneur Web" qui peut être incorporé dans d'autre serveurs JEE (tels que JBoss ou Jonas).
- ...

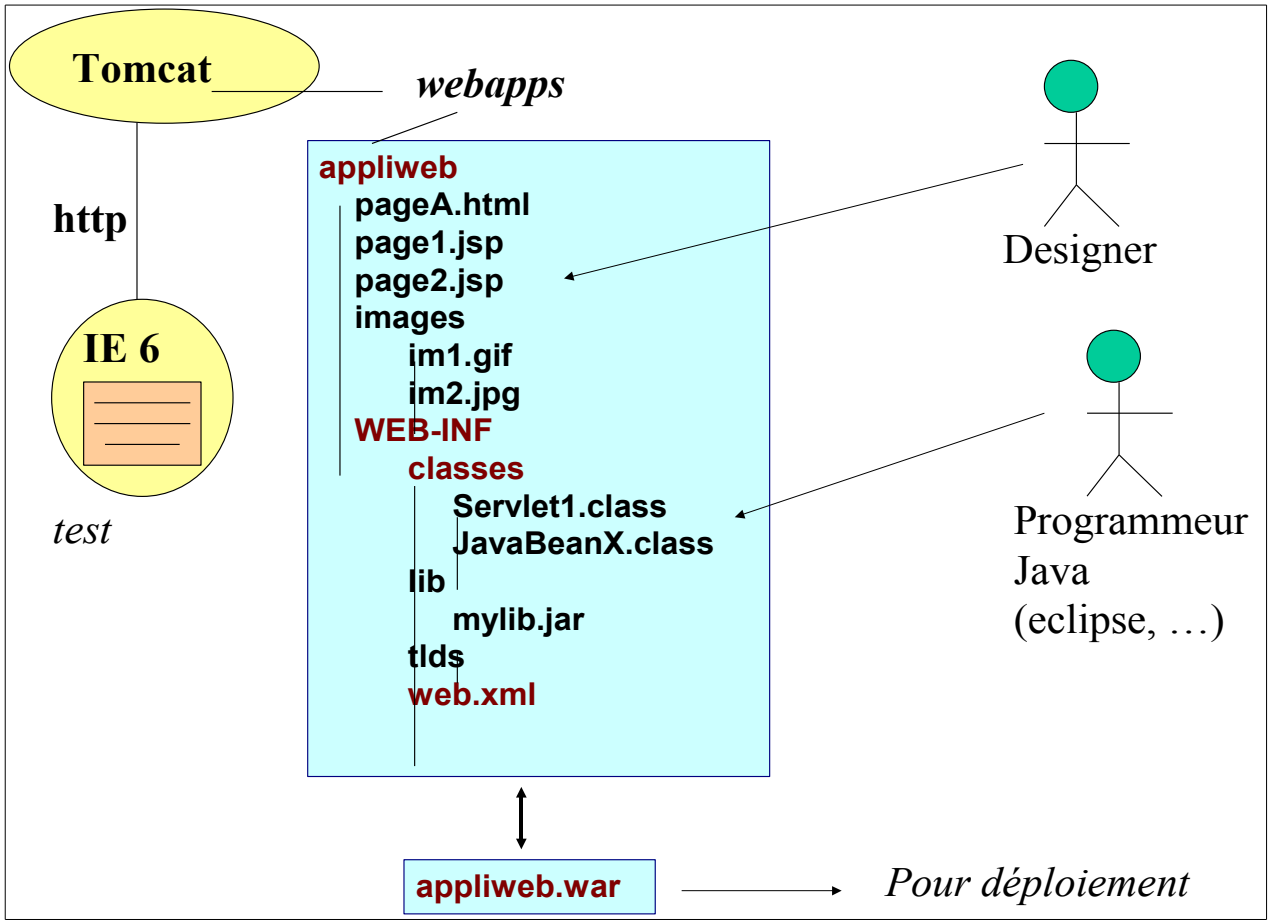
En mode développement, le navigateur internet se connecte directement au serveur Java "Tomcat" qui gère en direct le protocole HTTP (via le numéro de port 8080 par défaut).

En mode "production", le navigateur internet se connecte généralement vers un serveur HTTP (Apache2 ou équivalent) qui redirige à son tour les requêtes vers un serveur java (Tomcat ou Jboss ou WebSphere ou Weblogic ou autre).
Ceci permet:

- d'entourer le serveur Http Apache2 entre deux "firewall" qui assurent ainsi une assez bonne sécurité
- d'effectuer une éventuelle répartition de charge (en redirigeant les requêtes vers différents serveurs "Tomcat"/... fonctionnant en cluster sur différentes machines).

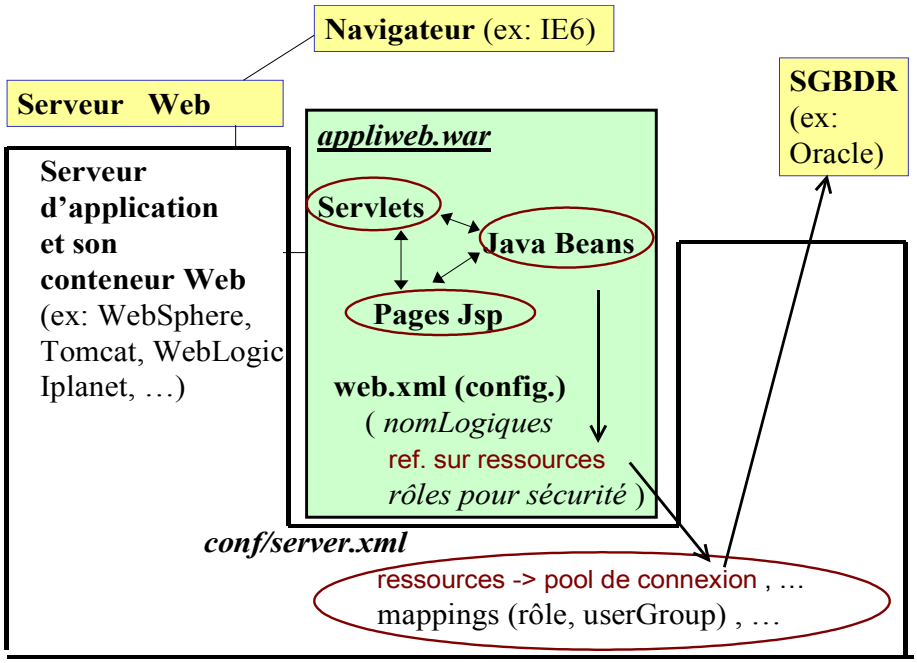


6. Structure d'une application WEB à déployer



Double configuration:

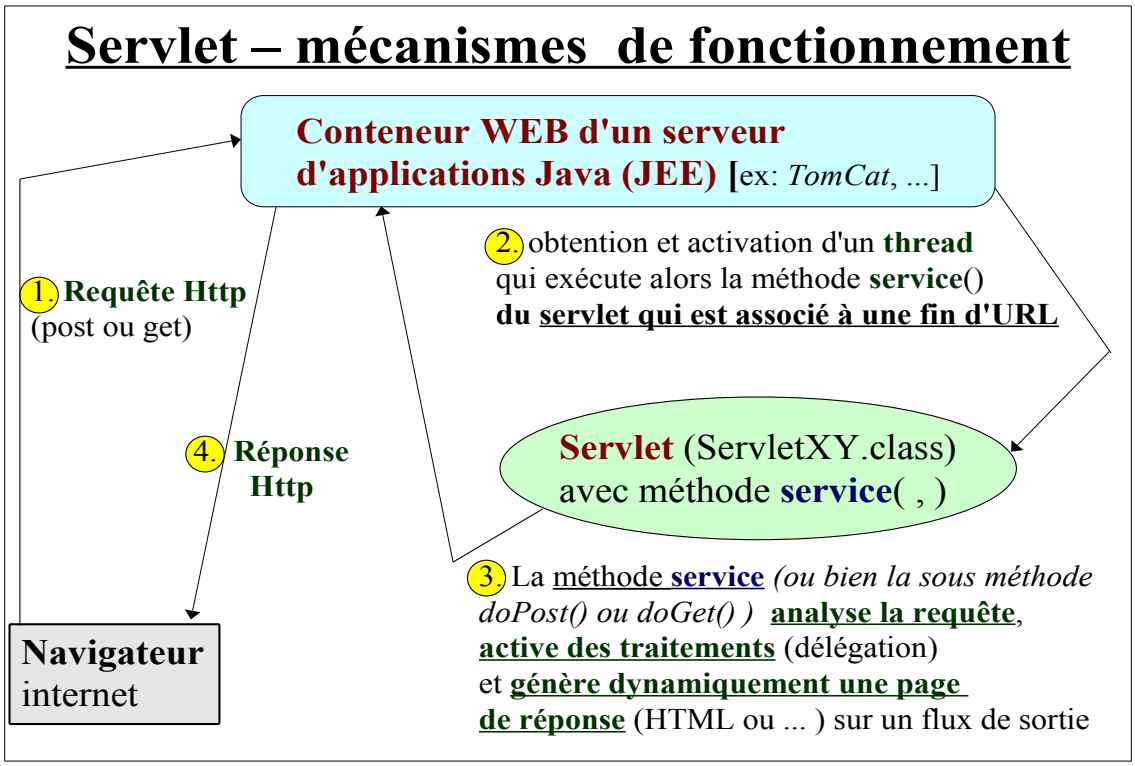
- application : ***web.xml***
- ressources du serveur : ***conf/server.xml***

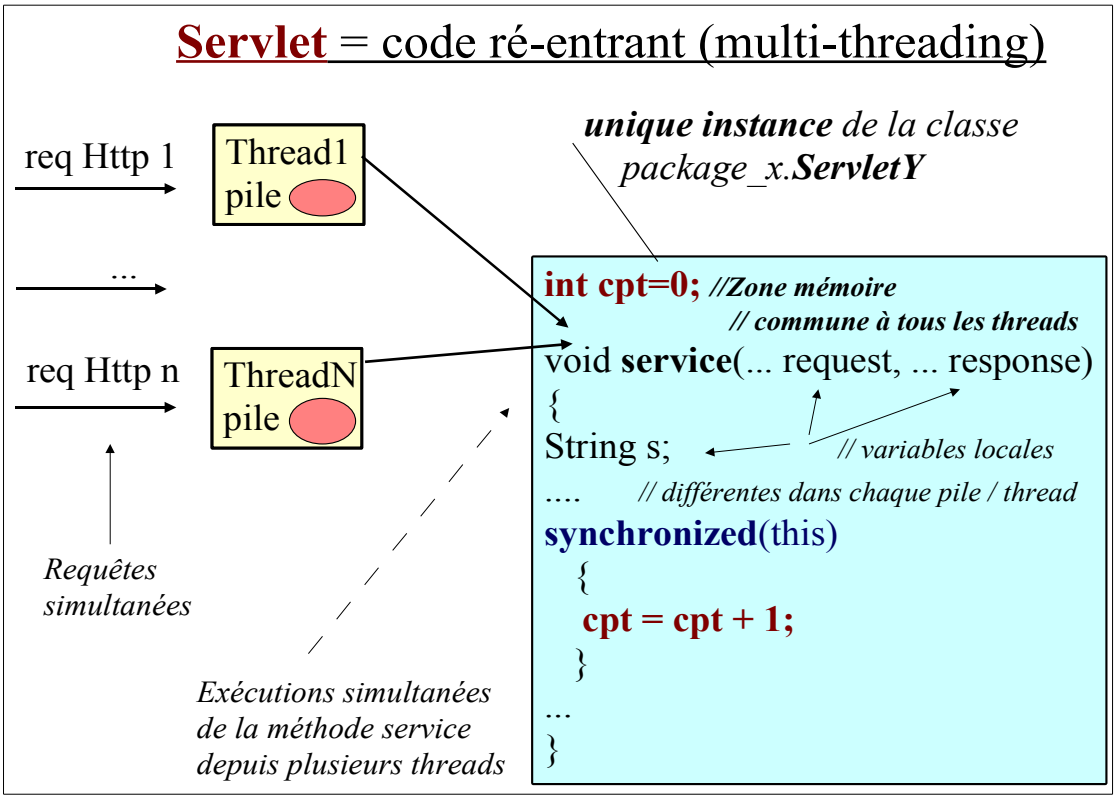


II - Servlet (essentiel)

1. Servlet

Un **servlet** est un **composant java** capable de **générer dynamiquement des documents WEB** (*html, xml, texte, image svg, ...*). Un ***servlet s'exécute toujours coté serveur*** au sein d'un "**conteneur WEB**" .





2. Servlets (structure, mise en oeuvre)

2.1. Mise en oeuvre avec Tomcat ou bien JBoss:

1. Télécharger Jakarta **TomCat** (depuis ***www.apache.org***)
   ou bien ***Jboss AS*** *incluant Tomcat* (depuis ***www.jboss.org***) et installer le produit.
2. Mettre à jour (si nécessaire) la variable d'environnement **JAVA_HOME** de façon à ce qu'elle pointe sur le répertoire du jdk à utiliser (ex: c:\jdk1.6 ou 1.5 ou ….).
3. Lancer **TomCat** en mode Serveur Web autonome (8080) via le script **bin\startup.bat** ou via **bin\tomcat6(w).exe** .
   ou bien Lancer **JBoss** en mode "default" (8080) via le script **bin\run.bat** .
4. Vérifier si tout marche bien via l'url **http://localhost:8080**
   ---
5. Au sein de l'IDE **Eclipse** , activer le menu "***Windows/préférences***" et développer la partie "***Servers/Runtime***" .
   Paramétrer alors une référence vers le nouveau serveur Tomcat ou bien JBoss (en précisant sa version et le répertoire où il a été installé) .
6. Créer un nouveau ***projet*** ***(ex: "myWebApp")*** de type "***Dynamic Web Project***" en choisissant "Tomcat" ou "Jboss" comme serveur de test (runtime) au sein d'un IDE Java (ex: Eclipse) .
7. Vérifier au sein du projet , l'arborescence suivante de sous-répertoires (*folders*):
   src
   webContent
   WEB-INF
   classes
   lib
   NB: ***webContent*** comportera l'ensemble des fichiers non confidentiels de l'application web (ex: pages html , jsp , images).
   src comportera le code source (quelquefois confidentiel) des classes java .
   Les sous répertoires ***WEB-INF/classes*** et ***WEB-INF/lib*** sont imposés par JEE.
8. Vérifier en suite le paramétrage du projet (*click droit / properties / java build path* sous eclipse) de façon à ce que ***src*** soit le répertoire des fichiers sources ".java" et webContent/***WEB-INF/classes*** *(ou bien* ***"build"*** *recopié ensuite dans WEB-INF/classes)* soit le répertoire où seront placées les classes compilées.
   A ce niveau , il faut également vérifier si le fichier ***servlet-api.jar*** est présent dans une des ***librairies*** du projet.
9. Programmer ServletS1.java (dans package p1 dans src) *[exemples en fin de chapitre*
10. *Vérifier la présence du fichier* ***web.xml*** *dans le répertoire* ***WEB-INF*** *et modifier si besoin ce fichier de façon à ce qu'il comporte les éléments suivants:*

```
<web-app
   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"
   version="2.4">
...
   <servlet>
      <servlet-name>Servlet1</servlet-name>
```

```
    <servlet-class>p1.ServletS1</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servlet1</servlet-name>
    <url-pattern>/s1</url-pattern>
  </servlet-mapping>
</web-app>
```

11. **Déployer l'application** de l'une des 2 façons suivantes:

    * en activant le menu contextuel "click droit / **run as / run on server**" de l'IDE **Eclipse**.

    * en créant (via ***ant*** ou via **Export de Eclipse**) une **archive web** (ex: ***myWebApp.war)*** à partir de l'ensemble du contenu interne du répertoire *webContent* du projet et en recopiant ensuite cette archive au sein du répertoire **JBOSS_HOME/server/default/deploy** ou bien dans le répertoire **TOMCAT_HOME/webapps .**

12. **Tester** directement le servlet de l'application via l'url http://localhost:8080/myWebApp/s1 (ou bien indirectement en passant par un lien hypertexte ou un formulaire d'une page *index.html* ou autre).

Spécificités de Tomcat

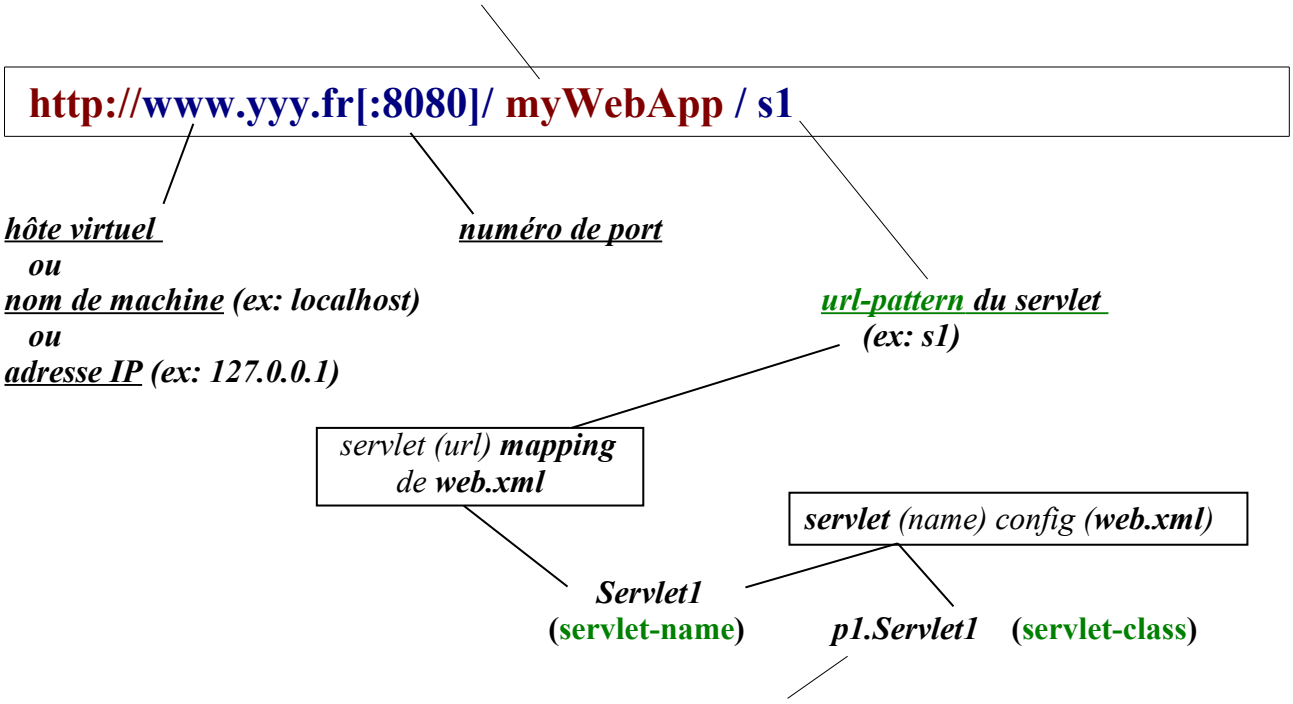
[Tomcat]**Manager** accessible depuis http://localhost:8080[/manager] permet d'arrêter et (re-)démarrer une application bien précise (*içi myWebApp*) .
Il faut pour cela connaître le mot de passe de l'utilisateur [ex: "admin"] ayant le rôle "**manager**" .
Ce mot de passe est choisi lors de l'installation et est stocké par défaut dans TOMCAT_HOME/***conf/tomcat-users.xml***.

3. Invoquer un servlet depuis un client http

3.1. URL standardisée pour invoquer un servlet:

racine de contexte (context-root) de l'application
Cette valeur correspond généralement à l'un des éléments suivants:

- *nom de l'archive (**myWebApp**.war)*
- *nom d'un sous répertoire de TOMCAT_HOME/webapps*
- *valeur de l'attribut **path** d'un "**Context**" de **Tomcat** (conf/....xml)*
- *valeur de context-root de META-INF/application.xml (si archive complète ".ear")*



NB: L'implémentation d'un ***Servlet*** (ou d'un ***JavaBean***) doit absolument être placée dans un package (içi p1) de façon à obtenir un fonctionnement correct sur tous les serveurs.

4. Cycle de vie d'un servlet

Un **servlet** est une **instance** d'une **classe Java** héritant de **HttpServlet**

1. Lorsque celle-ci est chargée en mémoire (via le serveur Web) , sa méthode **init**() est alors appelée. On peut redéfinir cette méthode si on a besoin d'effectuer quelques initialisations.

2. Ensuite, dès qu'un client va émettre une requête Http dont l'URL est le servlet , un nouveau thread sera lancé (ou récupérer depuis un pool) . Ce thread va alors exécuter la méthode **service**() .
   Si plusieurs clients ont lancé des requêtes en même temps , **plusieurs threads sont alors lancés en parallèle** ; **chacun d'eux exécutant** la même méthode **service**() **sur une seule et même instance du servlet** .

3. Juste avant que le servlet soit déchargé de la mémoire (lors de l'arrêt de l'application), sa méthode **destroy**() est automatiquement appelée.

NB : La méthode **service**() de la classe **HttpServlet** a une implémentation par défaut qui consiste à appeler la sous méthode **doGet**() , **doPost**() suivant le type de la requête Http (POST, GET, ...).

Le programmeur a donc 2 alternatives :
- soit redéfinir (programmer) la méthode **service**() (pas très conseillé mais ça marche).
- soit redéfinir (programmer) l'une des méthodes **doXxx**(). (***doGet*** ou ***doPost*** ou l***es deux***)

Démarche à adopter dans la plupart des cas:

Programmer doGet(...,...) et doPost(...,...) en faisant en sorte que ces 2 méthodes appellent si besoin une sous méthode commune (ex: doTask(...,...) ou doService(...,...)).

5. Code type d'un servlet:

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

public class DateServlet extends HttpServlet
{
  public void init(ServletConfig config) throws ServletException
        {
        super.init(config);
        // opérations supplémentaires à priori utiles.
        }

   public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
   {
     //String chFormat = request.getParameter ("format") ;

      Date today = new Date();

      response.setContentType("text/html");


     PrintWriter out = response.getWriter(); // encodage Unicode → cible (Html).
      out.println("<html><body>");
      out.println(today.toString());
      out.println("</body></html>");
   }
...
public void destroy() { super.destroy(); ... }
public String getServletInfo() {  return "My Job";  }
}
```

NB :

- La méthode (relativement pointue) **request.getInputStream()** peut éventuellement être utile pour récupérer (en binaire) le contenu d'une requête http complexe .
- La méthode request.**getMethod**() renvoie "POST" ou "GET" ou … selon de mode de la requête entrante.

6. Exemple simple complet:

A partir du formulaire HTML suivant:

```
<form name="recherche"
        action="http://localhost/myWebApp/RechercheServlet" method=post>
 <table border="1" width="72%" height="68">
  <tr>
   <td width="15%" height="1"><font color="#ff0080">Nom</font></td>
   <td width="85%" height="1"><input name="nom" size="65" ></td>
  </tr>
  <tr>... </tr> </table>
 <div align="center"><center><p><input type="submit" value="Rechercher" name="B1"></p>
 </center></div>
</form>
```

On émet une requête HTTP de type POST vers le servlet "**RechercheServlet**. Celui-ci va à son tour (via JDBC**) lancer une requête SQL vers une base de données (ex: MySql)**. Le résultat de la requête est ensuite mise en forme sous la forme d'une page HTML par le servlet pour être finalement renvoyé vers le navigateur.

