Développement WEB
avec JEE
(*Java Enterprise Edition.*)
&
Tomcat

Janvier 2007

Dernière MàJ: Mars 2020

Mr. NDONG

Plan du cours

Introduction

Les serveurs d'applications : Apache Tomcat,

Les applications Web: configurations

L'API Servlet

Technologie Java Server Pages (JSP)

Les frameworks

- **Struts**
- **Spring**
- **Hibernate**

Technologie Java Server Faces (JSF)

Introduction

JEE est un ensemble de technologies, créées par SUN MicroSystems, permettant de réaliser des applications WEB mais aussi des applications standalone, en utilisant le langage JAVA.

Cette édition est dédiée à la réalisation d'applications pour entreprises.

JEE est basée sur J2SE (Java 2 Standard Edition) qui contient les API de bases de java.

JEE est une plate−forme fortement orientée serveur pour le développement et l'exécution ***d'applications distribuées***.

Elle est composée de deux parties essentielles :

- un ensemble de spécifications pour une infrastructure dans laquelle s'exécute les composants écrits en java : un tel environnement se nomme **serveur d'application.**
- un ensemble d‘ API qui peuvent être obtenues et utilisées séparément. Pour être utilisées, certaines nécessitent une implémentation de la part d'un fournisseur tiers.

Cet ensemble regroupe les servlets, JSP, services web,….

Cette technologie permet de créer des sites web ou des intranet en Java.

La maîtrise du langage Java constitue un pré requis dans le développement d’applications avec JEE.

L'utilisation de JEE pour développer et exécuter une application propose plusieurs avantages :

- ✓ une architecture d'application basée sur les composants qui permet un découpage de l'application et donc une séparation des rôles lors du développement (***modèle MVC***)
- ✓ la possibilité de s'interfacer avec le système d'information existant grâce à de nombreuses API : JDBC, JNDI, JMS ...

- ✓ la possibilité de choisir les outils de développement et le ou les ***serveurs d'application*** utilisés qu'ils soient commerciaux ou libres.

JEE permet une grande flexibilité dans le choix de l'architecture de l'application en combinant les différents composants. Ce choix dépend des besoins auxquels doit répondre l'application mais aussi des compétences dans les différentes API de JEE. L'architecture d'une application se découpe en au moins trois tiers :

- ***la partie cliente*** : c'est la partie qui permet le dialogue avec l'utilisateur.
  Elle peut être composé d'une application standalone, d'une application web ou d'applets
- ***la partie métier :***
- ***la partie données :***

JEE-Platform

Une *platform* est une ***base générique*** fournissant un ensemble de fonctionnalités utiles pour une majorité d'applications. Une plateforme se construit sur la base d'un ensemble de besoins génériques partagés entre plusieurs applications.

Avantages d'une platform:

- ✓ l'équipe de développement n'a pas à s'acquitter de développer certaines tâches (connexion à la base de données par exemple, gestion d'objets …). Ce sont des tâches que l'on retrouve très souvent dans un grand nombre de projet et qui n'ont pas d'intérêt à être re-coder à chaque fois (perte de temps et d'argent).
- ✓ forte stabilité (ça évite des débuggages inutiles !).

- ✓ Un autre avantage est la facilité de prise en main des API de cette plateforme.
  En effet, celle-ci cache très souvent la complexité d'accès à telle ou telle ressource et permet donc un gain de temps énorme pour le développeur qui a donc plus de temps pour se préoccuper du fonctionnement réel de son application (pas de tâche ardue ou générique à développer).

Inconvénients: temps d'apprentissage et de maîtrise.

On parle généralement de **«plate-forme JEE»** pour désigner l'ensemble constitué des services (API) offerts et de l'infrastructure d'exécution.

Composantes de JEE

JEE comprend :

- Les spécifications du serveur d'application, c'est-à-dire de l'environnement d'exécution : JEE définit finement les rôles et les interfaces pour les applications ainsi que l'environnement dans lequel elles seront exécutées. Ces recommandations permettent ainsi à des entreprises tierces de développer des serveurs d'application conformes aux spécifications ainsi définies, sans avoir à redévelopper les principaux services.
- Des services, au travers d'API , c'est-à-dire des extensions Java indépendantes permettant d'offrir en standard un certain nombre de fonctionnalités. Sun fournit une implémentation minimale de ces API appelée JEE SDK (JEE Software Development Kit).

JEE bénéficie des avantages et inconvénients du langage Java, en particulier une bonne portabilité et une maintenabilité du code
L'architecture JEE repose sur des composants distincts, interchangeables et distribués, ce qui signifie notamment :

- *Qu'il est simple d'étendre l'architecture.*
- *Qu'un système reposant sur JEE peut posséder des mécanismes de haute disponibilité, afin de garantir une bonne qualité de service.*
- *Que la maintenabilité des applications est facilitée.*

Les acteurs d'une application JEE

La réalisation d'une application basée sur l'architecture JEE fait appel à différents types de compétences allant de la ***conception*** jusqu'à la ***supervision*** de l'application en passant par le ***développement*** et le ***déploiement***.

Afin de pouvoir maîtriser ce processus JEE adopte l'approche des partages des responsabilités.

Plus spécifiquement pour les EJB, cette approche définit plusieurs niveaux de responsabilité :

- *Le fournisseur des EJB* : c'est l'acteur qui fournit des composants métiers réutilisables soit par l'achat à un fournisseur de composants, soit par développement interne ;
- *L'assembleur d'applications* : l'acteur qui est capable de construire une application à partir d' EJB existants ;
- Le *déployeur* : l'acteur qui récupère l'application et s'occupe de son déploiement dans un serveur d'applications ;
- *L'administrateur* : l'acteur qui contrôle le fonctionnement du serveur d'application et assure la supervision des applications ;
- *Le fournisseur de conteneur* : l'éditeur qui commercialise un conteneur web ou un conteneur EJB ; cet éditeur commercialise souvent un serveur d'application incluant ces conteneurs ;
- *Le fournisseur de serveur* : c'est l'éditeur qui commercialise un serveur d'application (BEA, IBM etc...)

Environnement de Développement

JEE fournit un ensemble d'API permettant de construire des sites WEB avec la technologie JAVA.

Pour cela, on utilise un **serveur d'application JEE**:

Un ***serveur d'application*** permet de charger et d'exécuter les **servlets, EJB, …** dans une JVM.

C'est une extension du serveur web.

Ce serveur d'application contient entre autre un **moteur de servlets** qui se charge de manager les servlets qu'il contient.

L'utilisation d'un IDE (***Integrated Development Environment***) est obligatoire comme **Eclipse** ou **JBuilder,** ou **NetBeans**.

Dans ce cours, on utilisera ***Eclipse***, vu sa gratuité, sa robustesse et son extensibilité grâce aux plugins. On utilisera particulièrement la version nommée **Eclipse JEE** spécialement conçu pour le développement JEE.

Serveur d'applications

Tomcat est un **conteneur de servlets** et de JSP (Java Server Pages) développé par la *Fondation Apache* (ASF, Apache Software Foundation).
Apache est de nos jours le serveur Web le plus puissant du World Wide Web.

L'intérêt grandissant de Tomcat s'explique du fait que Sun et Apache ont annoncé que ce dernier deviendrait l'implémentation officielle des spécifications des *Servlets* et des *JSP*.

Tomcat est fiable, évolutif et adapté à l'utilisation professionnelle.

La définition suivante aussi est souvent donnée:

Tomcat est un serveur d'applications Java

Un *serveur d'applications* est un serveur capable d'exécuter des applications web c'est-à-dire des applications destinées à traiter des requêtes web entrantes, et à générer une réponse adéquate.
Un serveur d'applications a un rôle distinct de celui d'un serveur web classique.
Ce sont deux composants indispensables. Tomcat sait faire les deux.

L'objectif d' un serveur d'application est de *simplifier le développement des applications* à architecture distribuée, en permettant une **séparation** plus grande entre les 3 couches de l'application (présentation, logique et données).

Le développeur peut alors se concentrer sur la ***logique de son application***, le reste étant pris en charge par la structure d'accueil :

- ✓ sessions
- ✓ transactions,
- ✓ persistance,
- ✓ sécurité
- ✓ nommage,
- ✓ charge

Plusieurs voies existent pour ***programmer par composants*** sur la couche application, les plus connues étant **DCOM** de Microsoft (*Distributed Component Object Model*) et **EJB** (*Enterprise Java Bean*).