```
import java.io.*;
import java.net.*;
import java.sql.*; // JDBC
import javax.sql.*; //DataSource
import javax.naming.*; //InitialContext
import javax.servlet.*;
import javax.servlet.http.*;


public class RechercheServlet extends HttpServlet {

DataSource ds=null; // pour obtenir des connections depuis un pool.

public void init() throws ServletException
// version sans paramètre issue de GenericServlet
    {
     try{
        String dbRefName = "java:comp/env/jdbc/MyDB";
                        // Nom logique JNDI
      // Obtention via JNDI de l'objet DataSource:
       InitialContext ic = new InitialContext();
       ds = (DataSource) ic.lookup(dbRefName);   //voir doc. De Tomcat  pour config pool.
       }
     catch(Exception ex)
        {
        this.log("Erreur obtention DataSource",ex);
        }
    }
```

```
  public void doPost( HttpServletRequest  req,  HttpServletResponse  res)
       throws ServletException, IOException
  {
     String paramNom=req.getParameter("nom");
     String chSql="SELECT * from TableDico ";
     res.setContentType("text/html");

     PrintWriter out = res.getWriter();
     out.println("<HTML><HEAD> <TITLE> dico </TITLE></HEAD>");
     out.println("<BODY> <H1> Resultats de la recherche </H1>");
     Connection cn=null;
     Statement st=null;
     ResultSet rs=null;
     try {
     cn = ds.getConnection(); // obtention via un pool
     st = cn.createStatement();
     String chWhere = "";
     if(paramNom!=null && paramNom.length()> 0)
         chWhere = "nom='"+paramNom+"'";
     if(chWhere.length()>0)
           chSql += (" WHERE " + chWhere);
     rs = st.executeQuery(chSql);
     while(rs.next())
          {
           String fNom = rs.getString("Nom");
           String fAdr = rs.getString("Adresse");
           out.println("<P> " + fNom + " , "  +fAdr);
          }
     }
     catch(Exception ex)
        { throw new ServletException(ex); }
     finally {
              try { rs.close(); } catch(Exception e1) {}
              try { st.close(); } catch(Exception e2) {}
              try { cn.close(); } catch(Exception e3) {} // restitution de cn au pool
              out.println("</BODY></HTML>");
              }
  }
}
```

NB: Cet exemple n'est ici que pour illustrer l'utilisation d'un simple Servlet isolé.
Dans une véritable application, le servlet collabore avec des pages JSP et des "JavaBeans" en arrière plan (modèle MVC2 qui sera vu plus tard).

7. Considérations sur le multi-threading

7.1. Accès aux bases de données :

Dans l'exemple précédent, on aurait pu être tenté d'initialiser une fois pour toute une connexion jdbc:

```
Connection cn=null; // référence sur future connexion en lecture seule vers une base
                    // de données.
public void init() throws ServletException
    {
     try{
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      String chUrl = "jdbc:odbc:dico";
      cn = DriverManager.getConnection(chUrl);
        } catch(Exception ex) { throw new ServletException(ex); }
    }

public void destroy()
    {
   try { if(cn!=null) cn.close(); } catch(Exception ex)
            { getServletContext().log( ex.getMessage()); }
    }
```

Ce qui pourrait fonctionner correctement dans le cas où la connexion vers la base de données est en lecture seule.

En mise à jour, il peut être **très dangereux** de ***mettre à jour une même base de données*** depuis *plusieurs threads exécutant au même moment une même fonction service()* qui utilise **un seul et même objet « connexion »** (surtout si le mode ***auto-commit est désactivé***).

➔ Pour effectuer des mises à jour (ou bien pour obtenir des résultats bien frais) , ***il est d'usage d'obtenir et de libérer l'objet de type java.sql.Connection au sein de la fonction service()***. Malheureusement, cette opération est coûteuse en terme de ressources. **Pour obtenir de bonnes performances , il faut utiliser des pools de connexions .**

7.2. Synchronized :

Par défaut, la méthode **service**() d'un thread (ou l'une de ses sous fonctions **doPost**() ou **doGet**()) **peut très bien être appelée par plusieurs threads au même moment sur une seule et même instance de la classe de Servlet** .

- Etant donné que chaque Thread comporte sa propre pile, il n'y a aucun problème tant que l'on ne manipule que des variables locales et les paramètres request et response de la fonction.
- Par contre, dès que la fonction service() (ou doPost() / doGet()) souhaite manipuler une variable d'instance, il faut synchroniser l'accès à celle-ci :

```
synchronized(this)
{
this.compteur ++ ;
}
```

III - Java EE (évolution des spécifications)

1. Architecture JEE et évolutions

1.1. JEE en tant qu'ensemble d'API & conteneur JEE

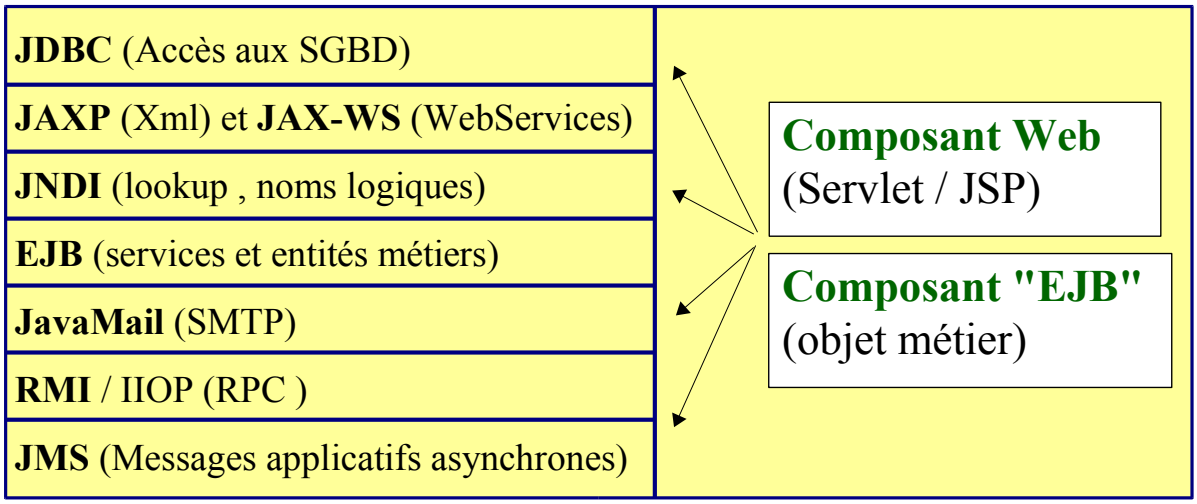
JEE (*signifiant Java Enterprise Edition*) peut être vu comme un ensemble d'API permettant de développer des applications évoluées à déployer sur un serveur d'entreprise.
Les API de JEE se rajoutent à celles du JDK (base JSE) . Elles concernent essentiellement les aspects "présentation WEB" , "EJB" , ... et "JMS" .

JEE & API

Java EE (+ EJB,JSP,Servlet,JMS, ….)

Java SE (Java Standard Edition – jdk 1.2 , 1.3 , 1.4, 1.5 , 1.6)

JEE peut être vu comme un modèle d'architecture basé sur des "**container**" qui offrent des **services techniques orthogonaux** aux ***composants métiers***:

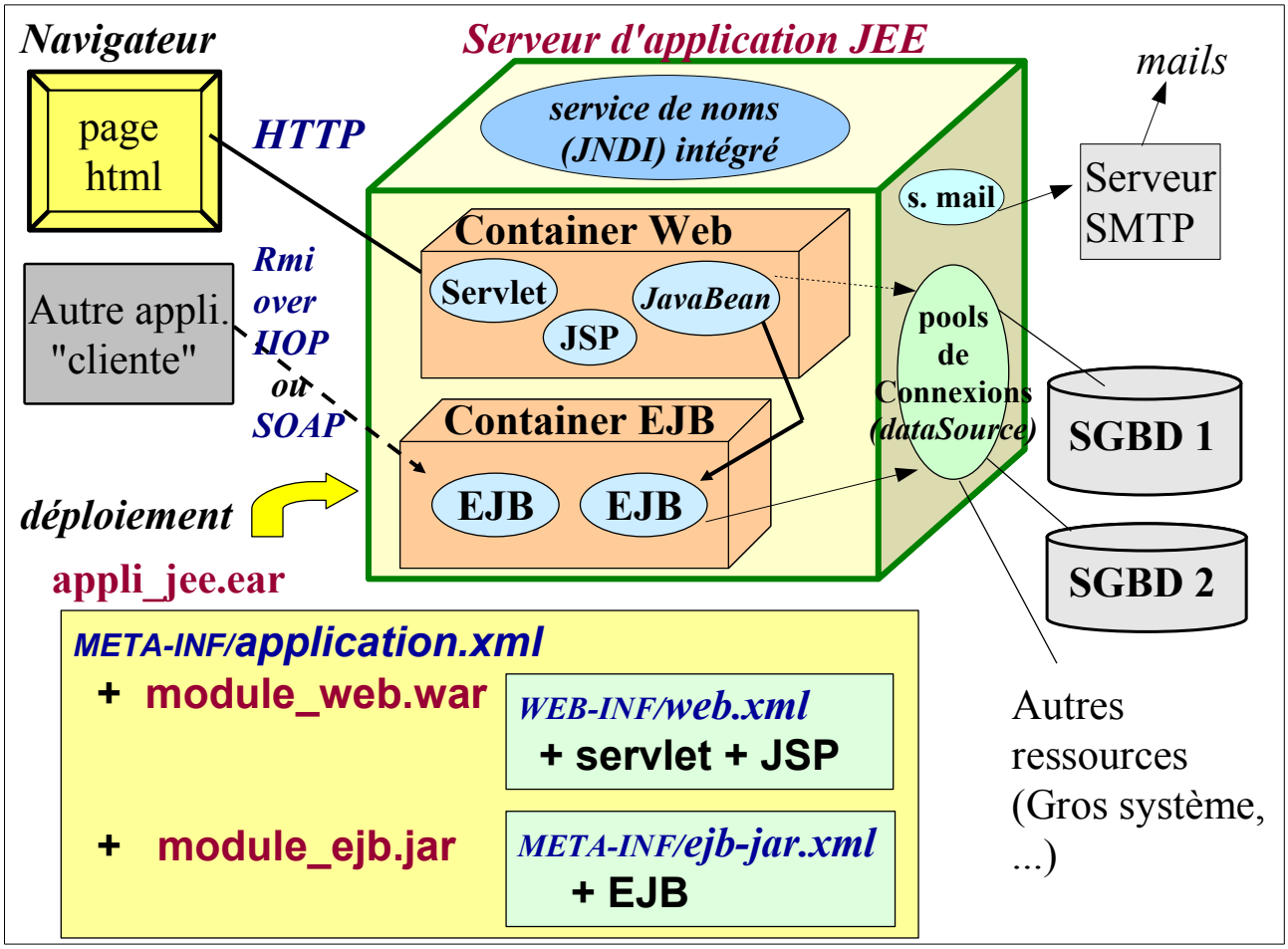


NB: La nouvelle version de J2EE est JEE5 , il vaut mieux utiliser maintenant le terme générique "JEE" pour s'adapter aux évolutions récentes.

2. Structure d'un serveur JEE

JEE peut également être vu comme un **modèle d'architecture** pour les **serveurs d'applications**.
Les **spécifications JEE** indiquent clairement le rôle des "**container**" : Ceux-ci doivent offrir aux composants applicatifs qu'ils hébergent un accès normalisé aux API standards de JEE .
Autrement dit , un **composant JEE** (*ex*: servlet , EJB, ...) fonctionne exactement de la même manière au sein des serveurs WebLogic , JBoss ou WebSphere car il peut appeler les mêmes

fonctionnalités (mêmes API) et qu'il expose lui même les mêmes points d'entrées pour la gestion de son cycle de vie.



Depuis J2EE 1.2 , le **déploiement d'une application JEE est standardisé**:

- Un fichier **".war"** (pour ***Web ARchive***) contient tous les composants "**web**" et les fichiers de configurations associés (***WEB-INF/web.xml*** , ...).
- Un fichier **".jar"** (pour ***Java ARchive***) contient tous les composants "**EJB**" et les fichiers de configurations associés (***META-INF/ejb-jar.xml*** , ...).
- Un fichier "**.ear**" (pour *Enterprise ARchive*) regroupe différentes sous archives (".war", ".jar" , ...) et un fichier de configuration globale : ***META-INF/application.xml*** dont la balise **context-root** de l'**application WEB** indique l'**URL** relative de celle-ci.

Au lieu de parler de J2EE 1.5 , 1.6 ...., Sun/JavaSoft a préféré baptiser **JEE5, JEE6** les nouvelles version des spécifications de sa plate-forme Java de niveau entreprise .

Les principaux apports de ces nouvelles versions sont les suivants:

- **EJB3** (avec api **JPA** pour la persistance des données).
- Nouveau support des ***services WEB*** via l'api **JAX-WS** (mieux que JAX-RPC)
- Intégration du framework **JSF** dans la partie WEB
- **Partie Web** de niveau "Servlet 1.5 / Jsp 2.1" ***supportant l'injection IOC des EJB3 et des ressources JEE5.***
- ***Pour JEE6 : annotations sur les servlets et profiles avec parties facultatives***

2.1. Principaux serveurs d'applications (JEE)

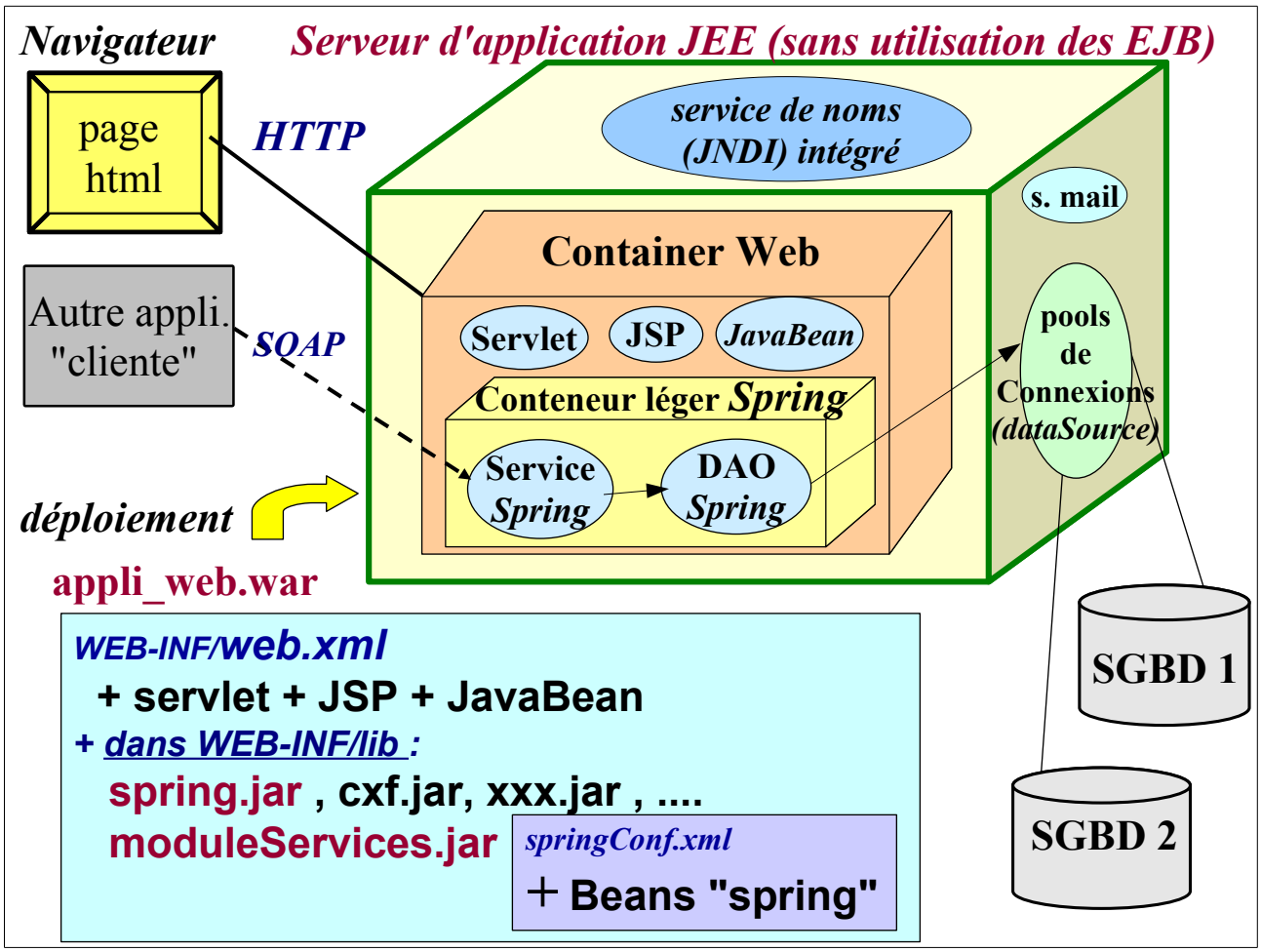
Serveurs d'applications	*Marques/Editeurs*	*Caractéristiques*
WebSphere	**IBM**	- Produit commercial avec le support d'une grande marque. - Serveur assez sophistiqué (très paramétrable et avec une bonne console d'administration). - surtout utilisé dans les grandes entreprises (banques, assurances, ...) - serveur assez complexe et assez cher .
WebLogic	**BEA --> Oracle**	- Autre bon produit commercial (à peu près aussi sophistiqué que WebSphere)
Jboss (4.2 , 5.1 , ~~6.0~~, 7.0)	**Jboss / Red Hat**	- Open source , existe depuis longtemps - Souvent innovant sur les technologies java (jmx , ejb3, ...) - Utilisation très simple pour les tests durant la phase de développement - console d'administration rudimentaire (en versions 4 et 5)
Jonas	**OW2 (INRIA + Bull + ...)**	- Open source (produit stable et sérieux) moins utilisé que Jboss car un peu en retard à l'époque des premières versions.
Geronimo	**Apache Group**	- Open source - Serveur récent (assez peu de recul)
Tomcat (*)	**Apache Group**	- Open source faisant office de référence sur la partie "conteneur Web". - Serveur JEE simplifié (partie "conteneur web" seulement (sans EJB)).
GlassFish	**SUN**	- Serveur JEE de SUN (en partie open source) assez complet et assez innovant sur certaines technologies (BPEL, ESB/JBI , ...) - Serveur récent (assez peu de retour/recul en production)

(*) NB: Tomcat peut être utilisé :
- soit de façon autonome en tant que mini serveur JEE (sans partie EJB)
- soit en tant que partie "conteneur web" intégrée dans un autre serveur JEE plus complet (Jboss , Jonas , ...). En règle générale on utilise/télécharge quasiment toujours une version pré-assemblée "Jboss + Tomcat" ou "Jonas + Tomcat".

2.2. Utilisation possible de "Spring" à la place des EJB

L' utilisation d'un conteneur d'EJB (en version 2 ou 3) n'est pas du tout obligatoire.

On peut préférer utiliser un conteneur léger tel que Spring .



2.3. Evolutions récentes : JEE6 / Tomcat7

Rappel : il y a toujours un décalage entre les sorties de Java_Se version n et Java_EE version n

ex: le jdk6 est apparu en 2007 et JEE6 en "fin 2010 / début 2011" .

La partie "web" des spécifications **JEE6** est prise en charge par la **version 7 de Tomcat**.

JEE6 et Tomcat7 apporte les nouvelles fonctionnalités suivantes:

- **configuration par annotations java** (directement dans le code d'un servlet) pour éventuellement simplifier le contenu de WEB-INF/web.xml
- support des spécifications **3.0** pour les **servlets** et **2.2** pour les pages **JSP**
- NB: **Tomcat 7** nécessite le **jdk 1.6** pour fonctionner .

2.4. Configuration par annotations

Jusqu'à l'époque JEE5 / Tomcat 6 , il fallait obligatoirement paramétrer un servlet au sein du descripteur de déploiement "***WEB-INF/web.xml***" pour qu'il puisse fonctionner .

A partir de la la version **JEE6** , [**servlet-api-3.0**] , la configuration xml n'est plus obligatoire si elle est compensée par de la configuration sous forme d'annotations JAVA insérées dans le code source d'un servlet .

Exemple:

```
//@WebServlet(name="mytest", urlPatterns={"/myurl"})
@WebServlet("/myurl")
public class TestServlet extends javax.servlet.http.HttpServlet {
  ...
}
```

Principales annotations sur la partie WEB:

@WebServlet avec attributs facultatifs ***name***, ***urlPatterns***, ***initParams***	Paramétrages d'un servlet (name=<servlet-name> , ...)
@WebInitParam(name="....", value="...")	Pour valeurs (en dur ???) du tableau initParams de @WebServlet
@WebFilter	Paramétrages d'un filtre web
@WebListener	Paramétrages d'un Listener Web(pour traitements au démarrage de l'application dans tomcat ou ...)
@GET, @POST, ...	Pour préfixer des méthodes de noms quelconques qui assurent les mêmes fonctionnalités que doGet() , doPost() , ...
@MultiPartConfig	Pour mime "multipart" sur Servlet avec méthodes getParts() et getPart()

2.5. web-fragment (depuis Servlet 3 , Tomcat 7)

Librairie/framework "xxx.jar" avec ***META-INF/web.xml*** (secondaire) :

```
<web-fragment>
  <servlet>
    <servlet-name>myservlet</servlet-name>
    <servlet-class>samples.MyServlet</servlet-class>
  </servlet>
  <listener>
    <listener-class>samples.MyListener</listener-class>
  </listener>
</web-fragment>
```

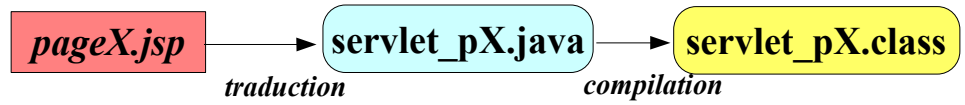
Autre apport de l'api "Servlet 3.0" : mode ***asynchrone*** (non bloquant) .

IV - Pages JSP

1. Pages JSP

Présentation des pages JSP

- **page xml ou html comportant des portions** (code java entre <% et %> ou bien balises spéciales) **interprétées coté serveur**.
- Les pages jsp sont essentiellement utilisées pour contrôler l'affichage des éléments.
- Les mécanismes internes de J2EE transforment la page Jsp en un servlet qui est ensuite automatiquement compilé avant l'exécution .



exemple

```
<% java.util.Date d = new java.util.Date(); %>
<html>
<body>
Date = <b> <%= d.toString() %> </b>
</body>
</html>
```

Les pages JSP peuvent assurer les mêmes fonctionnalités que les servlets.

Ils sont **beaucoup plus simples à écrire** (surtout en ce qui concerne l'encodage de l'affichage) .

NB:

- Des **taglib** (= bibliothèque de balises spécifiques codées sous forme de classes java) peuvent éventuellement être placées et interprétées au niveau des pages JSP pour simplifier la syntaxe.
- Le répertoire TOMCAT_HOME/**work**/Catalina/localhost/myWebApp comporte le code des servlets automatiquement générés à partir des pages JSP de l'application .
  Remarque: si Tomcat est lancé depuis eclipse en phase de test/développement, le répertoire "work" est alors situé dans "workspace/.metadata\.plugins\org.eclipse.wst.server.core\tmp0"

2. Intérêts des pages JSP

Un **servlet** est un morceau de code qui produit des lignes de texte correspondant souvent à une page HTML. Si l'on veut obtenir un résultat joli (bien présenté), cela devient généralement assez fastidieux car il faut écrire plein de lignes du genre *out.println(" <...>....<...>")* rien que pour gérer l'aspect "look" de la page à générer.

Une **page JSP** est à l'inverse un **fichier texte** (dérivant d'une page HTML ou XML**) qui comporte dès le départ quasiment toutes les balises liées aux aspects "structure" et "présentation"** et qui **contient quelques morceaux de code java qui seront utilisés pour activer certains traitements et générer dynamiquement certaines données**.

Une **page JSP** est beaucoup plus **facile à écrire** qu'un servlet (==> meilleure productivité).
Les **portions de code java** sont simplement **encadrées** par **<%** et **%>** .

Une page JSP comportant quelques instructions **Java est en fait automatiquement transformée** (de façon interne et complètement transparente) **en un servlet Java** qui est **compilé une bonne fois pour toute** et qui assure un **bon niveau de performance**.

3. Mise en oeuvre au sein d'une application Web

Une **page JSP** peut être directement déposée dans le répertoire ***webContent*** d'un projet Java (sous eclipse) ou bien dans un sous répertoire quelconque (ex: jsp).
Les pages JSP sont déposées au même niveau que les éventuelles pages HTML de l'application Web java/j2ee .Il n'est pas nécessaire de renseigner les pages JSP au sein du fichier **WEB-INF/web.xml .**

Pour le reste de la structure de l'application (**WEB-INF/web.xml**) et du déploiement (***myWebApp.war***) les règles à respecter sont les mêmes que celles exposées pour les servlets.

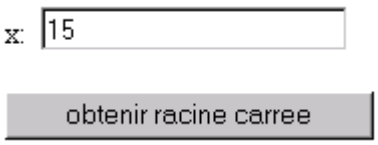
L'**url** menant à une **page jsp** est de la forme :

http://www.yyy.fr[:8080]/ myWebApp / page1.jsp

4. Exemple simple

essai.jsp

```
<html>
<%@ page session="false"%>
<body bgcolor="white">
<% String chX=request.getParameter("x"); %>
<% if(chX==null)
        {%>
        <form method=post>
                x:  <input name="x"><br/><br/>
                <input type=submit value="obtenir racine carree">
        </form>
        <%} else
        {%>
        <% int x = Integer.parseInt(chX); %>
          La racine carrée de <i><%=x%></i>
          vaut <b><%=Math.sqrt(x)%></b>
        <%}%>
</body>
</html>
```



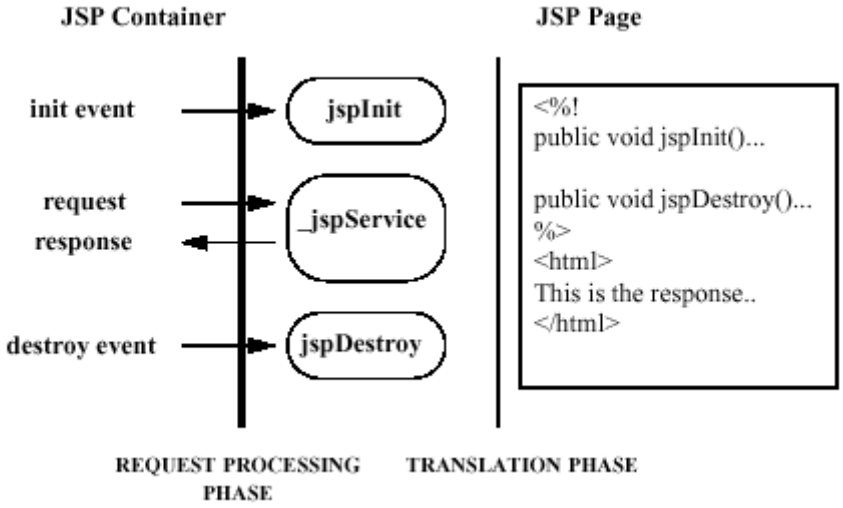
==>

La racine carrée de *15* vaut **3.872983346207417**

Cette page se rappelle elle-même et est un peu confuse. Le modèle MVC2 sera plus clair.

5. Principes fondamentaux

Le modèle **objet JSP** (depuis la version 1.1) s'intègre dans celui des **servlets Java** (version 2.2 et supérieure) de la façon suivante:



Commentaires :

Lors de la première requête vers une page JSP, le conteneur web effectue une transformation (translation) qui consiste à générer le code d'une nouvelle classe de servlet en fonction du code de la page JSP. Ce servlet sera ensuite compilé une bonne fois pour toute et servira à traiter toutes les autres requêtes ultérieures.

Les méthodes **jspInit()** et **jspDestroy()** que l'on peut éventuellement **déclarer** dans un bloc **<% ! %>** de la page JSP seront (si elles existent) automatiquement invoquées à l'initialisation et au déchargement du servlet en mémoire conformément au cycle de vie étudié dans le chapitre précédent.

Le reste de la page JSP (code HTML + scriplets java entre <% et %>) sera essentiellement utilisé pour générer le code de la méthode **_jspService()** du servlet.
Au sein de cette méthode, beaucoup d'objets seront prédéfinis de la façon suivante :