Motivations pour les serveurs d'application

Productivité

Les serveurs d'application permettent de développer plus ***rapidement*** les applications et de *se focaliser sur la logique métier* de l'application plutôt que sur des tâches de bas niveau (communication entre composants,…).

Intégration avec l'existant

L' un des objectifs des serveurs d'application est de mettre à la disposition des développeurs des ***moyens standards d'accès aux données*** de ces systèmes.

Liberté de choix

La rapidité de développement d'une application dépend de plus en plus de la mise en œuvre de solutions hétérogènes au sein d'une même application.

JEE mécanismes et technologies standards pour la création d'application, ce qui permet d'utiliser des produits d'éditeurs différents, par exemple des environnements de développements et des serveurs JEE issus de plusieurs éditeurs.

L'utilisation de standards permet d'utiliser ses propres composants pour les fonctions spécifiques de l'application et d'utiliser des composants crées par des éditeurs de logiciels pour les fonctions de base.

Motivations pour les serveurs d'application

Réponse adaptée à la demande

Les architectures avec serveurs d'applications permettent la mise en place d'applications qui supportent facilement la ***montée en charge***, sans avoir à recoder l'application.

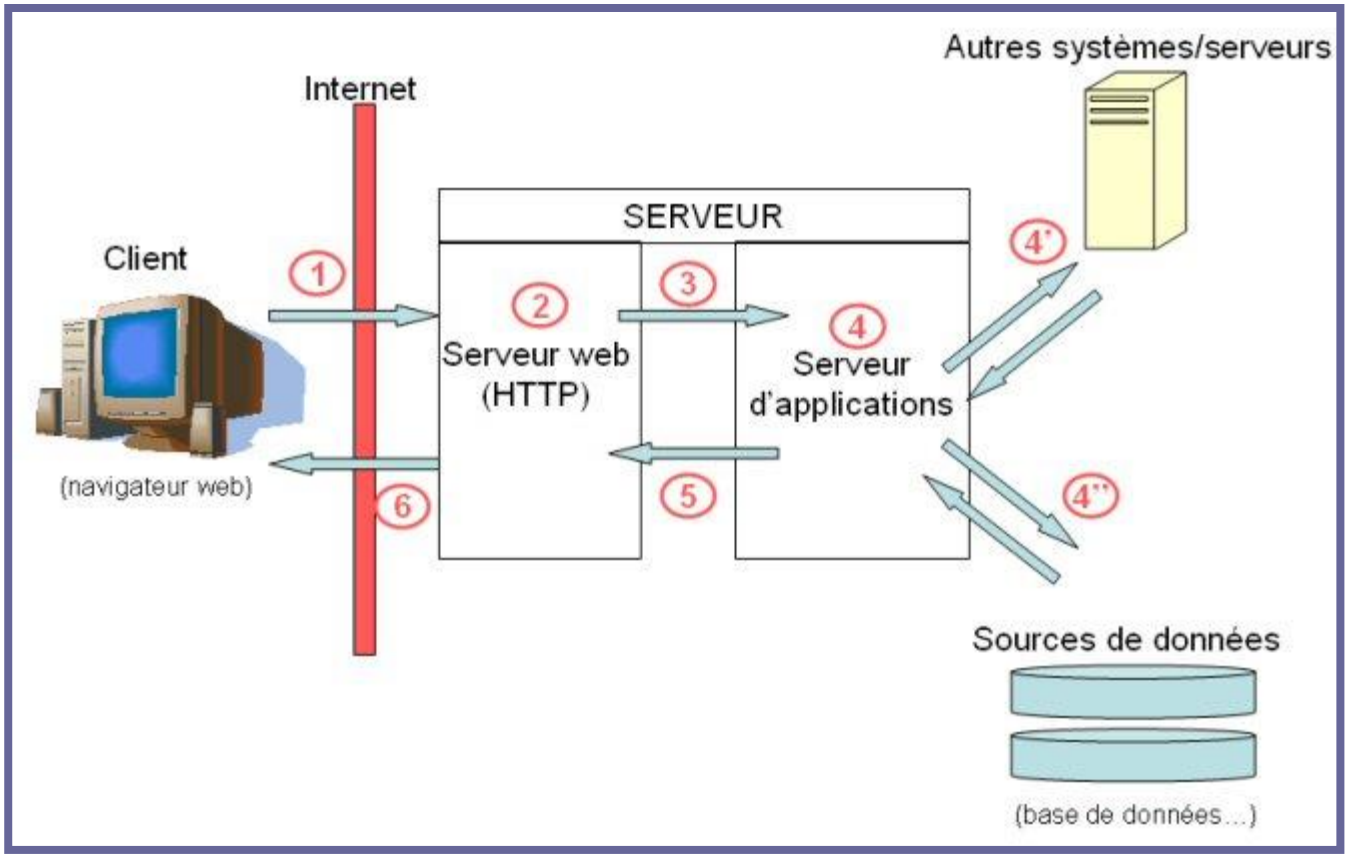
L'architecture JEE met en place des mécanismes simples pour répartir la charge entre plusieurs serveurs, sur les différentes couches de l'application.