```
Object page = this ; // le servlet courant
PageContext pageContext = _jspFactory.getPagecontext(this,request,response, ….);
HttpSession session= getSession();
ServletContext application  = pageContext.getServletContext();
JspWriter out = pageContext.getOut() ;
ServletConfig config = pagecontext.getServletConfig() ;
```

➔ **Objets implicites de la page JSP.**

Tout ceci permet de faire facilement le rapprochement avec le chapitre précédent.

- La nouvelle classe **javax.servlet.jsp.JspWriter** permet de gérer la « ***bufferisation*** ».
- La nouvelle classe **javax.servlet.jsp.PageContext** permet d'accéder à tous les environnements liés à la page : ***scope(portée)*** = **application**, **page**, **session**, …..

6. Principaux points de syntaxe "JSP" :

<% scriplet; %>
<%=expression%>
<%! declaration ;%>

6.1. Déclarations

Une déclaration de variable peut être encadrée par **<%!** et **%>**

exemple:

<%! int compteur=0; %>

ATTENTION :

La variable compteur (provenant d'une déclaration **<% ! %>**) sera généré en tant que **variable d'instance du servlet** (résultant de la translation) et non pas en tant que variable locale de jspService() .
Conséquences :

- Si plusieurs requêtes arrivent simultanément ➔ plusieurs Thread en // ➔ éventuelle nécessiter d'un bloc en **synchronized(this)** .
- Une telle variable est accessible depuis plusieurs méthodes (jspInit() , jspDestroy() , jspService() , …).

NB: la syntaxe **<% ! %> peut également être utilisée pour déclarer des méthodes de la future servlet (page Jsp translatée) :**

Exemple :

```
<% !
int compteur;

public String getStringDate()
{
String chDate =
java.text.DateFormat.getDateTimeInstance(
            java.text.DateFormat.FULL,
           java.text.DateFormat.FULL).format(new java.util.Date()) ;
return chDate;
}

public void jspInit() { compteur=0; }
public void jspDestroy() { … }

%>
```

6.2. Scriptlets

Un **scriptlet** est une **portion de code java** placée entre **<%** et **%>** et qui sera introduit dans la méthode _jspService() de la classe de servlet générée par la phase de translation.

Exemple :

```
<% int n=5 ;
  for (int i=0 ;i<x ;i++)
   { %>
        <b> blabla </b> <br/>
  <%
    } %>
```

6.3. Expressions

Une **expression** (introduite par **<%= %>** et sans ;) est une **portion de code java retournant une valeur qui sera automatiquement insérée dans le flux textuel de la réponse via un out.println() implicite.**

Exemple :

<%int x=6 ; %> Le carre de **<%=*x*%>** vaut **<%=*x***x*%>** **.**

6.4. Directives

Une **directive** (introduite par la syntaxe **<%@ %>**) est une **indication qui servira à générer le code du servlet durant la phase de translation**.

Inclusion **statique** d'un fichier texte **dès la phase de translation/compilation**:

<%@ include file="Subpage.jsp" %>

Attention: Dès éventuelles modifications dans la sous page "Subpage.jsp" ne seront prises en compte au niveau de la page Jsp englobante que lorsque celle-ci sera elle même re-transformée en servlet et recompilée.

Importation de packages:

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

Page participant à la session ?

<%@ page session="false" %> *true par défaut*

Principaux autres attributs de la directive page:

attribut	signification	valeur par défaut
buffer	"none" ou "24kb" ou … (taille buffer de **JspWriter** dont **out** est l'instance prédéfinie)	Environ 8ko
autoFlush	true : Flush automatique si buffer plein . false : exception si débordement du cache	"true"
isThreadSafe	"false" -> comportement "SingleThreadModel"	"true"
info	Chaîne d'info récupérable via **getServletInfo**()	
errorPage	**url vers une page d'erreur** si exception non récupérée.	
isErrorPage	Indique si la page jsp courante est une page d'erreur. Si tel est le cas, l'objet implicite **exception** fait référence à l'instance de java.lang.Throwable soulevée par une autre page.	"false"
contentType	Type MIME de la réponse	"text/html; charset=ISO-8859-1"

6.5. Actions jsp

Inclusion dynamique d'un fichier au moment de l'exécution:

```
<% if(....) %>
   <jsp:include   page="...."  />
```

Redirection vers un autre contenu (autre page):

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

Pour de plus amples informations , veuillez consulter :

- Les exemples livrés avec Tomcat
- Les spécifications (jsp.pdf) www.javasoft.com/..

6.6. Objets implicites (prédéfinis) disponible dans une page JSP:

request , **response** , **session** , **config** ==>comme pour un servlet.

page ==> page courante

pageContext ==> contexte de la page courante (dépend de l'environnement)

out ==> flux de type JspWriter servant à écrire la réponse

application ==> objet de type **ServletContext** (*Commun pour tous les utilisateurs*).

7. JSP2

7.1. Accès à un JavaBean préparé en amont (JSP2)

Au sein d'un texte d'une page **JSP2** on peut directement utiliser la syntaxe ***${beanXxx.proprieteYyy}*** pour afficher une propriété d'un "javaBean" préparé en amont par un servlet dans le cadre "mvc2" classique.

Attention: ceci n'est possible qu'au sein d'une application Java/WEB de niveau >= 2.4 (déclarée comme telle dans WEB-INF/web.xml et ne pouvant être gérée que via une version >=5 de Tomcat).

Nb: La portée (**scope**) permet de préciser à quel endroit est rattaché le composant JavaBean. Ceci a une influence sur la durée de vie de l'objet et sur sa plage d'accessibilité:

Portée (scope)	Durée de vie	Accès possibles depuis ...
page	_JspService() associée à la page JSP courante	page JSP courante seulement
request	Tous les servlets et pages Jsp qui s'enchaînent (collaborent) pour traiter la requête Http courante.	servlets et pages Jsp reliés par rd.forward(request,...) ou rd.include(request,...).
session	durée de vie de la session (HttpSession) (voir chapitre suivant)	Tous les servlets et pages Jsp de la même application Web.
application	Jusqu'à l'arrêt (ou ré-initialisation) du container web (voir ServletContext)	Tous les servlets et pages Jsp de la même application Web.

7.2. Objets prédéfinis accessibles dans une page JSP2:

pageScope	map d'objets "attributs" de portée "page"
requestScope	map d'objets "attributs" de portée "request"
sessionScope	map d'objets "attributs" de portée "session"
applicationScope	map d'objets "attributs" de portée "application"
pageContext	
param	map des valeurs simples des paramètres HTTP (équivalent de request.getParameter(paramName))
paramValues	map des valeurs multiples (String[]) des paramètres HTTP (équivalent de request.getParameterValues(paramName))
header	map des valeurs simples des entrées de la requête HTTP

headerValues	map des valeurs multiples (String[]) des entrées de la requête HTTP
cookie	map des valeurs des cookies (si valeur multiple ==> 1ere valeur)
initParam	map des "initParam" attachés au contexte globale de l'application au sein de ***WEB-INF/web.xml***

***Exemples*:**

```
${pageContext.request.requestURI}
${sessionScope.couleurPreferee}
```

${param.age}*** ou bien ***${param["age"]}

Remarque importante:

${objXY.zzz} n'affiche rien (et ne génère pas d'exception) si objXY n'exite pas (vaut null) .

NB: Les expressions entourées par ${...} sont exprimées dans un langage spécifique "EL : Expression language" .

EL ressemble beaucoup à javaScript .

possibilité d'utiliser au sein des **EL** JSP2:

- la syntaxe xxx**[**indice**]** sur des tableaux.
- des expressions arithmétiques (+,-,*,/,% ou mod)
- des comparaisons (> , < , >= , <= , == , !=)

Désactivation éventuelle de l'évaluation EL au sein d'un groupe de pages JSP2:

```
<jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
</jsp-property-group>
```

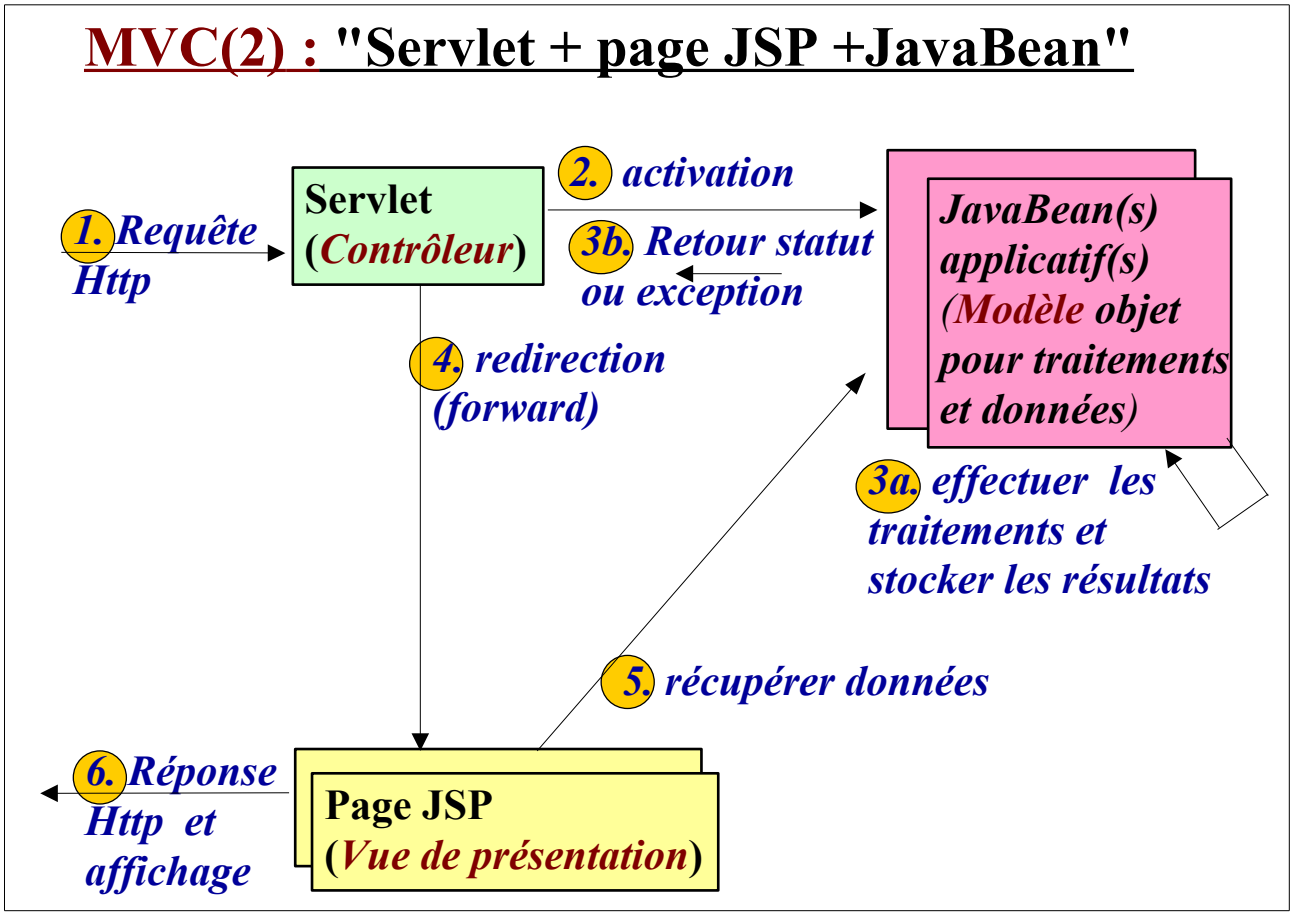
au sein de ***WEB-INF/web.xml***

NB: on peut également désactiver unitairement l'évaluation EL (${..}) via une directive **<%@ page *isELIgnored="false"* %>** à placer dans le haut d'une page JSP2.

V - MVC2 (Servlet + JSP + JavaBean)

1. Modèle MVC2

MVC = Modèle - Vue - Contrôleur



- Le **Servlet** joue le rôle de **contrôleur** : il reçoit une requête, lui applique un éventuel contrôle de saisie (**validation** quelquefois déléguée ou automatisée).

- Le contrôleur demande à un **JavaBean** d'**effectuer les traitements**. Le JavaBean effectue (ou bien ***délègue***) les traitements (accès Jdbc, ***EJB***,…) et récupère les résultats qu'il mémorise dans ses attributs (*ou dans d'éventuel(s) JavaBean(s) annexes de données)* . Ce(s) **JavaBean(s)** joue(nt) le rôle de **Modèle**.

- Le contrôleur après avoir reçu un statut (ok/ko) ou une exception effectue une **redirection** (**forward**) vers une page JSP pour l'affichage du résultat ou vers une page JSP d'erreur.

- La **page JSP** (jouant le rôle de **Vue**) ***récupère les données nécessaires à l'affichage auprès du JavaBean*** et génère (met en forme) la page HTML à renvoyer.

2. Collaboration --- (SSI & Redirection interne)

- **SSI** = **S**erver **S**ide **Include** ➔ Moyen d'incorporer une sous page JSP.

- Redirection via **forward** ➔ **Délégation de l'affichage vers un autre élément de la même application web (servlet , page jsp, …).**

2.1. Redirection depuis un servlet

```
RequestDispatcher rd ; // RequestDispatcher existe depuis la version 2.1 des servlets

rd = this.getServletContext().getRequestDispacher("/yyy/page2.jsp");
```

```
rd.forward(request,response);  // redirection vers autre url(html,servlet,JSP,…)
```

```
rd.include(req,rep); // SSI (Server Side Include) ➔ Ex: incorporation d'un bandeau.
```

Commentaires:

- La méthode **forward**() de l'objet ***RequestDispatcher*** permet de **rediriger** vers une autre entité (servlet, page JSP ou HTML) qui prendra alors à sa charge **l'écriture finale (et généralement complète) de la réponse .**
- La méthode **include**() de l'objet ***RequestDispatcher*** permet d'**incorporer le résultat d'une autre entité au sein de la réponse qu'il faudra alors compléter** .

Différentes façons d'obtenir l'objet « RequestDispatcher » :

rd = **request**.getRequestDispatcher**("ServletY");** *// chemin éventuellement relatif*

rd = **getServletContext()**.getRequestDispatcher**("/xxx/page_y.jsp");**
// chemin absolu par rapport au context root de l'appli web.

rd = **getServletContext().getNamedDispatcher("***NomLogiqueDuServlet***");**
// nom logique = celui de la balise <servlet-name> de web.xml.

Passage de paramètres d'un servlet à un autre (ou vers une page jsp):

Lorsqu'un **servletA** invoque un **servletB** (via rd.**include(request,response)** ou bien rd.**forward(request,response)**), **il peut éventuellement placer des attributs dans l'objet request**. **Ceux-ci pourront alors être récupérés par le servletB invoqué**.

C'est pour cette raison que l'interface **ServletRequest** comporte les méthodes **getAttribute**(), **setAttribute**(), **removeAttribute**() et **getAttributeNames**().

Cette importante considération est à l'origine de la portée **scope="request"** des JSP.

2.2. Redirection depuis une page jsp

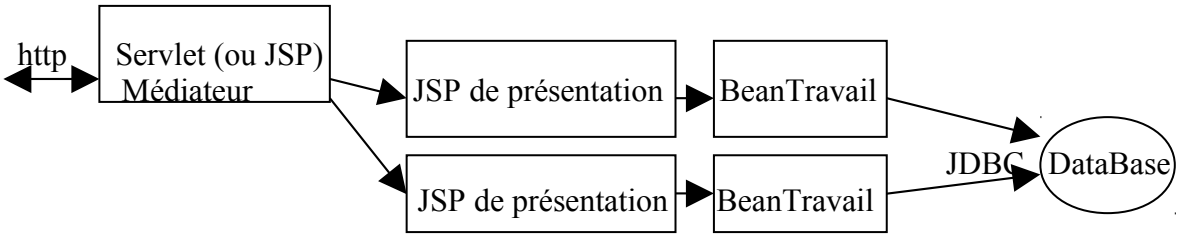
Inclusion dynamique d'un fichier au moment de l'exécution:

<jsp:include page="bandeau.jsp" />

Redirection vers un autre contenu (autre page):

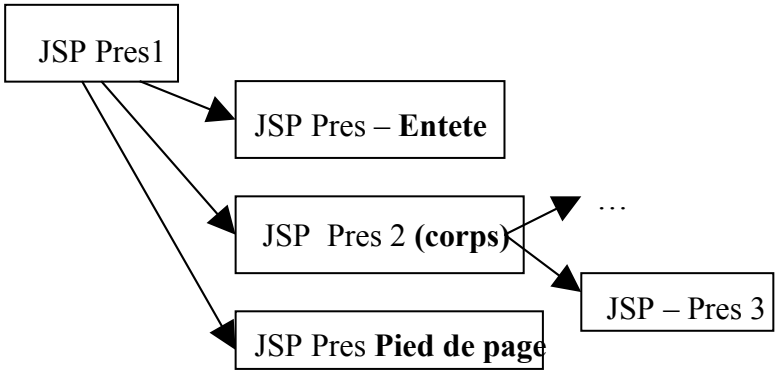
<jsp:forward page="aff_res_v1.jsp" />

2.3. Vue Médiateur



2.4. Vue Médiateur-Composite

Idem que précédemment mais en imbriquant les JSP de présentation selon le modèle composite:



MVC version 2 correspond à une variante améliorée où il n'y a qu'un seul contrôleur (servlet) générique supervisant toute les navigations entre les pages (avantage: un seul point central à sécuriser et à paramétrer dans web.xml et toutes les navigations sont centralisées).

2.5. Variante 1 (MVC simplifié):

racine.html

```
<html> <body>
<form method="POST" action="ServletCalculRacine">
 x=<input name="x"> <input type="submit" value="square root" >
</form>
</body></html>
```

Cette page html permet d'activer le servlet ci après:

code du servlet "calcul.Racine" faisant office de contrôleur:

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

public class Racine extends HttpServlet
{
  public void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException
    {
       double x  = Double.parseDouble( request.getParameter ("x") );
                if(x<0) throw new ServletException("Impossible de calculer la racine carrée
                                                    d'un nombre négatif");
        else {
             BeanCalcul obj = new BeanCalcul();
             obj.setX(x);              obj.calculerRacineCarree();
             request.setAttribute("objCalcul",obj);
            RequestDispatcher rd =
                        getServletContext().getRequestDispatcher("/racineResult.jsp");
            rd.forward(request,response);
            }
    }
}
```

Le servlet délègue les traitements vers le JavaBean "***calcul.BeanCalcul***" dont le code est le suivant:

```
package calcul;
public class BeanCalcul
{
private double x=0;
private double res=0;
public void setX(double x) { this.x=x; }
public double getX() { return x; }
public double getRes() { return res; }
public void calculerRacineCarree() { res = Math.sqrt(x); }
}
```

La page JSP "racineResult.jsp" correspond à la vue (Simple présentation des résultats récupérés dans le JavaBean):

old_racineResult_avec_syntaxe_has_been.jsp

```
<jsp:useBean scope="request" id="objCalcul" class="calcul.BeanCalcul" />
<html><body>
La racine carree de <i> <jsp:getProperty name="objCalcul" property="x"/> </i> est
<b><jsp:getProperty name="objCalcul" property="res"/></b>
</body></html>
```

racineResult.jsp

```
<html><body>
La racine carree de <i> ${requestScope.objCalcul.x} </i> est
<b> ${requestScope.objCalcul.res} </b>
</body></html>
```

La racine carree de *16.0* est **4.0**

Rappel: la syntaxe ${xxxScope.yyy.zzz} est valable depuis JSP 2.0 et Tomcat 5 .

Tomcat 5 et 5.5 → JSP 2.0
Tomcat 6 –> JSP 2.1
Tomcat 7 → JSP 2.2

2.6. Variante 2 (MVC2 sans framework automatisé)

Le point de départ est maintenant une page jsp. Ceci présente l'avantage de pouvoir afficher un éventuel message d'erreur et de pouvoir récupérer les anciennes valeurs saisies.

```
<html>
<body>
<!-- affichage d'un éventuel message d'erreur -->
<font color='red'> ${objCalcul.msg} </font>

<form method='POST' action='MVC2Calcul'>
   <input type='hidden' name='action' value='calcul_racine' />
   x=<input name='x' value='${objCalcul.x}' />
  <input type='submit' value='square root' />
</form>
</body>
</html>
```

Le Bean de données (Modèle) peut éventuellement comporter une propriété "msg" de façon à véhiculer un message d'erreur:

```
package calcul;

public class DataCalcul /* Modèle */
{
private double x=0; // x
private double sq_rt=0; // racine carrée de x
private String msg=""; // éventuel message

public void setX(double x) { this.x=x; }
public double getX() { return x; }

public double getRacine() { return sq_rt; }
public void setRacine(double racine) { sq_rt=racine; }

public void setMsg(String m) { this.msg=m; }
public String getMsg() { return msg; }
}
```

En version MVC2, le servlet contrôleur devient un point central à l'ensemble de l'application Web qui supervise tous les enchaînements d'actions et qui effectue des redirections intelligentes pour demander de nouvelles saisies en cas d'erreur:

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

public class MVC2CalculServlet extends HttpServlet
{
   public void doPost(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException
   {
   String action=request.getParameter("action");
   if(action.equals("calcul_racine")) doCalculRacine(request,response);
    /* else if(action.equals("autre_action"))  doAutreChose(request,response); */
            else throw new ServletException("action non prévue");
   }

   public void doCalculRacine(HttpServletRequest request, HttpServletResponse response)
             throws ServletException, IOException
                 {
                RequestDispatcher rd = null;
                 String chUrl=null;    double x=0;
                 DataCalcul obj = new DataCalcul(); // Bean de données (Modèle simplifié)
                  try{
                      x  = Double.parseDouble( request.getParameter ("x") );
                     obj.setX(x);
                     if(x<0)
                         {
                         obj.setMsg("Impossible calculer la racine carrée d'un nbre négatif");
                         chUrl="/racine.jsp";
                          }
                         else
                          {  CalculRacine calculObj = CalculRacine.getInstance();
                           calculObj.calculerRacineCarree(obj);
                          chUrl="/racineResult.jsp";
                          }
                     }
                    catch(NumberFormatException ex)
                         {
                           obj.setMsg("La valeur saisie n'est pas une valeur numérique");
                           chUrl="/racine.jsp"  ;
                          }
                     catch(Exception ex)
                         { obj.setMsg(ex.getMessage());
                            chUrl="/racine.jsp"            ;
                          }

                      finally
                      {
```

```
                        request.setAttribute("objCalcul",obj);
                        rd = getServletContext().getRequestDispatcher(chUrl);
                        rd.forward(request,response);
                    }
    }
}
```

Le Bean de traitement est quelquefois utile si les traitements sont complexes:

```
package calcul; // Bean de traitement

public class CalculRacine {

        private static CalculRacine uniqueInstance=null;

        public static CalculRacine getInstance()
          {
                if(uniqueInstance==null)
                        uniqueInstance=new CalculRacine();
                return uniqueInstance; // Singleton
          }

        public void calculerRacineCarree(DataCalcul data)
        {       // Les traitements souvent plus complexes (ex: JDBC) justifient
                //  habituellement un bean de traitement.
                data.setRacine(Math.sqrt(data.getX())) ;
        }
}
```

Vue habituelle pour afficher les résultats:

```
<body>
La racine carree de <i> ${objCalcul.x} </i> est
<b><font color='blue'> ${objCalcul.racine} </font></b>
<p><a href='racine.jsp'>autre calcul</a></p>
</body> </html>
```

2.7. Variante 3 (MVC2 avec parties automatisées)

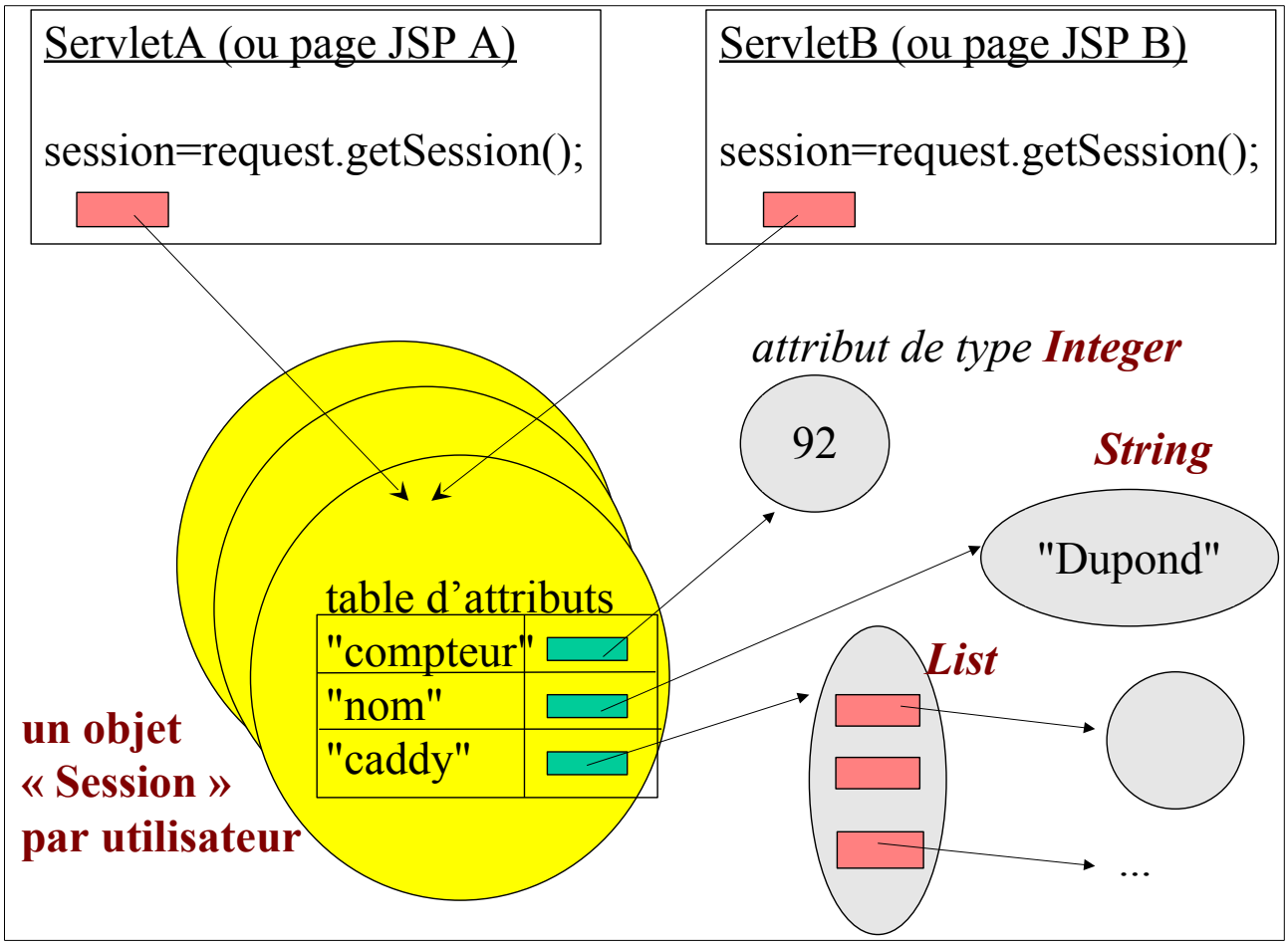
L'exemple de code vu précédemment n'est qu'**une variante possible parmi beaucoup d'autres**. Celle-ci peut s'avérer très longue à développer sur un gros projet (beaucoup de tests à effectuer) . D'autre part , on peut faire d'autres choix : Bean de validation , automatisations diverses via introspection , paramétrages xml et tables d'associations. Pour être productif et pour que l'ensemble soit bien structuré on utilise généralement des frameworks (**STRUTS**, ...).

VI - Session Http et ServletContext (application)

1. Session HTTP

Une **session HTTP** est un **objet (propre à chaque utilisateur)** qui est :

- maintenu en mémoire dans le "conteneur Web"
- utilisé pour établir un lien entre les différentes requêtes émises successivement par un même utilisateur (login.jsp , menu.jsp, ... , commande.jsp).



Les mécanismes internes des serveurs JEE utilisent des cookies (ou à défaut des ré-écritures d'URL avec suffixes) pour véhiculer un identifiant d'objet session et pour ainsi pallier le fait que le protocole HTTP est sans état**.**

Remarques techniques:

- Un objet "session" est automatiquement détruit coté serveur au bout d'un certain temps d'inactivité de la part de l'utilisateur *(timeout de session réglable et de 15/20 minutes par défaut)*.
- Dans le cadre d'un fonctionnement en cluster (application s'exécutant sur plusieurs serveurs) , les contenus des objets sessions sont quelquefois répliqués pour anticiper des pannes .

2. Gestion des Sessions HTTP

2.1. Session http

Une **session** est un objet qui est maintenu en mémoire au niveau du serveur et qui permet de **mémoriser certaines valeurs entre plusieurs requêtes Http successives issues d'un même client**. (Nb: **une session est visible depuis plusieurs servlets et pages «jsp» de la même application web**).

NB1:

Une session prend fin dans l'un des cas de figure suivants:

- Fin explicite demandée via l'instruction **session.invalidate()**
- Après un **certain temps d'inactivité** (durée paramétrable via **session.setMaxInactiveInterval(int nbSec)** ou *via* ***<session-config> <session-timeout>*** du descripteur de déploiement (en minutes dans *web.xml*)).

NB2:

Pour maintenir l'**identification d'une session** entre plusieurs requêtes successives transportées par le **protocole http sans état**, le container Web utilise l'un des moyens suivants:

- Un cookie créé et récupéré automatiquement (du genre ***jsessionid***) si le navigateur supporte les cookies (option pas désactivée au niveau du navigateur).
- Réécriture d'URL (avec paramètre supplémentaire jsessionid) si cookies désactivés.
  La réécriture d'url implique que toutes les URL vers un élément dynamique (Servlet, JSP) de notre application Web soit préparées de la façon suivante :

```
String chUrl = response.encodeURL("/myapp/servlet/ServletY" );
out.println("<a href='" + chUrl + "'> lien vers servletY </a>");
```

Autres méthodes:

- La méthode **getId()** de la classe HttpSession renvoie sous forme de String l'identifiant de la session.
- long nbMilliSecondesDepuis01_01_1970 = session.**getCreationTime**(); *//GMT*
  nbMilliSecondesDepuis01_01_1970 = session.**getLastAccessTime**(); *//GMT*
  out.println(new Date(nbMilliSecondesDepuis01_01_1970));

Ce premier exemple permet de gérer un compteur de type session qui s'incrémente à chaque appel successif du servlet concerné ainsi qu'une liste qui s'agrandit petit à petit:

```
...
HttpSession session = request.getSession(true /* create new one if ... */ );
...                          // il existe une version sans arg. de getSession (true=default value)
/* Il ne sert à rien de re-créer l'objet session dans une page jsp – c'est déjà fait  */

Integer intObj = (Integer) session.getAttribute("compteur");
 if (intObj == null)  intObj = new Integer(1);
     else intObj = new Integer( intObj.intValue() + 1 );
...
          session.setAttribute("compteur", intObj);
```

```
...
Vector vectObj = (Vector) session.getAttribute("liste");
if(vectObj == null)
    {
     vectObj =new Vector();
     session.setAttribute("liste",vectObj);
    }
vectObj.add(...);
```

Ce second montre comment rediriger la réponse sur une une autre page Web si la session vient d'être créée:

```
HttpSession session =request.getSession(true); // dans servlet seulement (déjà fait dans
                                                // page jsp où session est prédéfini.)
... if (session.isNew())
    { response.sendRedirect (welcomeURL); }
```

Autres fonctions intéressantes de l'objet Session:

session.**removeAttribute**(attrName);

Enumeration e= session**.getAttributeNames();**
while(e.hasMoreElements())
{ attr_name = (String) e.***nextElement***();
att_value = (String) session.getAttribute(att_name); … }

3. Notion d'application Web (ServletContext)

ServletContext contexte = this.**getServletContext**(); *// this = instance de HttpServlet*

3.1. ServletContext

Depuis la version 2.1 du Servlet SDK , l'objet **ServletContext** permet de jouer le rôle de l'objet *Application* de ASP(Micosoft). **Cet objet est lié (et accessible) à un groupe de servlet** . De plus cet objet est valable **(commun) pour tous les utilisateurs**.

De façon à gérer les valeurs liées à l'objet **ServletContext** , on peut utiliser les méthodes suivantes:

- **.getAttribute**("nomAttribut")
- **.setAttribute**("nomAttribut",obj)
- **.removeAttribute**("nomAttribut")

Depuis la version 2.2 , chaque contexte de servlet est affecté à un chemin d'accès spécifique sur le serveur Web (balise xml <**Context path**="appliY" **docBase**="c:/RepA/appliY" > de conf/*server.xml de Tomcat*).
String chPathName = servletContexte.**getRealPath**(cheminRelatif) ;

3.2. Obtention des valeurs des paramètres d'initialisation

Pour éviter d'utiliser des «valeurs en dur» et les recompilations associées, il est souhaitable de faire en sorte que le code Java soit en partie paramétrable.

- On peut récupérer via **getInitParameter("*nameParam*") ;**
  la **valeur d'un paramètre** présent dans le fichier **WEB-INF\web.xml**

<servlet>

```
<init-param>
   <param-name>nameParam</param-name>
   <param-value>valeur du parametre</param-value>
</init-param>
```

… </servlet> …

- On peut également découvrir la liste des paramètres existants (configurés) via la méthode **getInitParameterNames**() qui retourne une Enumeration.

NB:

- Des ***paramètres*** d'initialisations ***propres à un servlet*** sont rangés à l'intérieur d'une balise **<servlet>** de web.xml et sont récupérés via la méthode **getInitParameter() de la classe HttpServlet** (héritant de GenericServlet qui elle même implémente **ServletConfig**).
- Des ***paramètres*** d'initialisations ***globaux (valables pour toute l'application web et commun à tous les servlets)*** sont rangés à l'intérieur de <context-param> de web.xml et sont récupérés via de la méthode **getInitParameter() de la classe ServletContext** dont on peut récupérer l' instance via getServletContext():

<web-app>

```
<context-param>
    <param-name>nameParam</param-name>
    <param-value>valeur du parametre</param-value>
 </context-param>
<context-param>
   <param-name>PATH_X</param-name>
   <param-value>c:\repXXX</param-value>
 </context-param>
```

…<servlet> … </servlet> <servlet> … </servlet>
</web-app>

3.3. Structure d'une application web

```
monAppWeb\
        index.html
        login.jsp
        404NotFound.html
        images\
            logo.gif
            banner.jpeg
        docs\
            rapport_annuel.pdf
        WEB-INF\
                web.xml
                classes\
                    ServletA.class
                    ServletB.class
                    JavaBeanY.class
                lib\
                    XxxApi.jar
        …
```

- Le répertoire privé **WEB-INF** comporte des ressources qui ne sont pas destinées à être téléchargées vers le client.
- Le fichier **web.xml** correspond au **descripteur de déploiement** de l'appli-web.
  Ce fichier de configuration est fondamental.
- Le répertoire **WEB-INF\classes** est destiné à stocker les classes java correspondant aux servlets et aux JavaBean de l'application (des éventuels sous répertoires correspondent alors aux packages).
- Si le servlet a besoin de certaines **archives (xxx.jar)** , il faut alors placer celles-ci dans le répertoire **WEB_INF\lib** .

3.4. Déploiement

Une fois l'application Web (site) au point, on peut déployer tout son contenu dans une **ARchive Web** (fichier **.WAR**).

cd c:\RepMyApp *(répertoire contenant l'arborescence classique WEB-INF\classes , ...)*

```
jar –cf myApp.war *
```

puis copier **myApp.war** dans **%TOMCAT_HOME%\webapps**

Cette archive sera alors automatiquement décompactée par le conteneur Web (ex: TomCat), et le contenu sera directement accessible via une url du genre ***http://localhost:8080/myApp/…*** .

Nb1: Tomcat décompacte automatique l'archive **myApp.war** et associe automatiquement le context-root «**myApp**» à cette **application web** en se basant sur le **nom de l'archive.**

Nb2:
Le fichier *web.xml* peut éventuellement comporter des balises <servlet> … **<load-on-startup>** 10 **</load-on-startup>** </servlet> *où 20 est un numéro d'ordre (20 après 10)*.
Ceci permet de demander le chargement et l'initialisation en mémoire de certains servlets dès le démarrage du container Web.

Nb3:
web.xml peut comporter le nom par défaut d'une page d'accueil:

```
<web-app>
<welcome-file-list>
        <welcome-file> index.html </welcome-file>
        <welcome-file> index.jsp </welcome-file>
</welcome-file-list>
…
</web-app>
```

Nb4**:**

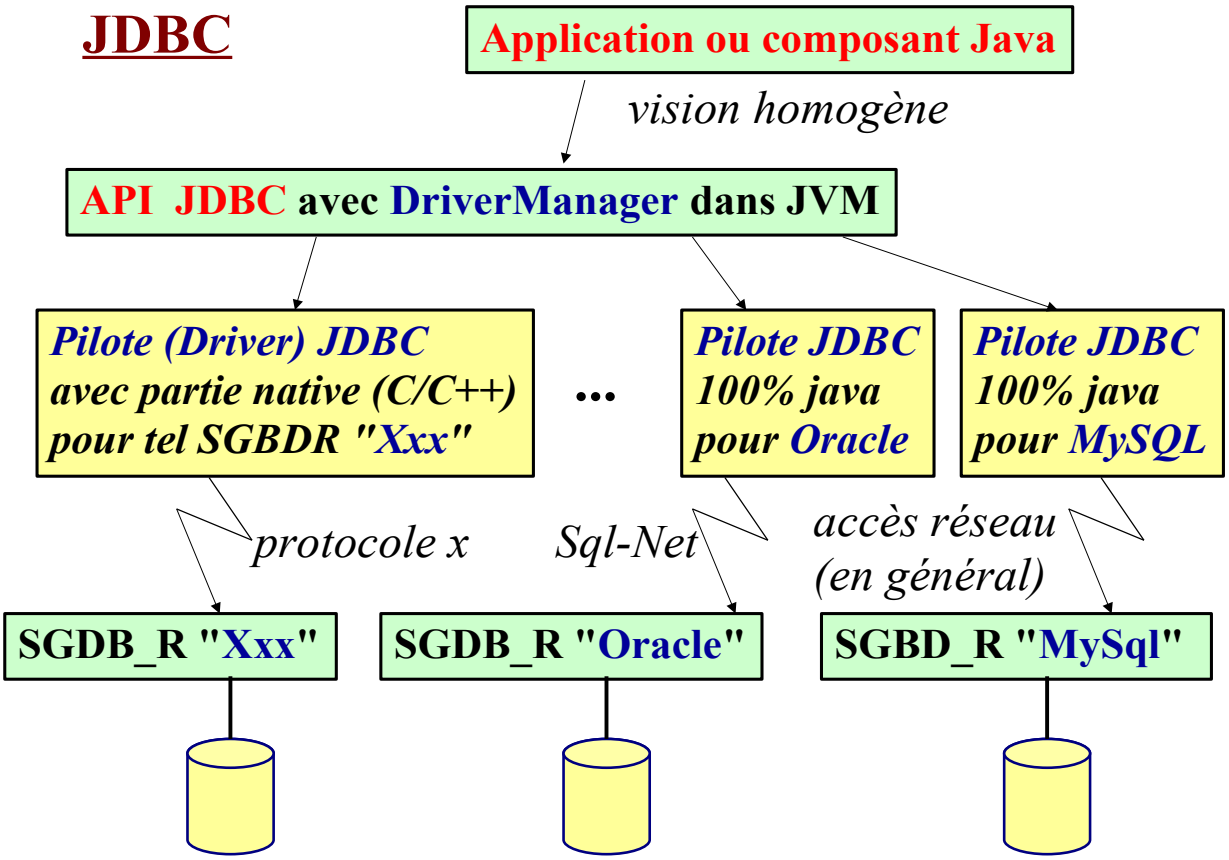
```
<web-app>
<distributable/>
…
</web-app>
```

La balise XML **<distributable/>** de web.xml permet d'indiquer que notre application web pourra être dupliquée (distribuée) au sein d'un **ensemble de container Web** fonctionnant en **cluster (plusieurs machines virtuelles Java sur plusieurs machines hôtes)** .
Il faut évidemment que le serveur d'application supporte cette fonctionnalité.
➔ Les temps de réponse peuvent rester corrects même s'il y a une grosse montée en charge.
→ Pour que tout se passe bien il faut que tous les **attributs de session** soient **sérialisables**.
→ Il ne vaut mieux plus utiliser de variables d'instances au niveau du servlet car il y aura plusieurs instances (pouvant éventuellement devenir incohérentes).

VII - DataSource JDBC , accès via JNDI

1. Sources de données JDBC

1.1. Api JDBC (Java DataBase Connectivity)



1.2. Pool de connexions et DataSource

Pool de connexions vers SGBDR

Rôles (utilités) des pools de connexions:

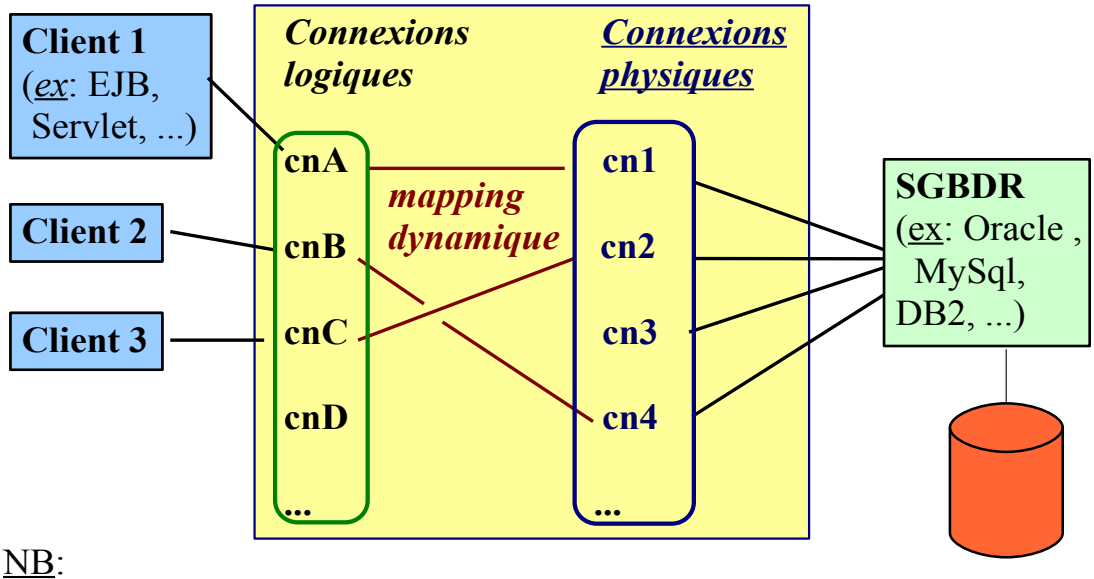
Recycler et ***partager*** *(par différentes attributions successives)* **un ensemble de connexions physiques vers un certain SGBDR**.

Ceci permet d'éviter les 2 écueils suivants:

- *Ouvrir, fermer et ré-ouvrir , ... des connexions vers le SGBDR* (opérations longues répétées ➔ mauvaises performances).

- *Utiliser simultanément une même connexion pour effectuer de multiples traitements* ➔ mauvaise gestion des concurrences d'accès et des transactions (joyeux mélanges)

Pool de connexions



NB:

Dès d'un client ferme une connexion logique , la connexion physique associée est considérée comme libre et peut alors être recyclée de façon à ce qu'un autre client puisse obtenir une nouvelle connexion logique.

Remarque: Etant donné qu'une connexion libérée (via close) par un composant n'est pas vraiment fermée mais peut être tout de suite réutilisée par un autre composant, chaque traitement (à l'intérieur d'une méthode d'un composant) doit:

- demander une connexion disponible dans le pool
- l'utiliser brièvement
- rapidement la libérer

Vue du pool par le client java - DataSource

Un client java voit un pool de connexions JDBC comme un objet de type *javax.sql.**DataSource***.
L'accès à cette source de données découle d'une **recherche JNDI** à partir d'un nom convenu (à paramétrer):

```
InitialContext ic = new InitialContext();
String dsName="java:comp/env/jdbc/dsBaseX"
DataSource ds = (DataSource) ic.lookup(dsName);
```

L'objet *DataSource* permet alors de **récupérer de nouvelles connexions logiques**:

```
Connection cn =  ds.getConnection();

// ... utilisation classique d'une connexion JDBC  ...

cn.close(); // fermeture de la connexion logique
```

2. Ressources générales accessibles via JNDI

TomCat offre un InitialContext **JNDI** (configurable dans /conf/server.xml).
A partir de ce service, du code java au sein d'une application web pourra obtenir une ressource (objet java automatiquement instancié et initialisé) en effectuant une recherche depuis un nom logique. Ce nom logique (que l'on passe en paramètre de la méthode **lookup**) est appelé **référence de ressource.**

Toute référence de ressource doit absolument être déclarée dans le fichier **web.xml** d'une application web par le biais de la balise <**resource-ref**> (ou bien **<env-entry>** pour une simple ressource de type valeur élémentaire de paramétrage: Integer, String) .

```
<resource-ref>  <!-- dans web.xml -->
  <description>ref vers le "datasource" offert par le serveur</description>
  <res-ref-name>jdbc/myDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Remarque: une référence de ressource (placée dans *WEB-INF\web.xml)* doit pointer vers une véritable ressource du serveur d'application (à paramétrer dans *conf\server.xml ou un ailleurs équivalent).*

2.1. Obtention d'une connexion (depuis du code en Java):

```
import  javax.naming.*; // JNDI
import  java.sql.*; // API JDBC standard (J2SE).
import  javax.sql.*;

private String dbName = "java:comp/env/jdbc/myDB";
                        // Nom logique JNDI

// Obtention via JNDI de l'objet DataSource:
InitialContext ic = new InitialContext();
DataSource ds = (DataSource) ic.lookup(dbName);

// Récupération de la connexion:
Connection cn =  ds.getConnection();

// ... utilisation classique d'une connexion JDBC  ...

cn.close(); // fermeture (virtuelle) de la connexion
            //celle ci est libérée et replacée dans le pool.
```

2.2. Configuration du pool de connexion

Remarque importante:

Derrière l'interface **DataSource** il peut se cacher **3 grands types de gestion des connections** :

- Le mode **basique**: pas de pool , simple connexion ordinaire.
- Le mode **pool**: véritable pool géré par un service du serveur d'application.
- Le mode **pool avec transactions distribuées** (protocole XA exigé au niveau du driver JDBC pour pouvoir gérer le commit à 2 phases).

➔ Les fonctionnalités réellement disponibles dépendent de toutes ces choses (à bien configurer):

- Driver JDBC compatible avec le pool de connexion.
- Service de Pool disponible? interchangeable? ….
- Le mode XA est-il pris en charge par le service de Pool et par le driver JDBC ?

* Rappel: le besoin du pool de connexion doit être déclaré en tant que **référence de ressource disponible** dans **web.xml**:

```
… <!-- après <servlet/> et après <servlet-mapping/> -->
<resource-ref>
  <description>reference vers le pool de cnx nécessaire</description>
  <res-ref-name>jdbc/myDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
…
```

```
Context initCtx = new InitialContext();
DataSource ds = (DataSource)  initCtx.lookup("java:comp/env/"+"jdbc/myDB");
```

Configuration d'un pool de connexion avec Tomcat 5.5 , 6 ou 7

* Le **Driver JDBC** adéquat doit être installé sous forme de **.jar** dans le répertoire **CATALINA_HOME/lib** (il sera ainsi disponible depuis Tomcat et les applications «web»).

- Les **paramétrages** du **pool de connexion** doivent être renseignés sous la balise <Context> de l'application web adéquate (ou bien sous la balise <DefaultContext> de <Host> ou <Engine>) dans le fichier `conf/`**`server.xml`** ou **`conf/Catalina/localhost/xxx.xml`** :

```
<Context ...>
  ...
  <Resource name="jdbc/myDB" auth="Container"
            type="javax.sql.DataSource"
            username="mydbuser"
            password="mypwd"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://localhost/myDB"/>
</Context>
```

NB: dans le cas particulier où tomcat est lancé par eclipse (en phase de test/développement), la configuration de la ressource doit être effectuée en fin du fichier **server.xml** situé dans le **projet "Server"** comportant la configuration de tomcat intégrée au workspace eclipse.

VIII - TagLib (balises pour pages Jsp)

1. TagLib (JSP) et JSTL

TagLib (pour pages JSP)

- **Bibliothèque** de **balises spécifiques** *codées sous forme de classes java* puis **placées au niveau des pages JSP** et **interprétées coté serveur**.
- ***permet de simplifier la syntaxe des pages JSP***
- ***permet d'obtenir une syntaxe plus homogène*** *(moins de mélange <% java %> , html) .*
- **JSTL** : Jsp Standard Tag Library
- *Autres TagLibs classiques:*
  - *celles de struts (ex: <bean:write /> , ...)*
  - *celles de JSF (ex: <h:inputText .../> , ...)*

Exemple (sans taglib):

```
<% java.util.Iterator it = listeProduits.iterator();
    while (it.hasNext()) {
     Produit prod = (Produit) it.next();
      %>
    <i> <%=prod.getLabel()%> </i> ,
    <b> <%=prod.getPrix()%> </b><br/>
<%}%>
```

Exemple équivalent (avec **<c:forEach>** de **JSTL**) :

```
<c:forEach var="prod" items="${listeProduits}" >
    <i> <c:out value="${prod.label}" /> </i> ,
    <b> <c:out value="${prod.prix}" /> </b><br/>
</c:forEach>
```

2. Présentation des "Tag Library"

Objectif: de façon à **clairement séparer** les aspects **"présentation (X)HTML"** et **"traitements Java"** , on peut **inventer de nouvelles balises (Tag)** qui seront **associées à des traitements java spécifiques**.
Ainsi le concepteur d'écran peut utiliser des produits sophistiqués tels que DreamWeaver pour mettre en forme la page JSP sans être embêté par des obscurs blocs en <% … %>.
Ces balises personnalisées sont disponibles depuis la version 1.1 de JSP.