Maintenance et sécurité

L'un des objectifs des serveurs d'application est de mettre en place une architecture de sécurité qui soit efficace sur tous les tiers d'une application

Fonctionnement des serveurs d'applications

Le schéma suivant montre le déroulement classique d'une requête vers un serveur d'applications :



Installation et configuration de Tomcat

Tous les serveurs JEE dépendent de la JVM pour pouvoir s'exécuter correctement. Donc vous devez installer le JDK.

Télécharger et installer une version de Apache Tomcat (par exemple **Apache Tomcat 9.0**).

Vous pouvez télécharger la version binaire **.ZIP** et décompresser dans une partition système (sous **Windows**, vous pouvez le décompresser dans le répertoire **C:**.

Configuration de Tomcat sous Windows

Pour que Tomcat s'exécute, il faut tout d'abord qu'il trouve le JDK.

Pour ce faire, vous devez configurer une ***variable d'environnement*** nommée

JAVA_HOME dont la valeur est le **chemin d'accès à la machine virtuelle**.
Elle est identique à tous les systèmes d'exploitation.

Par exemple si j'installe le **JDK 1.8**, la valeur de cette variable est:

C:\Program Files\Java\jdk1.8.0_91

Démarrer Tomcat sous Windows

Démarrage si vous avez installé le .exe

Sous Windows en installant le .exe de Tomcat, vous pouvez choisir de l'installer en **tant que service**.

De ce fait il sera lancé à chaque amorçage de votre système, ce qui vous évitera du coup de le démarrer manuellement.

Et pour l'arrêter ou le redémarrer utilisez le panneau de contrôle des ***services*** du *panneau de configuration*.

Si vous désirez le démarrer *manuellement* vous pouvez faire:

Démarrer → Programmes → Apache Tomcat 5.0 → Configure Tomcat

Et ensuite cliquer sur le bouton **start** de la fenêtre qui apparaît.

Démarrage si vous avez installé le .ZIP

Ou bien encore, ouvrir un terminal DOS et faire depuis la racine C:\:

cd tomcat 9.0

cd bin

Ce qui vous donne:

C:\tomcat9.0\bin> startup.bat

Pour arrêter **Tomcat** utilisez la commande **shutdown.bat**

Tomcat est démarré (1/2)

Lorsque Tomcat est bien démarré, vous pouvez vérifier s'il fonctionne correctement
En tapant l'URL http://localhost:8080/
Si tout se passe bien vous aurez la page de configuration par défaut.

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/9.0.10

APACHE SOFTWARE FOUNDATION http://www.apache.org/

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:
Security Considerations HOW-TO
Manager Application HOW-TO
Clustering/Session Replication HOW-TO

Server Status
Manager App
Host Manager

Developer Quick Start

Tomcat Setup
First Web Application
Realms & AAA
JDBC DataSources
Examples
Servlet Specifications
Tomcat Versions

Managing Tomcat

For security, access to the manager webapp is restricted. Users are defined in:

$CATALINA_HOME/conf/tomcat-users.xml

In Tomcat 9.0 access to the manager application is split between different users.
Read more...