2.1. Anatomie d'une balise:

```
<prefixe:nomBalise     attribut="valeur">
        <balise_imbriquee>  … </balise_imbriquee>
        contenu du corps (texte)
</prefixe:nomBalise>
```

3. Mise en place et utilisation d'une *TagLib*

3.1. Utilisation de balises personnalisées issues d'une tagLib:

MaPage.jsp

```
<%@ taglib  uri="http://www.xxx.com/yyy/zzz
                  ou  /WEB-INF/tlds/myLib.tld" prefix="p" %>
<html><body>
    Ligne1 </br> <p:welcome  name="toto" > </p:welcome>
</body></html>
```

- Si uri vaut simplement **"myLib"** (ou http://.....) et s'il existe une balise **<uri>** avec la même valeur dans un fichier ".tld" du répertoire **META-INF** situé dans le **".jar"** du paquet de balises (lui-même placé dans WEB-INF/lib), il y a alors une correspondance trouvée pour identifier le paquet de balises. Sinon, la balise ***<jsp-config>/<taglib>*** du fichier **WEB-INF/web.xml** peut éventuellement être utilisée pour indiquer le chemin menant au fichier **.tld** .
- Si uri vaut "**/WEB-INF/tlds/myLib.tld**" le chemin est alors codé en dur dans le jsp.

3.2. Fichier xml de configuration --- TagLib Descriptor (.tld) :

WEB-INF/tlds/myLib.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
    "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>1.2</jsp-version>
 <uti>http://www.xxx.com/yyy/zzz</uri>
```

```
<short-name>p</short-name>


 <tag>
       <name> welcome </name>
       <tagclass> package_qui_va_bien.WelcomeTag </tagclass>
       <bodycontent>JSP</bodycontent>
       <info> blabla sur balise. inutile de baliser ! </info>
       <attribute>
              <name> name </name>
              <required> true </required>
              <rtexprvalue> true </rtexprvalue>
      </attribute>
      <!-- + descriptions des autres attributs -->
 </tag>
 <!-- + descriptions des autres tags -->
</taglib>
```

NB: Une des annexes montre comment programmer une nouvelle balise sous forme de classe Java.

4. Utilisations courantes des TagLib

4.1. Suppression de la plupart des scriptlets

La principal avantage des TagLib réside dans le fait qu'un concepteur d'interface graphique web (designer de pages HTML, ...) n'a quasiment plus de code java à saisir dans une page jsp. Tous les traitements java se retouvent cachés derrières de nouvelles basiles dont la syntaxe est vraiment naturelle (plus de mélange entre le code html et le code java entre <%%>).

4.2. Ajout de nouvelles fonctionnalités de haut niveau:

Grâce à une bibliothèque sophistiquée de balises (ex: STRUTS ou JSTL) , on peut déclencher très simplement des fonctionnalités avancées telles que:

- Boucler automatiquement sur une collection d'éléments
- Envoyer un mail
- Gérer l' upload file
- Gérer l'internationalisation
- ...

4.3. Génération automatique de code javascript

L'incorporation de code javascript dans une page html est bien souvent problématique au sujet des points suivants:

- syntaxe quelquefois lourde (beaucoup de lignes).
- langage beaucoup moins rigoureux que java ==> bugs difficiles à localiser.
- interprétation quelquefois dépendante du navigateur (IE3, IE4, IE5, IE6, NS4, NS6,

Opera, ...).
- beaucoup de tests sont nécessaires pour s'adapter au navigateur (affichage dégradé mais affichage quand même et sans erreur !!!).

Malgré tous ses défauts, javascript offre les fonctionnalités non négligeables suivantes:
- très bon temps de réponse (car directement interprété coté client (navigateur)).
- interactivité sympathique.
- contrôle de saisie de premier niveau (coté client) permettant de soulager le réseau et le serveur web.
- Effets graphiques remarquables via le DHTML (DOM niv1 depuis IE 5.5 ou NS6).

==>

- La génération automatique de la partie "javascript" d'une page HTML à générer dynamiquement peut donc être très intéressante au niveau des pages JSP.
- Une bibliothèque de balises spécialisée dans la génération de code Javascript/Dhtml est pour cela presque indispensable.

4.4. Nécessité d'une certaine standardisation

Les paragraphes précédents ont clairement montrer qu'une bibliothèque de balises pour pages jsp est:
- très simple à utiliser.
- relativement complexe à programmer.

Autre écueil potentiel : Si chaque programmeur invente ses propres nouvelles balises, les pages jsp seront ainsi de plus en plus spécifiques et de moins en moins "standard".

Il faut donc absolument:
- s'appuyer sur des **standards (JSTL , STRUTS , ...)**.
- choisir (ou pas) d'utiliser une bibliothèque de balises au niveau d'un projet.
- bien documenter les choses.

IX - JSTL (1.2)

1. Bibliothèque standard "JSTL"

1.1. Présentation

JSTL (JSP Standard Tag Library)

Funtional Area	URI	Prefix	Example
Core	**http://java.sun.com/jsp/jstl/core**	**c**	**<c:*tagname* ...>**
XML processing	http://java.sun.com/jsp/jstl/xml	x	<x:*tagname* ...>
I18N capable formatting	**http://java.sun.com/jsp/jstl/fmt**	**fmt**	**<fmt:*tagname* ...>**
Database access (SQL)	http://java.sun.com/jsp/jstl/sql	sql	<sql:*tagname* ...>

Nb: Les balises de JSTL sont paramétrées via des attributs dont les valeurs sont généralement exprimées via un langage spécial ***(EL : Expression Language)*** fortement inspiré de javascript (EcmaScript).

```
a. String literal     att="15"
b. rtexprvalue        att="<%= foo.getBar() %>"
c. elexprvalue        att="${foo.bar}"
```

==> **objectif** : écrire des pages jsp dont la **syntaxe ressemble** à celle d'un langage de script (**javascript, feuille de style xslt**, ...).

1.2. Téléchargement & installation

depuis http://jcp.org/aboutJava/communityprocess/final/**jsr052**/index.html
http://java.sun.com/products/jsp/jstl/
et ***http://jakarta.apache.org/builds/jakarta-taglibs/releases/standard/***

==> récupérer **jstl1.1.2.jar** ou **jstl1.2.jar**
et **standard1.1.2.jar**

et placer ces fichiers dans WEB-INF/lib .

Au sein du fichier d'implémentation standard de JSTL "***standard1.1.2.jar***" , on trouve un répertoire **META-INF** avec tous les fichiers ***".tld"*** nécessaires à l'utilisation de JSTL.
Dans ces fichiers ".tld" , les uri sont prédéfinis :

- **http://java.sun.com/jsp/jstl/core**
- **http://java.sun.com/jsp/jstl/fmt**
- **...**

Ces URI peuvent directement être utilisées dans les pages JSP de l'application sans nécessiter de

configuration dans web.xml .

1.3. Utilisation

Ceci étant fait, une page jsp peut introduire une référence véritablement standard vers une des bibliothèques de JSTL:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
  ... reste de la page jsp en utilisant des "expression value"
    pour les valeurs des attributs  ...
   <c:jstlTag  att="${el}" />
```

ou bien

```
<%@ taglib uri="http://java.sun.com/jstl/jsp/core-rt" prefix="c-rt" %>
  ... reste de la page jsp en utilisant des "return value"
    pour les valeurs des attributs  ...
   <c:jstlTag  att="<%= ... %>" />
```

1.4. Expression language (EL)

valeur (résultat) d'une expression:

<c:out value="**${***el_expr}*" **/>** est a peu près équivalent à <**%=** *java_expr* **%>**

```
Hello <c:cout value="${requestScope.name}"/>
      est par exemple équivalent à :
Hello <%=pageContext.getAttribute("name", PageContext.SCOPE_REQUEST)%>
      et à
Hello <%=request.getAttribute("name")%>

Syntaxe EL générale: ${attributeName}
```

valeur par défaut:

```
<c:out value="${customer.address.city}" default="unknown"/>
```

1.5. préfixes pour le choix de la portée (scope):

${**pageScope.**xxxBean}
${**requestScope.**yyyBean}
${**sessionScope.**zzzBean}
${**applicationScope.**xyzBean}

et

```
<c:out value="${requestScope.yyyBean.p1}" />
```

est donc équivalent à:

```
<jsp:useBean id="yyyBean" scope="request" class="p.Cx" />
<jsp:getProperty name="yyyBean" property="p1" />
```

1.6. Tags conditionnels:

```
<c:if test="${user.visitCount == 1}">
     This is your first visit. Welcome to the site!
 </c:if>

<c:choose>
    <c:when test="${verbosityLevel == 'short'}">
       <c:out value="${product.shortDescription}"/>
    </c:when>
    <c:when test="${verbosityLevel == 'medium'}">
       <c:out value="${product.mediumDescription}"/>
    </c:when>
    <c:otherwise>
       <c:out value="${product.longDescription}"/>
    </c:otherwise>
 </c:choose>
```

1.7. Tag pour itérations (sur collection, ...)

```
<table>
    <c:forEach var="customer" items="${customers}">
      <tr><td><c:out value="${customer}"/></td></tr>
    </c:forEach>
 </table>
```

L'attribut **items** peut référencer l'une des choses suivantes:

- une instance de Collection (Vector, ...)
- un tableau d'Object.
- un instance de java.sql.ResultSet et dans ce cas l'élément courant pointe sur le ResulSet lui même. Celui-ci étant positionné sur la ligne courante.
- une chaîne du genre "valeur1,valeur2,valeur3"
- une instance de Iterator issue d'une collection, ...

Dans le cas où l'on boucle sur une **Map**, chaque élément comporte les parties **.key** et **.value**

```
<c:forEach var="entry" items="${myHashtable}">
    Next element is <c:out value="${entry.value}/>
</c:forEach>
```

récupération de l'indice courant durant la boucle:

```
<table>
    <c:forEach var="employee" items="${employees}" varStatus="status">
      <tr>
        <td><c:out value="${status.index}"/></td>
        <td><c:out value="${employee.name}"/></td>
      </tr>
    </c:forEach>
</table>
```

boucle for (pour i allant de n à m):

```
<c:forEach var="i" begin="0" end="110">
   <c:out value="${i}"/>
</c:forEach>
```

1.8. Tags pour l'internationalisation (in-18):

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```

Obtention de libellés en fonction de la langue de l'utilisateur:

Choix d'un certain bundle (dont le chemin de la classe est identifié par basename):

```
<fmt:setBundle basename="mypackage.MyResources" />
```

ou bien

*Utilisation d'un bundle par défaut paramétré dans **web.xml**:*

```
<web-app>
      <context-param>
        <param-name>javax.servlet.jsp.jstl.i18n.basename</param-name>
        <param-value>mypackage.MyResources</param-value>
      </context-param>
</web-app>
```

puis

```
<fmt:message key="welcome"/>
```

*Formatage avec paramétrage à la manière de java.text.**MessageFormat** (via {0} , {1} , ...):*

```
<fmt:message key="welcome_withname"/>
  <fmt:messageArg value="${nameArg}"/>
<fmt:message>
```

Nb: **mypackage.MyResource[_fr,_de,_es,...].properties** doit être trouvé dans le classpath (initialement placé dans src).

Ex:

```
welcome=bonjour
welcome_withname=bonjour {0}
```

Mise en forme des dates:

```
<fmt:formatDate timeStyle="long" dateStyle="long" value="${objDate}"/>
```

va générer

October 22, 2001 4:05:53 PM PDT aux U.S.

et *22 octobre 2001 16:05:53 GMT-07:0* en France.

Inversement :

```
<c:set value="May 22, 2001 4:05:53 PM PDT" var="chDate" />
<fmt:parseDate value="chDate"  var="parsedDate"/>
```

Mise en forme des nombres:

```
<fmt:formatNumber value="12.3" pattern=".000"/>
```

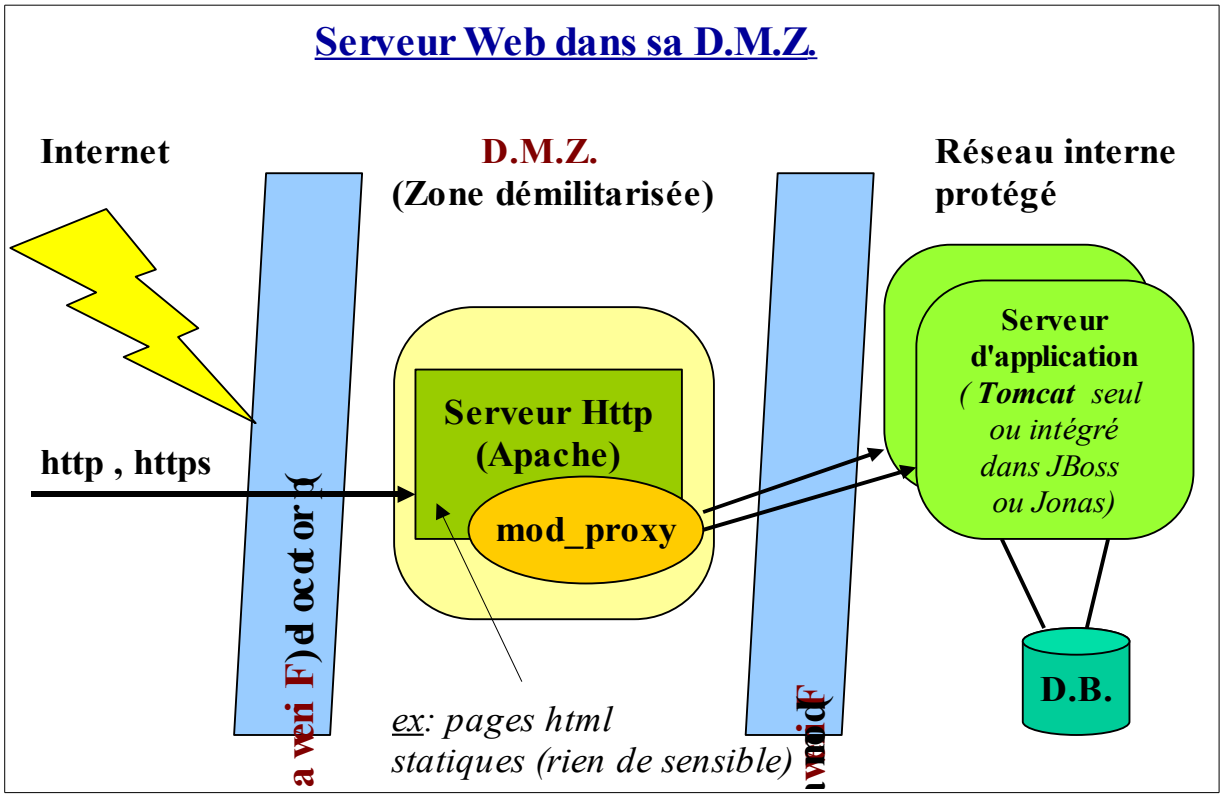
va générer *12.300* .

Inversement :

```
<fmt:parseNumber value="${chNum}" var="numParsed"/>
```

X - Sécurité JEE/Web (rôles, ...)

1. D.M.Z. et Firewalls



2. Sécurité J2EE/JEE5

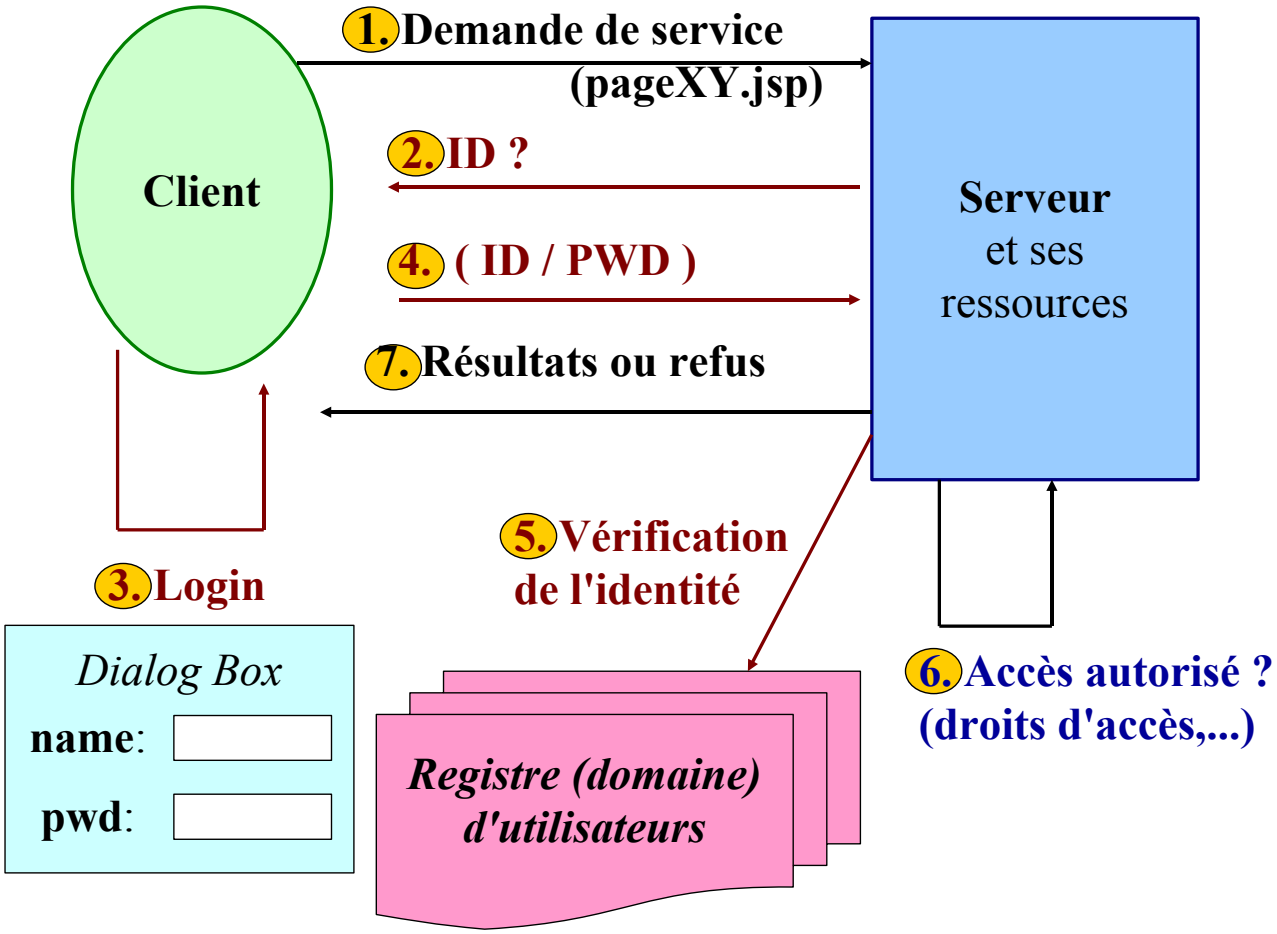
Sécurité J2EE/JEE5

Gérer la sécurité "J2EE/JEE5" consiste essentiellement à :

- **Préciser le ou les rôle(s) logique(s) requis pour pouvoir déclencher une certaine méthode sur un EJB ou pour activer certaines URL d'une application Web.**
  ➔ *Travail généralement effectué par le développeur*

- **Authentifier l'utilisateur** (UserName , PassWord).
  ➔ *Via technologie **HTTPS** ou **JAAS** ou ...*

- **Associer un ou plusieurs utilisateur(s) ou groupe(s) [d'un annuaire ldap ou ...] à chaque rôle logique**.
  ➔ *Paramétrage et configuration liés au déploiement d'une application J2EE et à l'administration du serveur*


Client
1. Demande de service
(pageXY.jsp)
2. ID ?
4. (ID / PWD)
7. Résultats ou refus
Serveur
et ses
ressources
3. Login
Dialog Box
name:
pwd:
5. Vérification
de l'identité
Registre (domaine)
d'utilisateurs
6. Accès autorisé ?
(droits d'accès,...)

3. Vue d'ensemble sur la gestion de la sécurité J2EE

3.1. Approche déclarative et «mapping» associés

La **gestion de la sécurité** est étroitement liée au déploiement car elle est **déclarative** (son paramétrage est effectuée par l'administrateur qui supervise les composants installés au niveau d'un serveur d'application).

Gérer la sécurité consiste essentiellement à

- **Authentifier l'utilisateur** (UserName , PassWord).
- Vérifier l'appartenance de l'utilisateur à un **groupe**.
- **Associer un ou plusieurs groupe(s) à un rôle**.
- **Préciser le ou les rôle(s) requis pour pouvoir déclencher certaines url vers des parties de l'appli web (Servlet , jsp, …)**

Exemple de configuration (à placer sous <web-app>) dans web.xml:

```
<security-constraint>
    <display-name>Example Security Constraint</display-name>

    <web-resource-collection>
      <web-resource-name>Protected Area</web-resource-name>
        <!-- Define the context-relative URL(s) to be protected -->
      <url-pattern>/pages/p1.jsp</url-pattern>
      <url-pattern>/pages/p2.jsp</url-pattern>
      <url-pattern>/pages/xy/*</url-pattern>
        <!-- If you list http methods, only those methods are protected -->
      <http-method>DELETE</http-method>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
     <http-method>PUT</http-method>
    </web-resource-collection>

    <auth-constraint>
      <!-- Anyone with one of the listed roles may access this area -->
      <role-name>employe</role-name>
      <role-name>role2</role-name>
    </auth-constraint>

</security-constraint>

<security-role>
    <description>employe de l'entreprise XYZ</description>
    <role-name>employe</role-name>
</security-role>
```

```
<security-constraint>
        <web-resource-collection>...</web-resource-collection>
        <auth-constraint>...</auth-constraint>
        <user-data-constraint>
            <transport-guarantee>CONFIDENTIAL</transport-guarantee>
        </user-data-constraint>
    </security-constraint>
```

Fixer le paramètre "transport-guarantee" à **CONFIDENTIAL** permet d'activer **SSL/HTTPS** dans l'authentification (si les échanges avec le serveur sont à sécurisés) . Ceci nécessite un paramétrage au niveau du serveur (Apache2 et/ou Tomcat) : certificats à mettre en place .

NONE	Aucun cryptage (données en clair)
CONFIDENTIAL	Seul le client connecté et authentifié peut lire les données (cryptées via SSL/HTTPS)
INTEGRAL	Assure en plus une intégrité des données véhiculées (elles ne peuvent pas être modifiée en cas d'interception).

3.2. Politiques d'authentification (gestion des comptes utilisateurs)

Un "**Realm**" (royaume/domaine) est une **collection d'utilisateurs qui sont contrôlés via une même politique d'authentification**.
Un «**Realm**» (domaine) peut être vu comme une **base de données d'utilisateurs** (avec mots de passe et rôles associés).

Tomcat 4.0 , **5.x , 6.x et 7.x** peut en standard gérer de 3 façons le domaine (Realm) :

- Via une **base de données** (*JDBCRealm*)
- Via un **serveur LDAP** (*JNDIRealm*)
- Via un simple **fichier XML** (*MemoryRealm*)

Tomcat offre en plus la possibilité de programmer soi même un accès à un « Realm » spécifique.

Realm de type «Fichier de conf en xml» pour Tomcat :

C'est le type de Realm utilisé par défaut avec Tomcat (balise <Realm> placé sous <Engine> de `$CATALINA_HOME/conf/`**`server.xml`**).

Le fichier de config s'appelle **`conf/tomcat-users.xml`** et a la structure suivante :

```
<tomcat-users>
  <user name="tomcat" password="tomcat" roles="role1" />
  <user name="toto"   password="xxx" roles="role1,role2" />
</tomcat-users>
```

3.3. Méthode d'authentification (Saisie username/pwd)

Dès q'un navigateur Web tente d'accéder à un composant Web (ex: Servlet ou page JSP) déclaré comme protégé, un des trois modes d'authentification sera déclenché:

- **BASIC**: le navigateur se charge de récupérer le UserName et le mot de passe.
- **FORM**: via un page .html ou .jsp de notre choix et comportant un formulaire pour saisir le nom et le mot de passe.
- **CERTIFICATE (coté client)**: permet de vérifier depuis quel ordinateur le client a émis la requête.

A titre d'exemple, le fichier ***web.xml*** peut comporter les entrées suivantes:

```
...
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>Domaine des utilisateurs XYZ</realm-name>
    <form-login-config>
      <form-login-page>/login.jsp</form-login-page>
      <form-error-page>/loginError.jsp</form-error-page>
    </form-login-config>
 </login-config>
```

NB:

- Avec le mode "**BASIC**" , l'information "realm-name" s'affiche automatiquement dans la boite de dialogue générée par le navigateur internet (IE, FX, ...) .
- Le mode "**FORM**" permet de mieux personnalisé la page d'authentification.
- Le mode "**FORM**" (conseillé) gère mieux les fin de session.

login.jsp (exemple pour le mode "FORM")

```
<html> <head><title>Login Page for Examples</title> </head>
<body bgcolor="white">
<form method="POST"
        action='<%=response.encodeURL("j_security_check") %>' >
  <table border="0" cellspacing="5">
    <tr> <th align="right">Username:</th>
      <td align="left"><input type="text" name="j_username"></td></tr>
    <tr>  <th align="right">Password:</th>
      <td align="left"><input type="password" name="j_password"></td> </tr>
    <tr>  <td align="right"><input type="submit" value="Log In"></td>
      <td align="left"><input type="reset"></td> </tr>
  </table>
</form></body></html>
```

loginError.jsp (exemple pour le mode "FORM")

```
<html> <head> <title>Error Page For Examples</title> </head>
<body bgcolor="white">
  <font color='red'>Invalid username and/or password</font>  <hr/>
  <form> <input value="try again"  type="button"  onclick="history.back();"/></form>
</body></html>
```

XI - Filtres & Listeners

1. Filtres (depuis l'api servlet 2.3 et Tomcat 4)

1.1. Notion de filtre

Un filtre est un élément supplémentaire qui s'insère en tant qu'enveloppe au niveau de la chaîne d'exécution des servlets et qui peut modifier la requête et/ou la réponse Http.