Release Notes
Changelog
Migration Guide
Security Notices

Documentation

Tomcat 9.0 Documentation
Tomcat 9.0 Configuration
Tomcat Wiki

Find additional important configuration information in:

$CATALINA_HOME/RUNNING.txt

Developers may be interested in:

Tomcat 9.0 Bug Database
Tomcat 9.0 JavaDocs
Tomcat 9.0 SVN Repository

Getting Help

FAQ and Mailing Lists

The following mailing lists are available:

tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).

tomcat-users
User support and discussion

taglibs-user
User support and discussion for Apache Taglibs

tomcat-dev
Development mailing list, including commit messages

Répertoires et fichiers de configuration Tomcat

Voici quelques détails de la structure du répertoire d'installation de Tomcat **<$CATALINA_HOME>/** :

Partie principale

bin **startup.bat**.....: Démarrage du serveur
shutdown.bat.....: Arrêt du serveur

conf.....: Contient les fichiers de configurations du serveur
server.xml.....: Fichier de configuration principale du serveur
web.xml.....: Fichier de configuration des valeurs par défaut pour toutes les applications WEB
tomcat-users.xml……...: Fichier XML contenant les comptes pour l'administration Intranet de Tomcat

webapps.....: Répertoire de déploiement des applications WEB

Lors du déploiement d'une application JEE web par exemple, on peut déposer le fichier **WAR** (**W**eb **AR**chive) du projet dans ce répertoire.

Environnement de développement: Eclipse-JEE

Le projet Eclipse fournit un outil adéquat au développement d'applications entreprises avec la technologie JEE. Il s'agit de la version d'Eclipse pour JEE.

J'utilise dans ce cours la version **Eclipse JEE 2019** pour Windows.

Pour l'installation:

Décompressez le zip dans un répertoire de votre choix (exemple C:\eclipse)

Maintenant, vous pouvez créer un espace de travail/workspace (exemple **D:\mesprojetsJEE**)

Configurations du serveur

Vous pouvez configurer plusieurs serveurs d'application ou conteneurs web dans cet environnement. On indique à Eclipse le (s) serveur (s) d'applications à utiliser pour le projet. Ici nous utilisons le serveur d'application Apache-Tomcat.
Cela se fait dans les propriétés d'Eclipse:

1) **Windows → Preferences**
2) Dérouler **Server** et cliquer sur **Runtimes Environment**
3) Cliquez sur le bouton **Add**
4) Dans la fenêtre qui apparaît sélectionnez le serveur **(**exemple **Apache Tomcat v9.0)**
5) Faites **Next** et dans la fenêtre suivante indiquez le *répertoire d'installation* de ce serveur dans **Tomcat Installation Directory**
6) Finish.

Vus pouvez reprendre le processus à partir de l'étape 2) pour ajouter autant de serveurs différents que vous voulez.

Des options de configuration avancées peuvent être adoptées, pour cela il suffit de double-cliquer sur le serveur dans la vue « servers ».

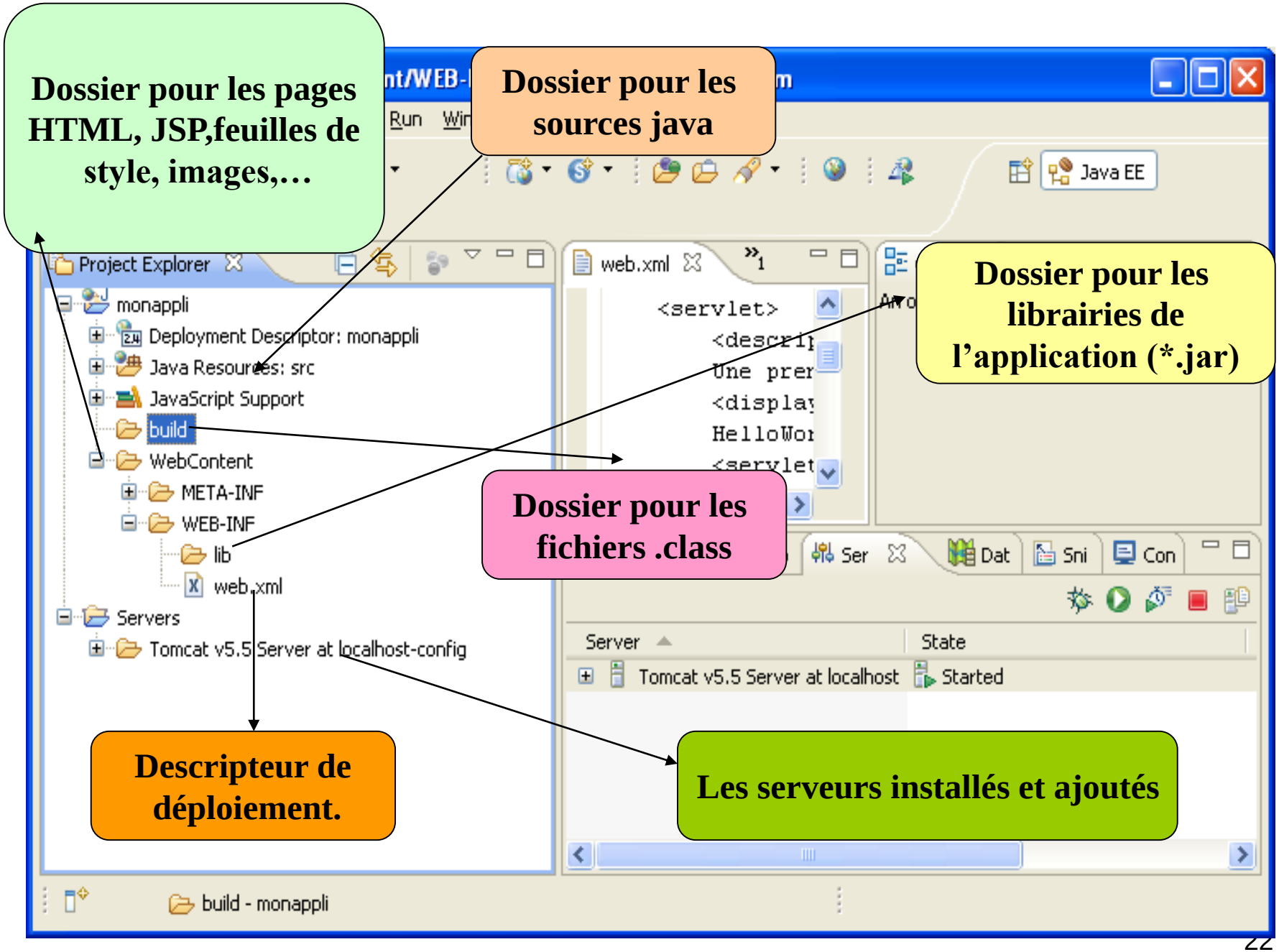
Création d'une application web dynamique

Pour développer une application web, il est nécessaire de créer un projet de type « **Dynamic Web Project** ». Ce type de projet est un projet Java possédant une organisation particulière et configuré pour fonctionner sur un serveur d'application (un de ceux qui ont été déclarés).

Voici les différentes étapes lors de la création d'un *projet web dynamique*:

1) File → New → Other
2) Clic sur le + devant **Web** et double clic sur **Dynamic Web Project**.
3) Clic sur **Next** : dans **Project name**, entrez le nom du projet: exemple: **MonAppli**.
   Sélectionnez **Apache Tomcat v9.0** dans **TargetRuntime**.
4) Faites **Next** et enfin **Finish**.

   Vous pouvez constater la structure de l'application web dynamique que vous venez de créer.
   Voici à quoi cela ressemble:


Dossier pour les pages HTML, JSP,feuilles de style, images,…
Dossier pour les sources java
Dossier pour les librairies de l'application (*.jar)
Dossier pour les fichiers .class
Descripteur de déploiement.
Les serveurs installés et ajoutés
Project Explorer
monappli
Deployment Descriptor: monappli
Java Resources: src
JavaScript Support
build
WebContent
META-INF
WEB-INF
lib
web.xml
Servers
Tomcat v5.5 Server at localhost-config
web.xml
Server
State
Tomcat v5.5 Server at localhost
Started
build - monappli
Java EE

Ajout de servlets et de pages JSP

L'ajout de servlets et de pages JSP se fait de façon très simple:

Pour l'ajout d'une servlet:

1) Faites un ***clic droit*** sur le nom du projet **MonAppli → New → Servlet**
2) Dans la fenêtre suivante donnez le **package** (**Java package**) et le **nom (Class name)** de votre Servlet.
3) Faites Next: vérifier l'url de mappage (rubrique **URL Mappings**) et enfin **Finish**.