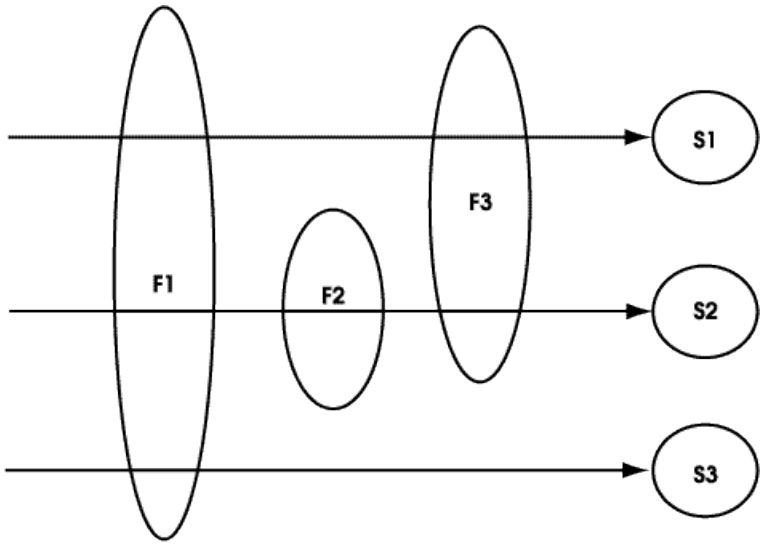
Ceci peut être utile pour effectuer l'une des tâches suivantes:

- Effectuer des conversions (images, données, ...)
- Gérer le cryptage des données
- Effectuer des transformations XSLT.
- Effectuer des compression & décompression (gzip)
- Rajouter automatiquement des entêtes , des compteurs , des stats, ...
- ...

1.2. Insertion d'un filtre (Filter Mapping de web.xml):

Un filtre n'est jamais directement mentionné dans une url , il est simplement associé à certaines url en tant qu'élément supplémentaire (s'activant avant en tant qu'enveloppe) :

Nb: l'ordre des filtres correpond à l'ordre des balises <filter-mapping> de web.xml.




```
<web-app>
<filter>
    <filter-name>XSLTFilter</filter-name>
    <filter-class>XSLTFilter</filter-class>
  </filter>
<filter>
    <filter-name>HitCounterFilter</filter-name>
    <filter-class>HitCounterFilter</filter-class>
</filter>
```

```
  <filter-mapping>
    <filter-name>HitCounterFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping>
    <filter-name>XSLTFilter</filter-name>
    <servlet-name>FilteredFileServlet</servlet-name>
  </filter-mapping>
 <servlet> ... </servlet>
 <servlet-mapping> .... </servlet-mapping>
...
<web-app>
```

1.3. Programmation d'une classe de Filtre:

```
public final class XXXFilter implements Filter
{
   private FilterConfig filterConfig = null;

   public void init(FilterConfig filterConfig)
      throws ServletException { this.filterConfig = filterConfig; }

   public void destroy() { this.filterConfig = null; }

   public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
      throws IOException, ServletException
        {
           ... code du filtre ....
        chain.doFilter(request,response); // déclenche l'élément
                          // suivant (autre filtre ou servlet , ...)
        }
}
```

De façon à pouvoir modifier la réponse générée par le code du servlet (élément suivant de la chaine) , il faut invoquer **chain.doFilter(request,wrapper);** où **wrapper** est une enveloppe autour de response dont la méthode **getWriter**() retourne un flux sur un paquet d'octets qui ne sera pas directement renvoyé mais qui sera retraité par la fin du filtre actuel:

```
public class CharRespWrapper extends   HttpServletResponseWrapper
 {
  private CharArrayWriter output;
  public String toString() {  return output.toString(); }
  public CharResponseWrapper(HttpServletResponse response)
            { super(response); output = new CharArrayWriter(); }
 public PrintWriter getWriter() {return new PrintWriter(output);}
}
```

```
...
PrintWriter out = response.getWriter();
CharRespWrapper wrapper = new
CharRespWrapper( (HttpServletResponse)response);
chain.doFilter(request, wrapper);
out.write("...." + wrapper.toString() + "...");
```

2. Listener (code activé au chargement/... d'une application)

2.1. Listeners de niveau "application"

Certains ***événements liés à une application WEB*** (ou plus exactement à l'objet central "ServletContext") peuvent être gérés au sein d'objets appelés "**Listener**" .
Ceci permet essentiellement de ***déclencher automatiquement certains traitements*** aux moments des ***chargement/initialisation*** et ***arrêt/déchargement*** d'une ***application WEB***.

Interfaces événementielles (javax.servlet)	*Descriptions*
ServletContextListener	objet "application" (ServletContext) tout juste créé ou bien sur le point d'être supprimé.
ServletContextAttributeListener	Attribut ajouté,supprimé ou modifié sur l'objet application (ServletContext)

NB:

Une classe d'objet "***Listener***" doit ***implémenter l'interface événementielle adéquate*** et son code compilé doit être placé dans **WEB-INF/classes** ou bien dans un des **"...jar"** de **WEB_INF/lib**.

D'autre part, un "***Listener***" (gestionnaire d'événements) doit être déclaré au sein du fichier ***WEB-INF/web.xml*** pour qu'il soit pris en compte:

```
<web-app ...>
<display-name>MyListeningApplication</display-name>
<listener>
    <listener-class>mypackage.MyListenerClass</listenerclass>
</listener>
<listener>
      <listener-class>mypackage.MyOtherListenerClass</listener-class>
</listener>
<servlet>
...
</servlet>
...
</web-app>
```

Exemple:

```
package mypackage;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class MyListenerClass implements ServletContextListener {

        public void contextInitialized(ServletContextEvent e) {

      // initialisation au chargement/démarrage de l'application WEB
                ServletContext application = e.getServletContext();
                Integer objCompteur = new Integer(1);
                application.setAttribute("compteur",objCompteur);
        }

        public void contextDestroyed(ServletContextEvent e) {

      //  terminaison lors de l'arrêt de l'application WEB
                ServletContext application = e.getServletContext();
                Integer objCompteur = (Integer) application.getAttribute("compteur");
                System.out.println("compteur:" + objCompteur.intValue());
        }

}
```

2.2. Listeners de niveau "Session" et "Request"

Il existe également des "***Listener***" de niveau "***Session***" et "***Request***" .
Ces derniers sont à déclarés au sein de **WEB-INF/web.xml** .

Gestionnaires d'événements liés à une "*Session*" :

Interfaces événementielles (javax.servlet)	*Descriptions*
***.http.*HttpSessionListener**	objet "session" (HttpSession) tout juste créé ou bien sur le point d'être supprimé (invalidé ou timeout).
HttpSessionAttributeListener	Attribut ajouté,supprimé ou modifié sur l'objet session (HttpSession)
HttpSessionActivationListener	objet "session" (HttpSession) activé (en mémoire) ou passivé (sur disque)
HttpSessionBindingListener	object "bound" or "unbound" from HttpSession

exemple:

```
package mypackage;

import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class MySessionListener implements HttpSessionListener {

	public void sessionCreated(HttpSessionEvent e) {
		System.out.println("nouvelle session:" + e.getSession().getId());
	}

	public void sessionDestroyed(HttpSessionEvent e) {
		System.out.println("fin de session:" + e.getSession().getId());
	   // + éventuelle sauvegarde des valeurs dans base de données
	}
}
```

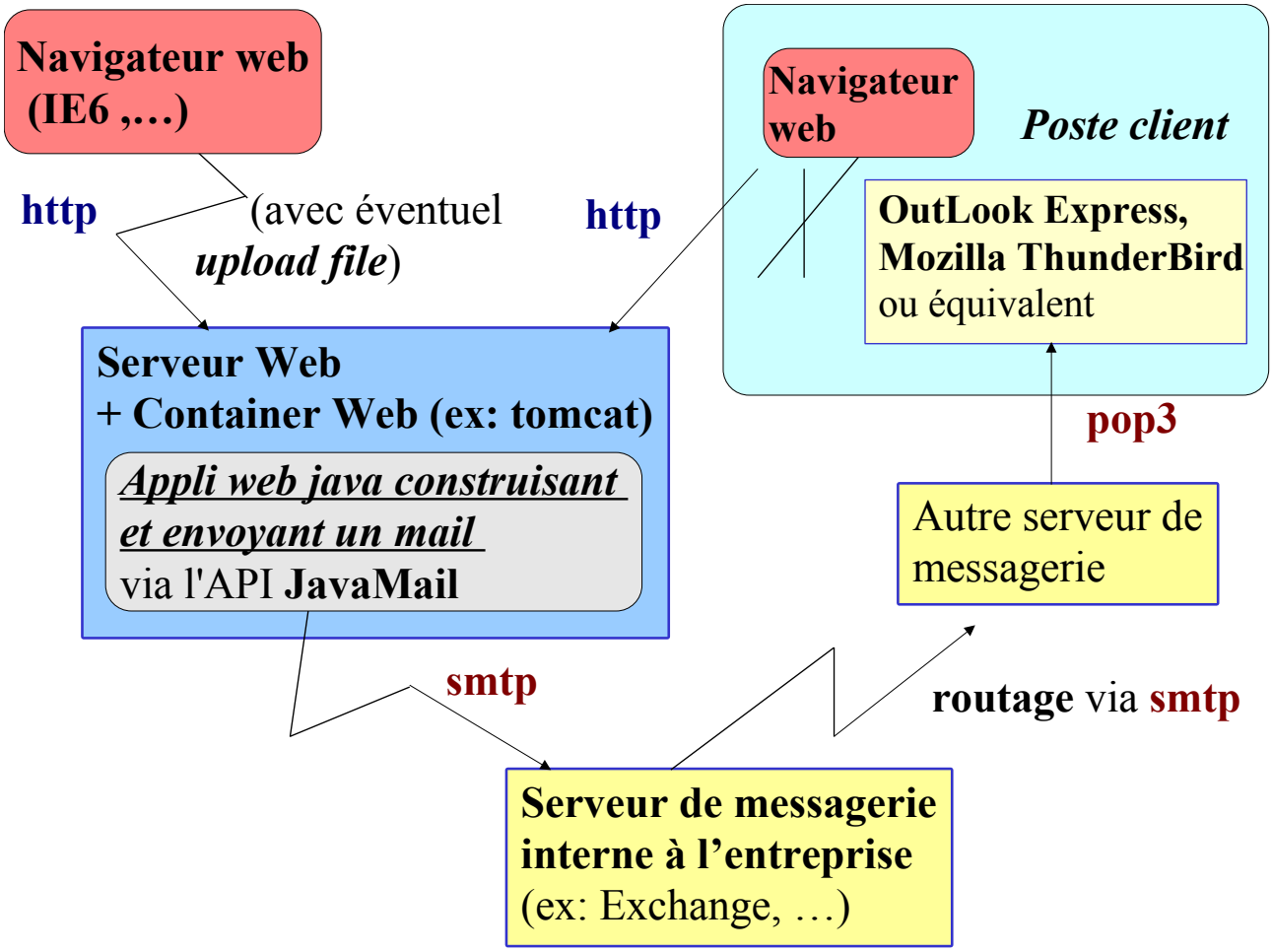
Gestionnaires d'événements liés à une "*Request*":

Interfaces événementielles (javax.servlet)	*Descriptions*
ServletRequestListener	objet "requête" (ServletRequest) initialisé.
ServletRequestAttributeListener	Attribut ajouté,supprimé ou modifié sur l'objet requête (ServletRequest)

XII - JavaMail (présentation de l'API)

1. L'api javax.mail

Pour envoyer (ou consulter) un mail depuis un programme java, on peut utiliser l'api standard **javax.mail** (à télécharger via l'url ***http://java.sun.com/products/javamail***).



Exemple :Séquence d'envoi d'un mail (sans pièce jointe)

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
```

```
String host = "localhost"; //"mydomain.com"
String from = "didier@mydomain.com";
String to = "destinataire@xxxyyyy.com";
Properties props = System.getProperties();
props.put("mail.smtp.host", host);
Session session = Session.getDefaultInstance(props, null);

MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient( Message.RecipientType.TO,   new InternetAddress(to));
```

```
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

Transport.send(message);
```

Intérêts de la messagerie / Utilisations possibles

Par rapport au schéma client / serveur classique , la messagerie se distingue en apportant la fonctionnalité fondamentale suivante: **le mode asynchrone**.

On peut en effet envoyer un message vers un destinataire qui est pour l'instant indisponible . Celui-ci pourra ultérieurement consulter tranquillement son courrier lorsqu'il aura un moment de libre.
Autres notions connexes: **Diffusion**, **GroupWare** et **Workflow**.

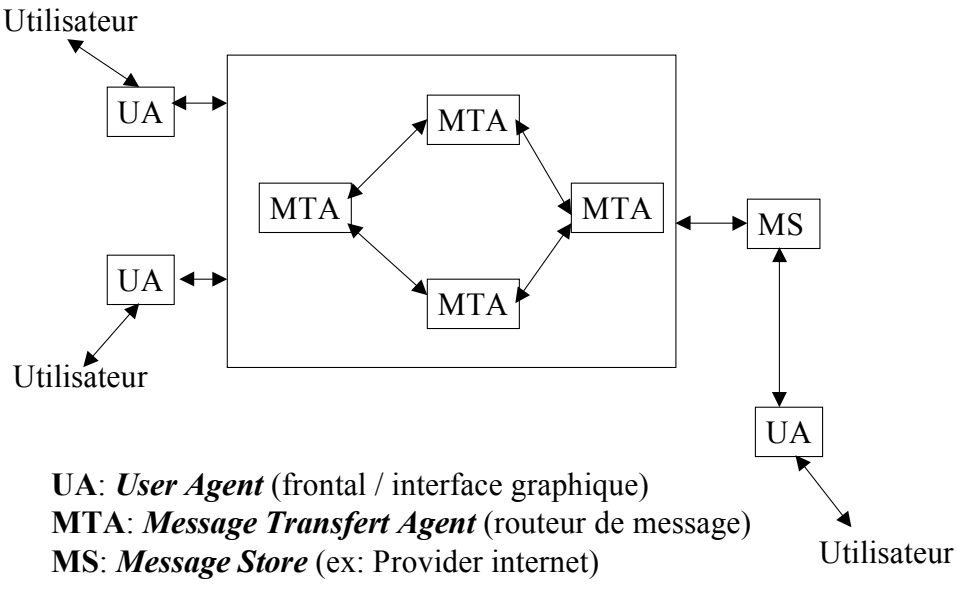
NB: Les principaux produits liés à le messagerie sont aujourd'hui les suivants:

- **Microsoft Exchange et OutLook Express**
- **Lotus Notes / Domino**
- **Eudora World Mail,** Mozilla **ThunderBird** , Apache/James , ...

2. Présentation des différents protocoles

2.1. Terminologie générale

Messagerie



UA: ***User Agent*** (frontal / interface graphique)
MTA: ***Message Transfert Agent*** (routeur de message)
MS: ***Message Store*** (ex: Provider internet)

2.2. SMTP

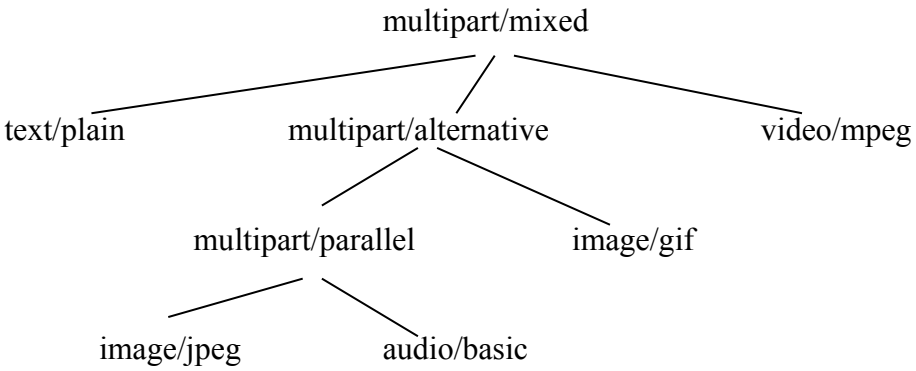
SMTP (***Simple Mail Transfert Protocol***) est le protocole utilisé par la **messagerie internet.** Ce **protocole** sert à **envoyer et router les messages.**
Une adresse de boite aux lettres internet est du type NomPersonne@NomSite.Pays (exemple didier-

defrance@wanadoo.fr) .

2.3. MIME

MIME (***Multipurpose Internet Mail Extenstions***) normalise (depuis 1993) les types et les sous types de fichiers échangés dans les messages SMTP.

Exemple:



2.4. POP3

Le protocole **POP3** du monde internet **permet d'accéder à distance à une boîte aux lettres** (qui se trouve généralement chez le provider internet).
Par exemple le programme **OutLook** utilise **POP3** pour obtenir les nouveaux messages.

POP3 signifie **Post Office Protocol** version 3

Variante du protocole : **IMAP4 (Internet Message Access Protocol)** est un **protocole plus élaboré** que POP3 que l'on peut quelquefois utiliser (si dispo).

3. L'api javax.mail

Pour envoyer (ou consulter) un mail depuis un programme java, on peut utiliser l'api standard **javax.mail** (à télécharger via l'url ***http://java.sun.com/products/javamail***).

Attention, l'api **javax.mail** doit être utilisée conjointement avec deux autres api :

- Java **Activation** Framework (javax.activation)
- **Provider POP3**

Il faudra donc installer et utiliser les 3 fichiers "jar" suivants:
mail.jar , activation.jar , pop3.jar

NB:

- Il faudra penser à renseigner la propriété **mail.host** de façon à préciser le serveur SMTP.
- Dans le cadre d'un simple développement il peut être intéressant d'utiliser un Serveur de mail simple à installer et à configurer. Le site sourceforge.net comporte quelques projets "OpenSource" intéressants (mailserver12.zip = Serveur de Mail (SMTP et POP3 programmé

en Java)).

Principales classes de l'api **JavaMail** (package ***javax.mail***):

Session	**Session de mail** (envoi , consultation) **liée à un certain utilisateur** (cf Authenticator)
Message (classe abstraite) javax.mail.internet.**MimeMessage**	**Message à envoyer** (généralement constitué de différentes parties "MIME" (cf *interface* **Part**))
Address (classe abstraite) javax.mail.internet.**InternetAddress**	**Adresse e-mail** classique (ex:xxx@yyy.fr)
Message.***RecipientType.TO*** Message.***RecipientType.CC*** Message.***RecipientType.BCC***	**Types de destinataire** (remarque: bcc = copie cachée)
Transport	Représente le protocole de transport (généralement SMTP), permet d'envoyer le mail
Store , **Folder**	**Pour lire des messages** (**Store** est généralement associé à **POP3**) et **Folder** à "**INBOX**"
BodyPart (classe abstraite) javax.mail.internet.**MimeBodyPart**	Partie d'un message
Multipart (classe abstraite) javax.mail.internet.**MimeMultipart**	Composition de différentes parties
javax.activation.**DataHandler**	Paquet de données disponible selon différents formats (MIME) ou sources (Presse-papier, ...)
javax.activation.**DataSource** (interface) javax.activation.**FileDataSource** javax.activation.**UrlDataSource**	Source de données (avec type et nom) à partir de laquelle on peut récupérer un flux en lecture (InputStream) ou en écriture.

3.1. Séquence d'envoi d'un mail (sans pièce jointe)

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
```

```
String host = "localhost"; //"mydomain.com"
String from = "didier@mydomain.com";
String to = "toto@mydomain.com";

// Get system properties
Properties props = System.getProperties();
// Setup mail server
props.put("mail.smtp.host", host);
// Get session
Session session = Session.getDefaultInstance(props, null);

// Define message
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient( Message.RecipientType.TO,
  new InternetAddress(to));
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

// Send message
```

```
Transport.send(message);
```

4. Obtention d'une session de mail via JNDI

La plupart des serveurs compatibles J2EE supportent la recherche via JNDI des sessions de mail. Ceci offre l'immense avantage de pouvoir fixer (hors du code java) les paramétrages spécifiques (et quelquefois sensibles) permettant d'accéder à un serveur de messagerie.

Une référence vers cette sorte de ressource se déclare de la façon suivante dans le fichier WEB-INF**web.xml** :

```
                                                                    ...
<resource-ref>
  <description>
   Session session = (Session) ic.lookup("java:comp/env/mail/SessionXY");
   ...
  </description>
  <res-ref-name>mail/SessionXY</res-ref-name>
  <res-type>javax.mail.Session</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
...
```

La ressource "Session de mail" doit ensuite être paramétrée au niveau du serveur.
Tomcat utilise pour ceci la syntaxe suivante:

```
<Context ...>
   ...
   <Resource name="mail/SessionXY" auth="Container"
             type="javax.mail.Session"
             mail.smtp.host="localhost"/>

   ...
</Context>
```

ANNEXES

XIII - Annexe – programmation taglib

1. Différentes versions pour les "TagLib"

1.1. Avant-propos

Les TagLibs en version "jsp 1.2" (ex: bibliothèque "displayTag") ne sont pas très faciles à programmer car elles sont basées sur une logique événementielle (avec découpages très fins).

La version suivante "jsp 2.0" a permis de considérablement simplifier la programmation des Tag ordinaires (les plus simples).

La première partie de ce chapitre montre comment programmer des "Taglibs" en version 1.2 (Ceci peut éventuellement être encore utile en 2010 si l'on doit reprendre le code source d'un tag existant pour l'améliorer ou le personnalisé ou bien pour d'autres raisons spécifiques).

Par contre, si l'on doit construire de nouvelles balises à partir de "Zéro" , il vaut mieux se baser sur la version "JSP 2.0" lorsque c'est possible.

1.2. Tableaux des versions (pour étudier certaines compatibilités)

Version JSP/TabLig	*Version Java EE minimum*	*Version minimum de Tomcat*
"1.1"	"1.2.1"	Tomcat 3
"1.2"	>= "1.3"	Tomcat 4
"2.0"	**>= "1.4"**	**Tomcat 5 & 5.5 (pour Java4 et 5)**
"2.1"	**>= "5"** *(époque EJB3/JPA1)*	**Tomcat 6 (java 6 bien supporté , J5 ok)**
"2.2"	>= "6" (bientôt)	Tomcat 7 (java 6 mininum) (bientôt)

En règle générale, des choses programmées pour une version n-1 fonctionnent encore avec les versions n , n+1, n+2 .

Des TagLibs 1.2 fonctionnent donc encore dans Tomcat6 et le jdk 1.6 .

Il faut cependant effectuer une analyse assez poussée pour déterminer si certaines bibliothèques (xxx.jar) utilisées par différents frameworks (Struts , JSTL , ...) ne rentrent pas en conflit .

Par exemple une implémentation de JSTL en version 1.1.x nécessite au minimum JSP 2.0 .

Pour approfondir les différentes versions des "TagLibs" , l'URL suivante permet d'accéder à une vue claire et détaillée des Taglibs 1.1 , 1.2 et 2.0 .

http://adiguba.developpez.com/tutoriels/j2ee/jsp/taglib/

2. Programmation de "TagLib" personnalisés (jsp 1.2)

2.1. Classe Java implémentant la balise (gestionnaire de balise):

WEB-INF/classes/package_qui_va_bien/**WelcomeTag.java**

```
import javax.servlet.jsp.tagext.*;

public class WelcomeTag  extends TagSupport
{
private String  nameAttribute;

public void setName(String ch)   { nameAttribute = ch;  }

public String getName()   { return nameAttribute ;  }

 public int doStartTag() throws JspTagException
  {
   return EVAL_BODY_INCLUDE;  /*  !=  SKIP_BODY */
        // donne l'ordre d'évaluer récursivement le contenu/ (sous balises éventuelles).
  }


public int doEndTag() throws JspTagException
  {
 try{
 pageContext.getOut().write("Welcome " + nameAttribute );
    } catch(IOException ex) {  throw new JspTagException("Erreur"); }
  return EVAL_PAGE;  /* != SKIP_PAGE */
 }
}
```

NB: la méthode prédéfinie **getParent**() retourne une référence (éventuellement nulle) sur un objet de type **Tag** correspondant à la balise parente (si elle existe).

L'interface **Tag** (implémentée par la classe **TagSupport**) permet de gérer une balise qui n'a pas besoin d'accéder au contenu de son corps.

2.2. Gestion des variables qui seront utilisées par les scriptlets

Un gestionnaire de balise personnalisée peut souhaiter mettre à jour des variables qui seront plus tard utilisées par des scriptlets (ou bien des "EL" en JSP2) pour générer les affichages.
L'avantage d'une telle approche réside dans le fait que le nouveau tag personnalisé ne code pas en dur l'affichage . Il est donc plus simplement réutilisable.

Cette approche nécessite l'introduction d'une classe supplémentaire héritant de **T**ag**E**xtra**I**nfo. Cette classe java devra en outre être référencée dans le fichier de configuration .tld (sous balise <**tei**class> de <tag>).

```
public class VarWelcomeTagExtraInfo extends TagExtraInfo
{
  public VariableInfo[] getVariableInfo(TagData data)
    {
    return new VariableInfo[]
        {
          new VariableInfo("name","java.lang.String",true,VariableInfo.NESTED),
          new VariableInfo("varName2","java.lang.Integer",true,VariableInfo.AT_BEGIN)
        }
    }
}
```

Nb:

- Le 3ème paramètre du constructeur permet d'indiquer s'il faut au besoin déclarer une nouvelle variable (true) ou s'il faut se contenter de mettre à jour une variable existante.
- Le dernier paramètre indique la visibilité de la variable (NESTED ? entre fin et début de la nouvelle balise, AT_BEGIN ? depuis l'ouverture de la balise jusqu'à la fin de la page, AT_END ? depuis la fermeture de la balise.)

Finalement , le gestionnaire de balise pourra mettre à jour une de ces variables de la façon suivante:
pageContext.setAttribute("name",chNameValue); */* dans doStartTag() */*

Résultat (utilisation):

```
…
<p:WelcomeVar   nameAttr="toto">
        Hello, my name is <%=name%> !      ou   bien ${name} en JSP2 via EL
</p:WelcomeVar>
```

2.3. Gestion du contenu du corps de la balise

L'interface **BodyTag** (héritant de Tag et implémentée par la classe **BodyTagSupport**) permet en outre d'accéder au contenu du corps de la balise.

Une balise pouvant comporter plusieurs sous balises. Il peut y avoir une sous boucle sur les fonctions événementielles **doInitBody**() et **doAfterBody**() .

L'accès au contenu d'une sous balise se fait de la façon suivante:

```
BodyContent bodyContent = getBodyContent();
if ( bodyContent != null) {   …. }
```

2.3.a. itération sur le contenu de la balise

```
<% java.util.List  names= new java.util.LinkedList();
    names.add("Didier");
    names.add("Dupont");  %>
….
<s:iter_welcome names="<%=names%>" >
   <%=index%>. Bonjour <%=name%> !
</s:iter_welcome>
```

➔ **0. Bonjour Didier**
1. Bonjour Dupont

```
public class ItemWelcomeTag  extends  BodyTagSupport
{
private List names; // + get/set
private int index=0;
private StringBuffer output  = new StringBuffer(); // pour affichage total différé

private void setLoopVariables() // fonction utilitaire appelée avant chaque itération
   {
    pageContext.setAttribute("name",names.get(index).toString());
    pageContext.setAttribute("index",new Integer(index));
   }

public int doStartTag() throws JspTagException
{ if(names.size() > 0) { setLoopVariables();  return EVAL_BODY_BUFFERED; }
    else return SKIP_BODY;
}

public int doAfterBody() throws JspTagException
{
BodyContent bodyContent = getBodyContent();
    if(bodyContent != null)
   {
   //récupérer l'interprétation JSP du corps et l'ajouter dans le buffer output:
     output.append( bodyContent.getString() );
   try {
        bodyContent.clear(); // ne pas afficher partiellement pour chaque itération
       }
    catch(IOException ex) { throw new JspTagException(" I/O Error") ; }
   }

if((++index) <names.size())
   {
    setLoopVariables();
    return EVAL_BODY_BUFFERED; /* provoque une nouvelle itération , une nouvelle
interprétation (JSP) du contenu du corps et un nouvel appel à doAfterBody() */
   }
else return SKIP_BODY; /* fin de liste */
}

public int doEndTag() throws JspTagException
{
try {  BodyContent bodyContent = getBodyContent();
    if(bodyContent != null)
            bodyContent.getEnclosingWriter().write(output.toString());
    }
catch(IOException ex) { throw new JspTagException(" I/O Error") ; }
return EVAL_PAGE; /* pour la suite */
}
}
```

déclaration (dans TLD):

```
<tag>
        <name>iter_welcome</name>
        <tag-class>tp.tags.v12.ItemWelcomeTag</tag-class>
        <body-content>JSP</body-content>
        <variable>
                  <name-given>name</name-given>
                  <variable-class>java.lang.String</variable-class>
                  <declare>true</declare>
                  <scope>NESTED</scope>
            </variable>
            <variable>
                  <name-given>index</name-given>
                  <variable-class>java.lang.Integer</variable-class>
                  <declare>true</declare>
                  <scope>NESTED</scope>
            </variable>
        <attribute>
                 <name>names</name>
                 <required>true</required>
                 <rtexprvalue>true</rtexprvalue>
        </attribute>

</tag>
```

2.3.b. filtrage

bodyContent.getString();+ traitements + bodyContent.getEnclosingWriter().write(..);

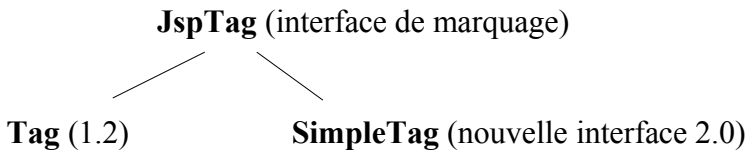
2.3.c. imbrication de balises

Une balise imbriquée peut accéder au contexte d'une balise englobante:

```
XXXContext ancetre = (XXXContext)
                TagSupport.findAncestorWithClass(this,XXXContext.class);
ancetre.getYYY();   // avec XXXContext = interface personnalisée implémentée par
                    //  balise (directement ou indirectement) englobante.
```

3. Programmation de "TagLib" personnalisés (jsp 2.0)

La version 2.0 a introduit une nouvelle interface très pratique "SimpleTag" qui n'hérite pas de Tag (car fonctionnement complètement différent).



3.1. Méthodes de l'interface SimpleTag

- **setParent(JspTag)** et **getParent()** qui permettent de définir et d'accéder au tag parent.
- **setJspContext(JspContext)** qui remplace le **PageContext** de l'interface **Tag**.
- **setJspBody(JspFragment)** qui définit le corps du tag.
  Cette méthode sera automatiquement appelée si le tag comporte un corps non vide.
- **doTag()** qui est l'unique méthode de **traitement** du tag

Lors de l'exécution d'un tag implémentant **SimpleTag**, Les méthodes **setParent(JspTag)** et **setJspContext(JspContext)** sont automatiquement déclenchées pour renseigner les valeurs, ainsi que d'éventuels attributs présents dans le tag.

NB:

- Contrairement à l'interface **Tag**, les objets de type **SimpleTag** ne sont pas mis en cache. A chaque exécution de la page un nouvel objet est créé ...
- La méthode **doTag()** traite alors son corps via l'objet **JspFragment** renseignée par le serveur d'application. Cet objet représente le code du corps du tag et peut être évalué autant de fois que nécessaire grâce à la méthode **invoke(Writer)** qui écrit le résultat dans le **Writer** spécifié.
- **invoke(null)** écrira le résultat de l'évaluation du corps du tag directement dans la page JSP ...
- L'interface **SimpleTag** permet une gestion plus simple des tags JSP. En effet, au lieu de gérer plusieurs méthodes pour chaque étape du tag (**doStartTag()**, **doInitBody()**, **doAfterBody()**, **doEndTag()**), l'unique méthode **doTag()** permet autant de possibilité en utilisant un **JspFragment** représentant le corps du tag.

3.2. Exemple simple (itération – n fois):

```
package tp.tags.v20;
import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class SimpleIterateTag extends SimpleTagSupport {
        private int count = 0;

        public void setCount (int value){this.count = value;}

        public void doTag() throws JspException, IOException {
                for (int i=0; i<count; i++)
                        getJspBody().invoke(null);
        }
}
```

déclaration dans le TLD:

```
<tag>
        <name>simpleIterate</name>
        <tag-class>packagexy.SimpleIterateTag</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
                <name>count</name>
                <required>true</required>
                <rtexprvalue>true</rtexprvalue>
                <!-- rtexprvalue à true si l'attribut peut recevoir le résultat
                   de l'interprétation d'un tag (ou équivalent en <%= %>) -->
        </attribute>
</tag>
```

NB: les "SimpleTag" supporte des "EL" mais ne supportent plus de contenu de type scriptlet (en <% .... %>) , d'où la valeur ***scriptless"*** dans le body-content .
Les autres valeurs possibles de body-content sont "***empty***" et "***tagdependent***" (l'interprétation du corps est faite par le tag).

Conséquence:

depuis JSP2 , de corps d'un tag doit plutôt être exprimé via des "EL" plutôt que des "<%= ...%>".

Par exemple l'équivalent d'un ancien *<%= page.getAttribute("personne").getNom() %>*

devra être ré-exprimé via : **${page["personne"].nom}**

Une éventuelle directive <%@ page isELIgnored="false" %> permet d'être certain que les "EL" seront interprétées (même sans fichier web.xml) .

Utilisation :

```
<%@taglib prefix="p" uri="WEB-INF/tlds/myTagLibV20.tld" %>
...
<p:simpleIterate count="5">
    <b>ok</b>
<p:simpleIterate>
```

3.3. TLD en version 2.0

Les **TLD** en version **2.0** ont maintenant une entête basée sur un schéma XSD plutôt qu'une DTD :

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<taglib 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/web-jsptaglibrary_2_0.xsd"
              version="2.0">
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <short-name>myTaglib</short-name>
               ...
</taglib>
```

3.4. JspContext (pour gérer les attributs selon les "scopes")

Au sein d'un "**SimpleTag**" , la méthode **getJspContext**() mène à un objet "**JspContext**" qui comporte les méthodes suivantes:

- JspWriter **getOut**() : accès à la variable out de la JSP
- Object **getAttribute**(String) : retourne un objet associé au paramètre (scope à page)
- Object **getAttribute**(String, int) : retourne objet avec un scope précis
- **setAttribute**(String, Object) : associe un nom à un objet (scope à page)
- **setAttribute**(String, Object, int) : associe un nom à un objet avec un scope
- Object **findAttribute**(String) : cherche l'attribut dans les différents scopes
- **removeAttribute**(String) : supprime un attribut

PageContext.PAGE_SCOPE , .REQUEST_SCOPE ,
.SESSION_SCOPE , .APPLICATION_SCOPE

Quelques exemples d'utilisations:

getJspContext().**setAttribute**("date1", new Date(), PageContext.PAGE_SCOPE);
getJspContext().**findAttribute**("date1");
getJspContext().**getAttribute**("date1", PageContext.PAGE_SCOPE);

Remarque:

Il est plus simple de définir la valeur d'un nouvel attribut qui sera potentiellement utilisé par la suite (dans le corps de la balise ou bien dans le reste de la page) que d'utiliser la gestion explicite des variables de scripts (via **TagExtraInfo** des versions antérieures**)** toujours valables en version 2.0 .

3.5. Collaboration entre tag "parent" et "enfant"

Le plus simple consiste à faire en sorte que:

* le tag parent initialise un attribut (en lui ou bien en scope)
* le tag enfant récupère cette valeur et en tienne compte (via getParent() instanceof .... ou bien getJspContext().findAttribute())

3.6. Corps d'un Tag / JspFragment

Rappel : on sein de la méthode ***doTag***() , un appel à **getJspBody**() retourne un objet de type **JspFragment** permettant de manipuler/évaluer le corps d'un "SimpleTag" .

L'objet JspFragment peut être évalué autant de fois que nécessaire grâce à la méthode **invoke**(Writer) qui écrit le résultat dans le "Writer" spécifié.

getJspBody().invoke(null); *// écrit directement dans la réponse (vers page html ou ...).*

StringWriter mon_buffer = new StringWriter();
getJspBody().invoke(mon_buffer);
// possibilité de traiter ici le buffer (analyse , filtrage , concaténation , ...)
getJspContext().getOut().println(mon_buffer.toString());*// ou mon_buffer.toString().toUpperCase()*

3.7. Attributs dynamiques

En plus des attributs statiques (qui doivent être déclarés via <attribute> dans la TLD) , il est possible de récupérer les valeurs de certains attributs dynamiques (non déclarés dans la TLD)

```
public class InputTag extends SimpleTagSupport implements DynamicAttributes {
  private Map attributes = new HashMap();
  private String type = null;  // premier attribut statique
  private String name = null; // deuxième attribut statique
  private String value = null;// troisième attribut statique

 public void setName(String name) {this.name = name;}
 public void setType(String type) {this.type = type;}
 public void setValue(String value) {this.value = value;}

  public void setDynamicAttribute(String uri, String localname, Object value)
                                                                    throws JspException {
      attributes.put(localname, value);
  }
...
}
```

Ceci nécessite

```
<dynamic-attributes>true</dynamic-attributes>
```

dans le bloc <tag> de la TLD.

XIV - Annexe – détails (WebApp,Servlet,useBean)

1. Quelques détails sur l'API des Servlets

1.1. Evolution de l'api des servlets:

« **Web Container** » et d' **Servlet API 2.x** .
Produit de référence (situation actuelle) : ***Jakarta-tomcat (versions 5.0 & 5.5) .***

NB : **Tomcat 5** intègre deux grands **connecteurs** :

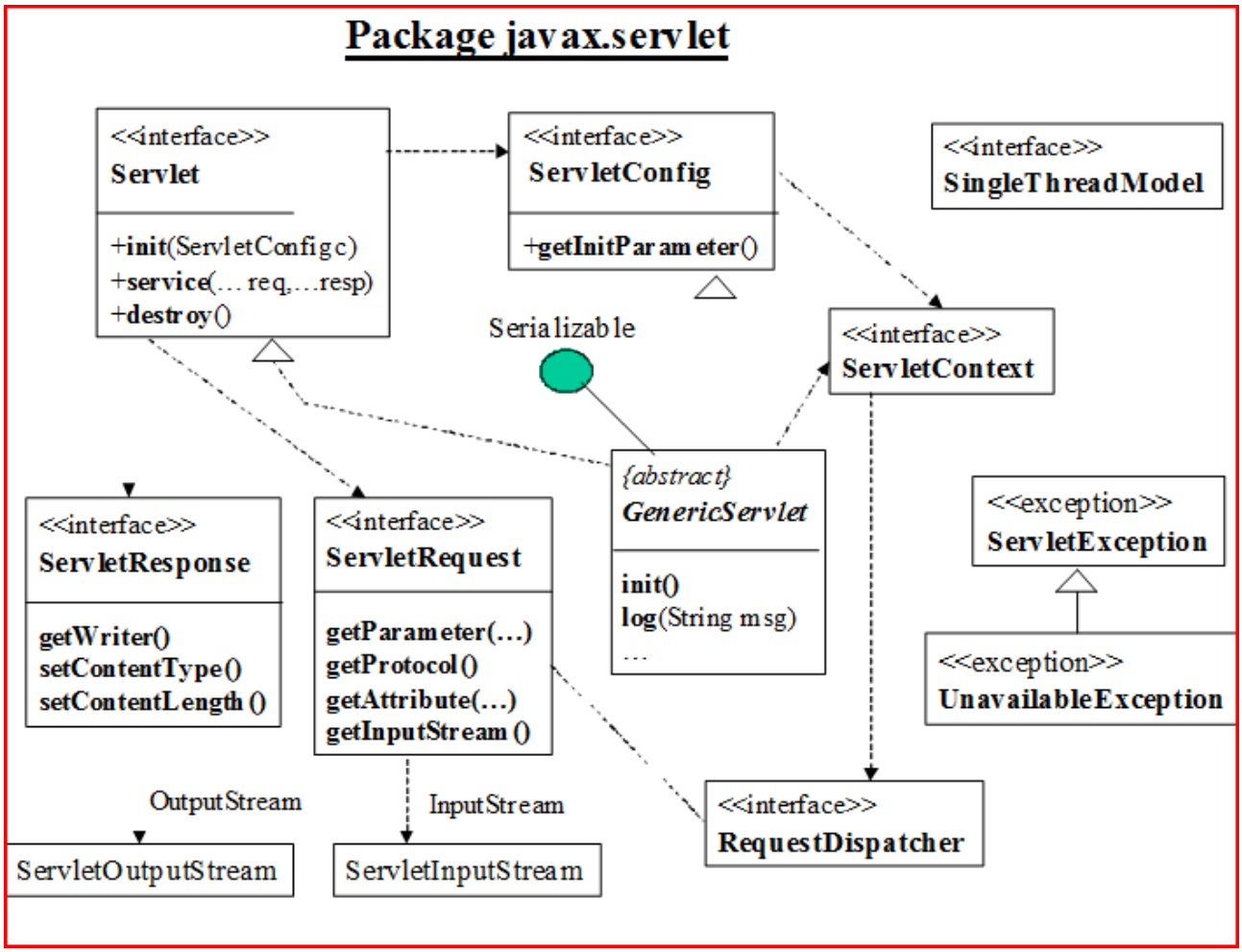
- « **http 1.1** » permettant à Tomcat de fonctionner en **autonome** (il intercepte alors directement les requêtes ***http://localhost:8080*** et y répond tout seul). C'est généralement le mode de fonctionnement utilisé pour le développement.
- « **mod_jk2** » permettant à Tomcat de réceptionner les requêtes http qui ont été préalablement interceptées par un vrai serveur Web (**httpd** de Apache, …) qui offre généralement de meilleurs performances pour les fichiers statiques (html) et qui est plus sophistiqué en ce qui concerne la gestion de la sécurité.
  C'est le mode utilisé en production (***http://www.xxx.yyy:80***/myJavaWebSite/…)

Le tableau ci dessous montre les principales différences entre les différentes versions :

Servlet API	***Principales caractéristiques (nouveautés)***
1.0	Bases de l'API des servlets, **Requêtes & Réponses http** , …, **service**() ,**doGet**() , **doPost**(), **init**(), **destroy**() , …
2.0	Gestion des **Cookies** et des **Sessions**. **Interface** de marquage **SingleThreadModel** (*alternative à synchronize*) . Response.**getWriter**() , request.**getReader**() pour l'internationalisation
2.1	Ajout de la classe **RequestDispatcher** (pour **forward**() et **include**()). Contrôle sur la durée de vie de la session : **setMaxInactiveInterval()** Méthode **setAttribute**() et **getAttribute**() sur **ServletContext** Nouvelle version de **init()** sans argument de type *ServletConfig* . Souvent accompagné de « JSP 1.0 »
2.2	Intégration dans J2EE (Servlet Engine ➔ Servlet Container). ***Ajout de la notion de descripteur de déploiement (web.xml***). + get/setLocale() + getNameDispacher() avec nom logique Pour l'objet **HttpSession**: plus de ***~~set/getValue()~~*** mais **get/setAttribute()** . Clarification : une seule instance de Servlet (si pas SingleThreadModel) Souvent accompagné de « ***JSP 1.1*** » et inclus dans ***J2EE 1.2***
2.3	Clarification : Java 2 (jdk >=1.2) obligatoire. *HttpUtils deprecated* . Notion de **Filtre** (nouvelle classe **Filter**). OK dans ***Tomcat 4.1*** et ***5*** Souvent accompagné de « ***JSP 1.2*** » et inclus dans ***J2EE 1.3***
2.4	Version associée à ***JSP 2*** et à ***J2EE 1.4*** . Descripteur de déploiement (web.xml) validé via schéma XML (pas DTD). Nécessite ***Tomcat 5*** (ou supérieur [5.5]).
2.5	Version associée à **JSP 2.1** et **JEE5** injection directe de dépendances (EJB3 / Ressources) via annotations Java5

	Nécessite **Tomcat6** (et/ou ***JBoss 5*** , ...) (jdk 5 ou 6)
3	Version associée à **JSP 2.2 et JEE6** **Tomcat 7 (jdk 6)**

1.2. Formalisation des bases de l'API des servlets:



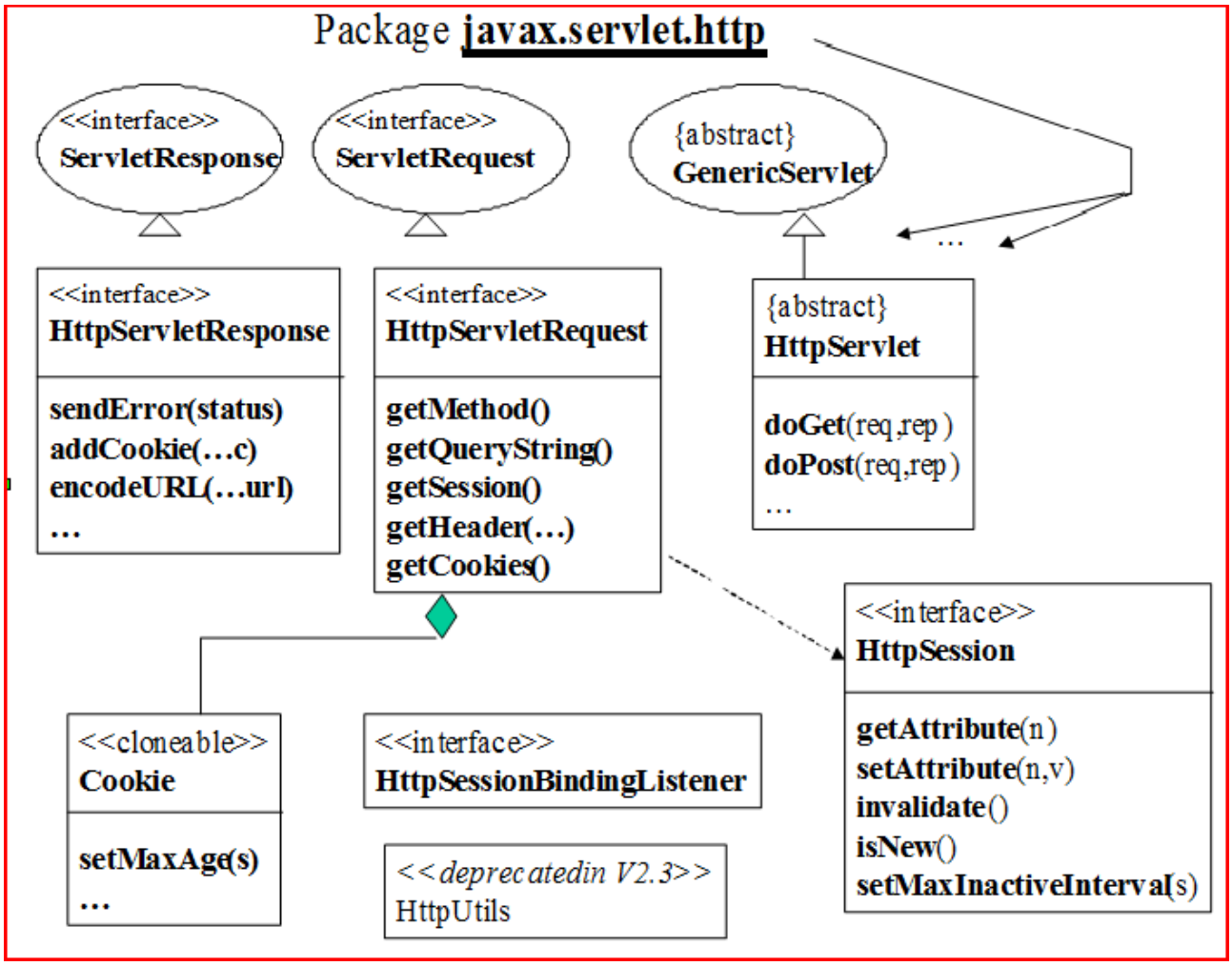
Nb:

- request.**getParameterNames()** retourne un objet de type ***Enumeration*** permettant de parcourir la liste des paramètres de la requête http.

- via l'objet **ServletContext** on peut ***journaliser des événements*** en invoquant la fonction **log("***message***")** ou bien **log("***message***",Throwable ex)** .
  TOMCAT utilise le journal ***%TOMCAT_HOME%\log\servlet.log*** .

- Etant donné que la classe abstraite **GenericServlet implémente à la fois les interfaces Servlet et ServletConfig** , on peut (dans une de ses sous classes) manipuler directement la méthode **getInitParameter**() sans avoir à appeler getServletConfig().

- La classe GenericServlet a définit une nouvelle version de la fonction **init() sans aucun argument**. Ce qui est beaucoup plus simple à surcharger/redéfinir.



NB:

- response.**setHeader("name","value")** permet de spécifier des valeurs dans l'entête de la réponse http.

- String chUrl = **response.encodeURL**("/siteY/servlet/servY") permet d'encoder une URL en tenant compte des caractères spéciaux et en ajoutant s'il le faut une information de session (?jsessionid=123456789) si le navigateur client ne supporte pas les cookies.

1.3. SingleThreadModel:

L'interface de marquage **SingleThreadModel** permet de demander au moteur de servlet de mettre en œuvre un mécanisme pour que **la méthode service() soit appelée par un seul thread à la fois sur une même instance du servlet** (Solutions courantes : **Pool d'instances**, Sérialisation des Threads (traitements effectués les un après les autres) , méthode mixte basée sur les 2 précédentes).

→ public ***class MonServlet extends HttpServlet*** **implements SingleThreadModel**
{…}

L'utilisation de SingleThreadModel est assez déconseillée (==> performances moyennes).

NB:
Depuis la version 2.2 de l'API des Servlets, il est clairement mentionné que **sans SingleThreadModel ,** les **différents Threads exécutant (de façon concurrente) le code d'un même type de servlet utilisent une et une seule instance du servlet**. L'utilisation (souvent

indispensable) du mot clef **synchronized** permet d'obtenir des résultats corrects.

1.4. ServletException

ServletException correspond au type d'exception qu'il est prévu de lancer au sein d'une méthode d'un servlet.

- Le constructeur **ServletException(String msg)** permet de lever une exception basée sur un simple message.

- Les deuxième et troisième constructeurs **ServletException(Throwable cause)** , **ServletException(String msg, Throwable cause)** permettent **d'encapsuler une exception quelconque dans une exception récupérable par l'environnement des servlets**.

- La méthode publique **getRootCause**() de la classe ServletException permet de récupérer l'exception encapsulée.

1.5. UnavailableException

UnavailableException est une ***sous classe de ServletException*** qui est associée à une sémantique particulière :

*Une méthode interne du servlet doit remonter une exception de type **UnavailableException** lorsqu'il souhaite indiquer au container Web que la servlet est **temporairement ou définitivement indisponible**.*

- Le premier constructeur **UnavailableException(String msg)** permet d'indiquer une **indisponibilité permanente**.

- Le second constructeur permet d'indiquer une **indisponibilité temporaire** dont la durée est exprimée en secondes : **UnavailableException(String msg, int nbSec**).

1.6. Page d'erreur personnalisée pour un type d'exception

Le descripteur de déploiement (***web.xml***) peut contenir une balise <error-page> permettant d'associer une url de page d'erreur avec une classe (précise) d'exception (ServletException ou dérivé).
Exemple :