Exécution d'une application web dynamique sous Tomcat

Pour exécuter notre application, nous devons la déployer dans un serveur d'applications comme Tomcat.
L'exécution demande plusieurs considérations:
- ajout d'un serveur d'application
- publication de l'application dans ce serveur
- démarrage du serveur
- lancement de l'application

Ajout du serveur:
1) Dans la *perspective JEE*, cliquez sur l'onglet **Servers**. La vue **Servers** affiche la liste des serveurs et permet de les gérer (arrêt, démarrage, publication, …).

Donc commençons par ajouter un serveur (ici **Tomcat**) dans cette vue:
2) Faites un clic droit dans la zone vierge et faites **New → Server**.
3) Dans la fenêtre qui apparaît: cliquez sur le serveur à ajouter: ici **Tomcat v9.0 Server**.
4) Finish.
Le serveur Tomcat vient d'être ajouter comme serveur. Maintenant, l'étape suivante consiste à la ***publication de notre application*** dans ce serveur qui va la gérer.

Publication d'une application web dynamique sous Tomcat

La publication d'une application sur un serveur consiste à ajouter le projet de l'application dans la liste des projets pris en compte par le serveur.
Voici une façon de procéder:

1) Dans le vue Servers, faites un clic droit sur le nom du serveur et ensuite faites **Add and Remove projects**…
2) Sélectionnez le (s) projet (s) à ajouter et cliquez sur **Add**.
3) Finish.

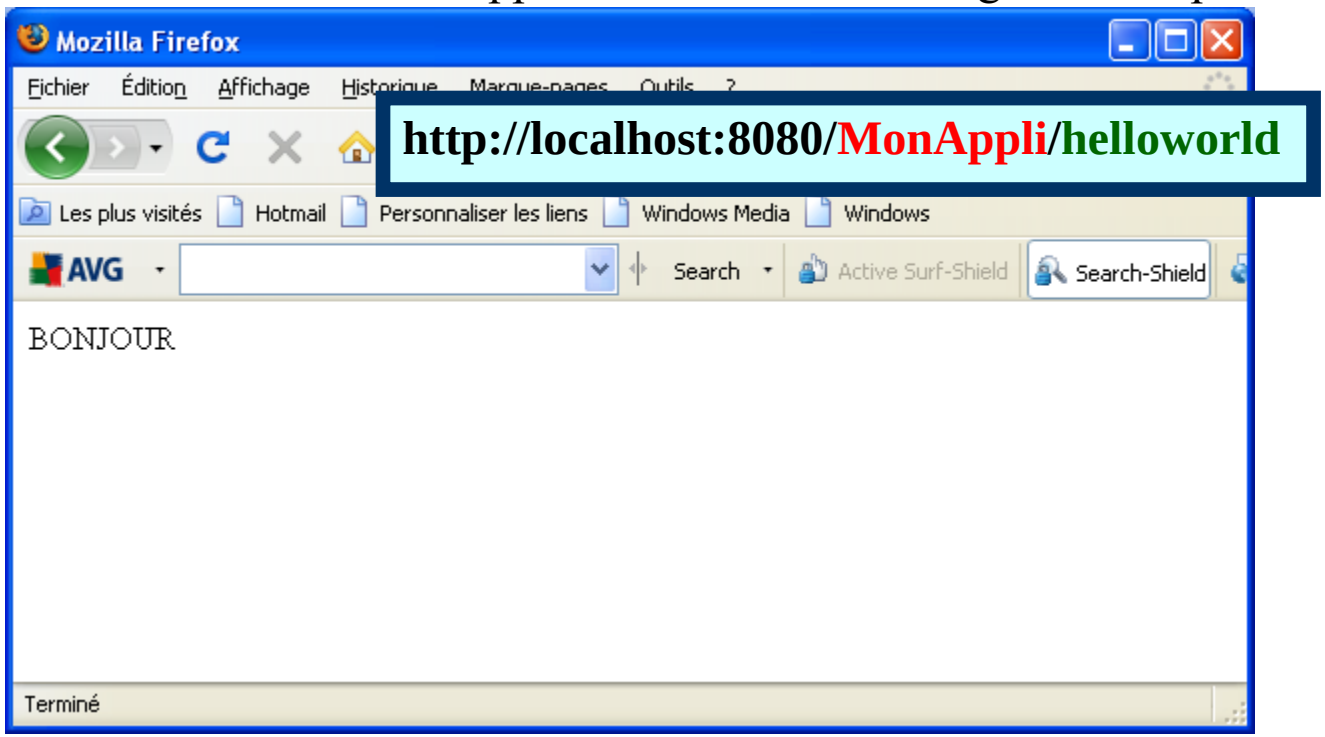
Notes: l'ajout d'une application existante peut aussi se faire pendant l'étape précédente concernant l'ajout du serveur.