```
<web-app>
…
<error-page>
        <exception-type>javax.servlet.UnavailableException</exception-type>
        <location>/unavailable.html</location>
</error-page>
…
</web-app>
```

1.7. Renvoyer des statuts « http »

Le protocole http comporte certains statuts bien normalisés (**constantes** de la classe **HttpServletResponse)** :
200 (OK),
301(Move) , 302 (Redirection)
400 (Bad request) , 401(Unauthorized), 403(Forbidden) , 404 (Not found) ,
500 (Internal Server Error),501 (Not Implement), 503(Unavailable Server, try again) …

L'objet **response** (de type **HttpServletResponse**) comporte :

- Une méthode **sendError(int status)** ou **sendError(int status,String msg)** pour renvoyer un statut d'erreur. ➔ Le container renvoie alors une page personnalisée ou bien une page par défaut pour signaler la raison de l'erreur.
- Une méthode **setStatus(int status)** pour renvoyer un statut qui n'est pas une erreur.
- Une méthode **sendRedirect(String chLocation)** pour indiquer au client qu'il est en cours de redirection (statut 302).

On peut içi encore préciser une page d'erreur personnalisée :

```
<web-app>
…
<error-page>
        <error-code>503</error-code>
        <location>/errors/TryAgain.html</location>
</error-page>
…
</web-app>
```

2. Détails sur les pages JSP

2.1. Différentes versions de l'Api « JSP » :

Api JSP	Principales caractéristiques (nouveautés)
1.0	Pas de standard portable pour les TagLib (extension pour balises personnalisées). Objet prédéfini session avec les anciennes méthodes getValue() et setValue() ➔ ***Souvent accompagné de « Servlet API 2.0 »*** .
1.1	Support standard pour les **TagLib** (***extension de balises***) . Nouvelles méthodes **setAttribute()** et **getAttribute()** sur les objets **session**, **application** et **request**. ➔ ***Souvent accompagné de « Servlet API 2.1 »***
1.2	***Quelques ajouts pour XML*** (**jsp:root** avec namespace et **jsp:text**)
2.0	Ajout de la syntaxe ***${nomBean.nomPropriete}*** (inspirée de JSTL / EL) et qui permet d'afficher directement une propriété d'un Bean. Autres ajouts importants inspirés de JSTL (boucle, ...) NB: **JSP 2** est associé à **J2EE 1.4** et est supporté par Tomcat 5

2.2. Equivalents syntaxiques pour XML

Si l'on souhaite manipuler automatiquement la page JSP par des générateurs de code basés sur des parseurs XML, on ne peut plus utiliser la syntaxe **<% %>** car il ne s'agit pas d'une balise bien formée XML.

Heureusement, il existe un ensemble d'équivalents XML .

Syntaxe classique	Equivalent XML
<%@ directiveName ATTRS **%>**	**<jsp:directive:directiveName** ATTRIBUTES />
<%! declarations **%>**	**<jsp:declaration>** declarations **</jsp:declaration>**
<% scriptlet code **%>**	**<jsp:scriptlet>** scriptlet code **</jsp:scriptlet>**
<%= expresion code **%>**	**<jsp:expression>** expression code **</jsp:expression>**

Une page JSP doit également comporter une unique balise racine (document) pour être conforme aux impositions d'XML :

```
< ! DOCTYPE root
PUBLIC "-//SUN Microsystems Inc.//DTD JavaServer Pages Version 1.1//EN"
"http://java.sun.com/products/jsp/dtd/jspcore_1_0.dtd ">

<jsp:root  version="1.0" xmlns:jsp=" http://java.sun.com/products/jsp/dtd/jsp_1_0.dtd ">
reste de la page JSP
</jsp:root>
```

2.3. Eventuelle incorporation d'applet via <jsp:plugin>

Une page HTML ou JSP peut incorporer une référence sur un applet qui sera interprété par le client (Internet Explorer, Mozilla, Opera, ...).
Etant donné qu'un Applet utilisant les classes SWING de java 2 ne fonctionne pas avec la machine virtuelle de niveau jdk1.1 de internet explorer, on est obligé d'utiliser un plug-in java dans le navigateur (client).
Ce gros plug-in (8 à 12 Mo) peut être téléchargé automatiquement (sur un réseau haut débit de type ADSL ou intranet). Le plugin Java comporte essentiellement le JRE (Java Runtime Environment) compatible JDK 1.2 ou 1.3 , ...
Autre considération: <APPLET > est une balise officiellement dépassée , la nouvelle norme (HTML4 , XHTML du W3C) préconise l'utilisation de <OBJECT type="..." >.
Pour masquer tous ces détails, il suffit d'employer l'action **<jsp:plugin>** .

```
<jsp:plugin type="bean|applet" code="packx.ClasseA.class" height="300" width="300"
 codebase="." archive="fic1.jar;fic2.jar" jreversion="1.2" iepluginurl="..." name="a" >
        <jsp:params>
                <jsp:param name="paramName" value="paramValue" /> ...
        <jsp:params>
         <jsp:fallback> Alternate text to display </jsp:fallback>
</jsp:plugin>
```

3. Détails sur les applications WEB / J2EE

3.1. Cookies

La gestion des **Cookies** (*RFC 2109*) s'effectue via la classe ***javax.servlet.http.Cookie***

```
Cookie c = new Cookie("back_color","blue");
c.setMaxAge(2*24*60*60); // expire dans 2 jours
response.addCookie(c);
```

Les instructions précédentes ont pour effet d'insérer un couple (nom=valeur) dans l'entête de la réponse http:

Set-cookie: back_color=blue; Domain="UrlServerWeb"; ...

Le navigateur (client) reçoit et stocke alors ce cookie (en mémoire ou bien sur fichier si la date d'expiration est précisée).

Lorsque le navigateur va plus tard envoyer de nouvelles requêtes vers le même site Web, il placera alors automatiquement dans l'entête http la liste des cookies préalablement récupérés et associés au site:
Cookie: back_color=blue; cookie2=valeur2; ...

Finalement, tout servlet de la même application web qui sera invoqué après cet «aller-retour» du cookie, pourra récupérer la valeur d'un des cookies de l'entête http de la requête:

```
Cookie[] tabCookies = request.getCookies();
…
```

3.2. Evénement activé en début et fin de session :

Un objet «*Attribut*» que l'on peut rattaché à une session (via les fonctions du précédent paragraphe) **peut être averti lorsqu'il rejoint ou quitte la session**.
Pour cela , notre objet «*Attribut*» doit être une instance d'une classe java quelconque mais implémentant néanmoins l'interface **HttpSessionBindingListener** (héritant elle même de java.util.EventListener).

Cette interface comporte les deux fonctions imposées suivantes:
public void **valueBound**(HttpSessionBindingEvent event) { }
→ *cette méthode est invoquée automatiquement lorsque l'objet est lié à la session.*
public void **valueUnbound**(HttpSessionBindingEvent event) { }
→ *cette méthode est invoquée automatiquement lorsque l'objet est détaché de la session (soit explicitement, soit implicitement: session invalidée ou expirée (timeout)).*

NB: l'objet event de type **HttpSessionBindingEvent** comporte:

- Une méthode **getName**() retournant de l'on de l'attribut (celui indiqué via setAttribute)
- Une méthode **getSession**() renvoyant une référence sur l'objet Session.

3.3. Gestion de l'internationalisation (in-18):

La mise en forme du texte, des dates et des nombres doit normalement s'effectuer en fonction des paramètres de localisation du poste client (sur lequel est situé le navigateur internet) et non pas en fonction de la localisation de la machine virtuelle java du conteneur web qui accompagne le serveur web.

A cet effet, l'interface **ServletRequest** comporte une méthode appelée **getLocale()** permettant de récupérer les infos de localisation liées au poste client:

```
java.util.Locale currentLocale = request.getLocale();
```

On peut ensuite déclencher tout un tas de formatages en fonction de la bonne localisation:

```
ResourceBundle myResources =
           ResourceBundle.getBundle("MyResources", currentLocale);
```

String chWelcome =myResources.getString("welcome");

DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT,currentLocale);
String chDate = df.format(date);

NumberFormat nf = NumberFormat.getInstance(currentLocale);
String chNum = nf.format(123.456);

...

3.4. Générer des graphiques SVG depuis un servlet

Le format de fichier **SVG** (Scalable Vector Graphic) est **basé sur XML**, et permet de coder des **images vectorielles** (compactes et très pratiques pour des diagrammes à générer dynamiquement).
Attention: un fichier svg ne pourra être convenablement affiché que si le navigateur est très récent ou s'il dispose d'un plug-in capable de le faire (*ex*: **SvgViewer** d'adobe et accompagnant acrobat reader 5, 6 et 7).

Un servlet java peut assez facilement générer un fichier svg en adoptant les quelques principes suivants:

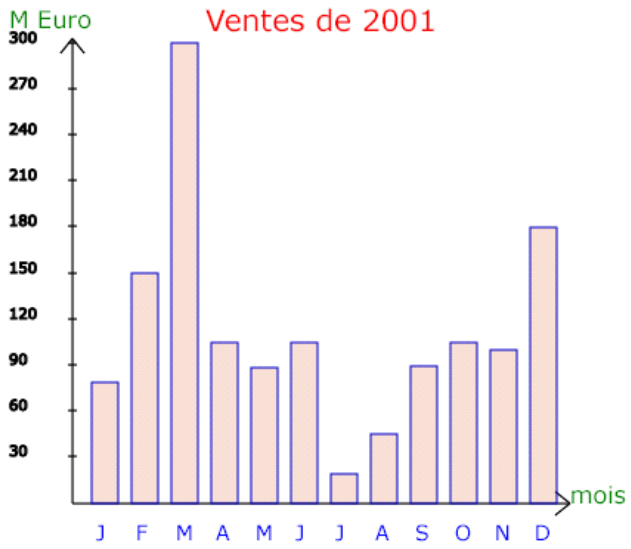
- Renvoyer le bon type MIME:

```
response.setContentType("image/svg+xml");
```

- Déléguer la génération de la "grosse chaîne de caractère Xml " à un java bean spécialisé pour ça et qui pourra avantageusement tirer parti de la classe **MessageFormat** pour passer des valeurs spécifiques aux balises SVG classiques suivantes:

```
String sSvgFt = "<svg width="{2}" height="{3}" >";
String sRectFt = "<rect x="{0}" y="{1}" width="{2}" height="{3}" style="{4}" />";
String sLineFt = "<line x1="{0}" y1="{1}" x2="{2}" y2="{3}" style="{4}"/>";
String sTextFt = "<text x="{0}" y="{1}" style="{4}"> {5} </text>";

String sStyleTexte="font-family: Verdana; font-size: 14; fill: blue; ";
String sStyleRect="stroke: mediumblue; fill: coral; fill-opacity: 0.2";
```



Nb: on peut générer des diagrammes sous forme d'images gif (ou jpeg):
source = new **MemoryImageSource**(....);
defaultToolkit.**createImage**(source); .... GifEncoder --> produit tiers (ACME).

4. Utilisation de JavaBean (JSP 1 et 2)

Principe:

Au lieu de coder l'intégralité des traitements entre des balises <% %> disséminées aux quatre coins d'une page JSP, il est préférable de déléguer une grande partie des traitements (non graphiques) à un composant java invisible (JavaBean).

Avantages:

- Meilleur lisibilité.
- Composant générique réutilisable
- Meilleur Modularité
- ...

Syntaxe:

La construction suivante (généralement placée dans le haut de la page JSP) permet de créer un nouveau composant Java ou bien de s'y rattacher s'il existe déjà.

```
<jsp:useBean id="nomObj" class="pack.ClasseDuBean" scope="session" />
```

- **scope = "session" , "application" , "page" ou "request" .**
- **type="SurClasse ou Interface" (pour retrouver via nom , insuffisant en création)**
  *(**type** peut être mentionné à la place de **class** dans certains cas)*

Code java approchant l'action <jsp:useBean> :

```
<% pack.ClasseDuBean nomObj = null;
   nomObj = (pack.ClasseDuBean) <scope>.getAttribute("nomObj");
   if(nomObj==null) { nomObj = new pack.ClasseDuBean();
                      <scope>.setAttribute("nomObj",nomObj); } %>
```

Nb: La portée (attribut <**scope**>) permet de préciser à quel endroit est (ou sera) rattaché le composant JavaBean. Ceci a une influence sur la durée de vie de l'objet et sur sa plage d'accessibilité:

Portée (scope)	Durée de vie	Accès possibles depuis ...
page	_JspService() associée à la page JSP courante	page JSP courante seulement
request	Tous les servlets et pages Jsp qui s'enchaînent (collaborent) pour traiter la requête Http courante.	servlets et pages Jsp reliés par rd.forward(request,...) ou rd.include(request,...).
session	durée de vie de la session (HttpSession) (voir chapitre suivant)	Tous les servlets et pages Jsp de la même application Web.
application	Jusqu'à l'arrêt (ou ré-initialisation) du container web (voir ServletContext)	Tous les servlets et pages Jsp de la même application Web.

L'instruction suivante permet de déclencher la mise à jour d'une ou plusieurs propriétés d'un composant Java en fonction de la valeur des paramètres d'entrée de la requête HTTP.

```
<jsp:setProperty  name="nomObj"  property="*"  />
```

pour déclencher automatiquement toutes les fonctions internes de type ***setXxx(x)*** *du composant à partir des valeurs obtenues par* ***x=request.getParameter("xxx")*** .

Pour mettre à jour la seule propriété intitulée "user" en fonction de la valeur du paramètre http "username" on peut utiliser la variante suivante:

```
<jsp:setProperty  name="beanName" property="user" param="username" />
```

<==> <% beanName.setUser(request.getParameter("userName")); %>

Pour directement associer une valeur à une propriété d'un composant javaBean, il suffit de lancer l'instruction suivante:

```
<jsp:setProperty  name="beanName" property="nb" value="<%=i+1%>" />
```

<==> <% beanName.setNb(i+1); %>

Inversement pour récupérer et afficher la valeur d'une propriété d'un composant javaBean, il suffit de lancer l'instruction symétrique qui suit:

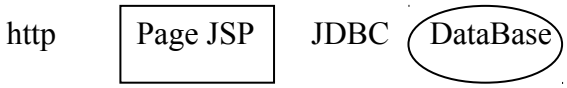
```
<jsp:getProperty  name="objName"  property="nb" />
```

<==> <% out.println(objName.getNb()); %> <==> <%=objName.getNb() %>

Modèles d'architectures simples

4.1. Vue de Page

La page JSP fait tout le travail (Présentation + Traitements)



4.2. Vue de Page avec Bean

La page JSP ne s'occupe que de l'affichage et délègue les traitements à un ou plusieurs objets "JavaBeans".

XV - Annexe – aspects avancés (servlets, ...)

1. Utilisation de fichiers ".properties"

Bien que ***web.xml*** puisse comporter des paramètres d'initialisation (<init-param> dans <servlet> ou bien <context-param>) , il est quelquefois nécessaire d'avoir recours à des ***fichiers de propriétés*** .

Ceux-ci offrent l'avantage d'être interchangeables et se prêtent ainsi très bien au paramétrage des aspects spécifiques au serveur où sera effectué le déploiement.

WEB-INF/fic1.properties

```
propriete1=valeur_de_la_propriete1
propriete2=valeur_de_la_propriete2
propriete3=valeur_de_la_propriete3
```

web.xml

```
...
<web-app id="WebApp">
      <display-name>tp_web</display-name>

      <context-param>
          <param-name>PROPERTIES_FILENAME</param-name>
          <param-value>WEB-INF/fic1.properties</param-value>
      </context-param>
...
```

Morceau de code d'un servlet:

```
try{
  ServletContext application = this.getServletContext();
  String propFicName=application.getInitParameter("PROPERTIES_FILENAME");
  String propPathName=application.getRealPath(propFicName);
  java.util.Properties prop = new java.util.Properties();
  prop.load(new FileInputStream(propPathName));
  String valeurP2=prop.getProperty("propriete2","valeur_par_defaut2");
  System.out.println("P2="+valeurP2);
  String valeurP4=prop.getProperty("propriete4","valeur_par_defaut4");
  System.out.println("P4="+valeurP4);
} catch(Exception ex)
 { ex.printStackTrace();
}
```

2. upload file (remontée de fichier vers le serveur web)

Le formulaire HTML doit être de type "**multipart**" de façon à pouvoir véhiculer le contenu du fichier joint du navigateur vers le serveur Web:

```
<HTML>
<HEAD><TITLE>WebSMTP</TITLE></HEAD>
<BODY>
<FORM ACTION="WebSMTP" METHOD="POST" ENCTYPE="multipart/form-data">
<B>From:</B> <INPUT NAME="from"><BR/>
<B>To:</B> <INPUT NAME="to"><BR/>
<B>Cc:</B> <INPUT NAME="cc"><BR/>
<B>Subject:</B> <INPUT NAME="subject"><BR/>
<TEXTAREA NAME="body"></TEXTAREA><BR/>
                                                    Parcourir...
<INPUT TYPE="FILE" NAME="attachment1"><BR/>
<INPUT TYPE="FILE" NAME="attachment2"><BR/>
<INPUT TYPE="FILE" NAME="attachment3"><BR/>
<INPUT TYPE="SUBMIT">
</FORM>
</BODY>
</HTML>
```

Lorsque l'on appuie sur le bouton "**Submit**", les données sont envoyées au serveur via http en mode post sous la forme d'un seul grand bloc d'octets où chaque partie est délimitée par quelque chose du genre:

----------------------------7d211c3040280

Un petit dump binaire effectué à partir de la routine suivante permet de connaître le format du corps d'une requête http dont le type MIME est "**multipart/form-data**" :

```
public void doBinaryDump(HttpServletRequest request)
  throws IOException
  {
      ServletInputStream in = request.getInputStream();
      FileOutputStream out = new FileOutputStream("c:\\temp\\uploadDump.bin");
      int l = request.getContentLength();
      byte buf[]= new byte[l];
      in.read(buf);
      out.write(buf);
      out.close();
  }
```

Exemple de résultat:

```
----------------------------7d211c3040280
Content-Disposition: form-data; name="from"

didier@xxx.fr
----------------------------7d211c3040280
Content-Disposition: form-data; name="to"

toto@yyy.com
----------------------------7d211c3040280
Content-Disposition: form-data; name="attachment1"; filename="C:\tp\Readme.txt"
```

```
Content-Type: text/plain

ligne1 du fichier
ligne2 du fichier
----------------------------7d211c3040280
Content-Disposition: form-data; name="attachment2";
filename="C:\tp\tp_eclipse_ant\tp_web\SendMail.html"
Content-Type: text/html

<HTML>
... conenu html ...
</HTML>
----------------------------7d211c3040280
Content-Disposition: form-data; name="attachment3"; filename=""
Content-Type: application/octet-stream


----------------------------7d211c3040280--
```

Le servlet doit donc s'appuyer sur un composant utilitaire capable de décrypter et décomposer cet ensemble de façon à extraire chaque partie dans différentes zones mémoires (ex: paires(clef,valeur) d'une Hashtable).

Bien qu'assez complexe en interne, cet exemple de classe utilitaire s'utilise très simplement depuis un servlet:

```
...
if(request.getContentType().startsWith("multipart/form-data"))
        {
         UploadUtil upld = new UploadUtil();
         upld.doUpload(request);

         String valFrom = upld.getParameter("from");
         String valTo = upld.getParameter("to");

         Map attachementsMap = upld.getAttachmentEntries();
         if(attachementsMap != null)
              {
              Iterator it = attachementsMap.entrySet().iterator();
               while(it.hasNext())
                  {
                   Map.Entry mapEntry = (Map.Entry) it.next();
                   AttFileEntry f= (AttFileEntry) mapEntry.getValue();
                   ...(f.fileName,f.contentType,f.dataBuf);
                   }
              }
          }
...
```

Code interne de la classe utilitaire gérant l' "upload file" :

```
public class AttFileEntry
{
public String fileName=null;
public String contentType=null;
public byte[]dataBuf=null;
}
```

NB: Le code source (très technique) qui suit est très difficilement compréhensible (ne surtout pas essayer de le décortiquer en cours) ==> une fois encapsulé dans une classe utilitaire on peut le réutiliser simplement. !!!!

L'***algorithme*** est une ***machine à état*** qui ***lit le fichier lire par ligne*** et qui analyse celles-ci en fonction du contexte:

state	signification
0	début d'une nouvelle partie (ou fin)
1	lecture de filename=... , Content-type:....
2	lecture du nom d'un champ (name=....)
3	lecture des données d'un fichier joint
4	lecture de la valeur associée à un champ

```
public class UploadUtil {
        private Hashtable fields = null;
        private Hashtable attachements = null;

        // retourne une table de couples (nom,ObjetValeur)
        public Map getTextEntries()
        {       Map entries; entries =  fields; return entries;         }

        // pour récupérer une par une les entrées textuelles du form (POST)
        public String getParameter(String paramName)
        { if(fields == null) return null;
        return (String) fields.get(paramName);      }

        // retourne une table de couples (nom,AttFileEntry)
        // où chaque AttFileEntry correspond à [.fileName, .contentType, .dataBuf ]
        public Map getAttachmentEntries()
        {Map entries;      entries =  attachements; return entries;      }

public void doUpload(HttpServletRequest request)
                throws IOException, MessagingException
  {
    // récupérer le séparateur "--189018217873" dans la variable boundary
    // en analysant le "Content-Type" de l'entête de la requête Http:
     String boundary =  request.getHeader("Content-Type");
     int pos = boundary.indexOf('=');
     boundary = boundary.substring(pos + 1);
     boundary = "--" + boundary;

          // flux permettant de lire l'ensemble de la requête Http:
     ServletInputStream in =  request.getInputStream();

     byte[] bytes = new byte[512]; // données qui seront lues dans une ligne
     int state = 0;
     // buffer pour récupérer le contenu d'un fichier joint :
     ByteArrayOutputStream buffer = new ByteArrayOutputStream();
     String name = null, /* nom d'un champ */
           value = null, /* valeur d'un champ */
           filename = null,  /* nom d'un fichier joint */
           contentType = null; /* type Mime d'un fichier joint */
     // Dictionnaire à remplir :
     fields = new Hashtable();//couples (nom,valeurTexte)
     attachements = new Hashtable();//couples (nom,AttFileEntry)

     int i = in.readLine(bytes,0,512);    // lire la première ligne (512 octets maxi)

    while(-1 != i)  // tant qu'il reste des lignes à lire et à analyser
     {
       String st = new String(bytes,0,i); // st = ligne lue sous forme de String

       if(st.startsWith(boundary)) // si séparateur final trouvé
       {
         state = 0;
         if(null != name) // si un champ a été précédemment lu
```

```
        {
          if(value != null) // si champ simple (pas de fichier joint)
            fields.put(name, value.substring(0, value.length() - 2)); // -2 to remove CR/LF
          else if(buffer.size() > 2)  // si buffer rempli avec données d'un fichier joint
         else if(buffer.size() > 2)
          {
         AttFileEntry fe = new AttFileEntry();
         fe.fileName=filename;
         fe.contentType=contentType;
         fe.dataBuf=buffer.toByteArray();
         attachements.put(name,fe);
          }
         // replacer les variables dans l'état initial:
          name = null;  value = null;
          filename = null;  contentType = null;
          buffer = new ByteArrayOutputStream();
        }
      }

      else if(st.startsWith("Content-Disposition: form-data") &&  state == 0)
     // si début d'une nouvelle partie
      {
        StringTokenizer tokenizer = new StringTokenizer(st,";=\"");
        while(tokenizer.hasMoreTokens())
        {
          String token = tokenizer.nextToken();
          if(token.startsWith(" name"))
          {
            name = tokenizer.nextToken();
            state = 2; // le nom du champ vient d'être lu
          }
          else if(token.startsWith(" filename"))
          {
            filename = tokenizer.nextToken();
            StringTokenizer ftokenizer = new StringTokenizer(filename,"\\/:");
            filename = ftokenizer.nextToken();
            while(ftokenizer.hasMoreTokens())
              filename = ftokenizer.nextToken();
            state = 1; // le nom du fichier joint vient d'être lu
            break;
          }
        }
      }
      else if(st.startsWith("Content-Type") &&  state == 1)
      // s'il faut lire une nouvelle ligne correspondant au type d'un fichier joint
      {
        pos = st.indexOf(":"); // + 2 to remove the space, - 2 to remove CR/LF
        contentType = st.substring(pos + 2,st.length() - 2);
      }
      else if(st.equals("\r\n") && state == 1)  // si state == 1 et lecture d'une ligne vide
        state = 3; // pour lire ensuite les données du fichier joint
      else if(st.equals("\r\n") && state == 2)
        state = 4;  // pour lire ensuite la valeur d'un champ
      else if(state == 4) // lire la valeur d'un champ sur la ligne courante
        value = value == null ? st : value + st; //concaténer si besoin avec ligne
                                                 // précédente cas d'une TextArea)
      else if(state == 3) // ajouter le contenu de byte dans le buffer (données fic joint)
        buffer.write(bytes,0,i);
      i = in.readLine(bytes,0,512); // lire la ligne suivante (itération - boucle principale)
    } // end of while
  } // end of doUpload()
}// end en class
```