Démarrage de Tomcat et lancement d'une application web dynamique.

Pour le démarrage de Tomcat, cliquez sur le bouton . Vérifier sur la vue Console que Tomcat à bien démarré.

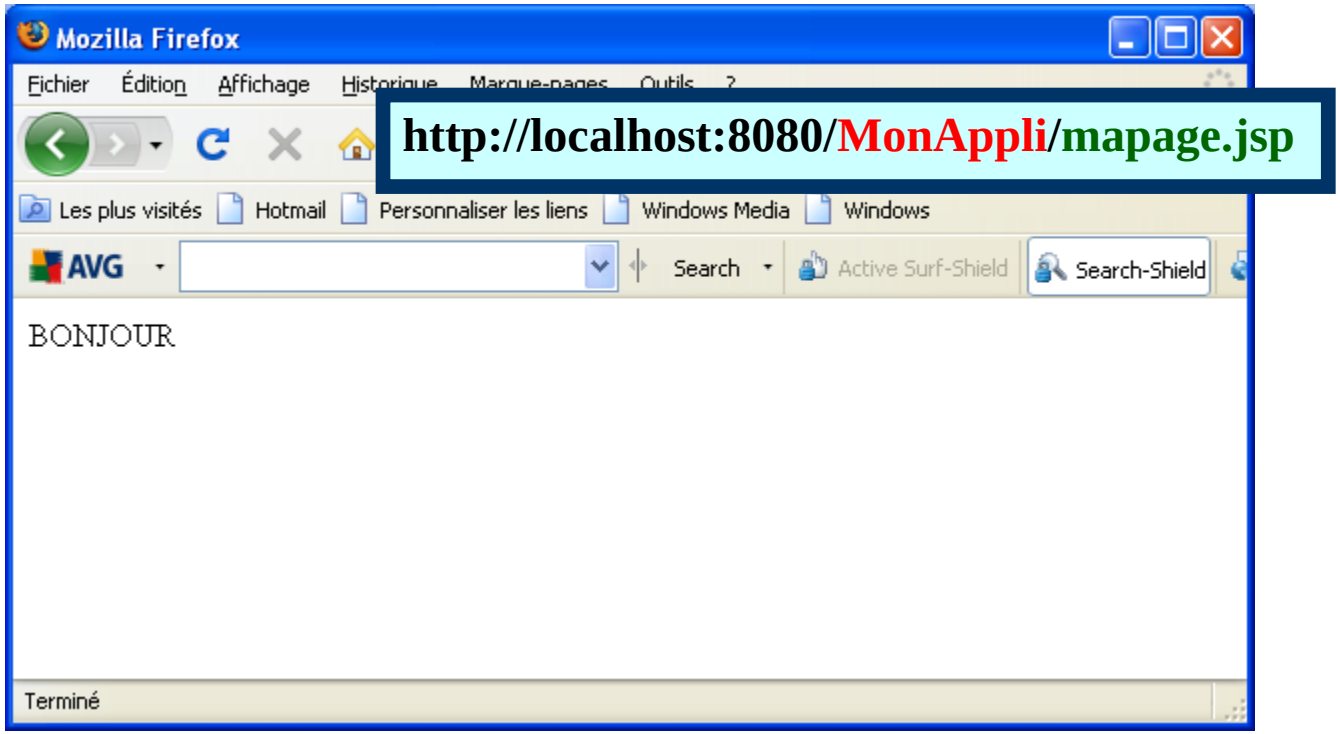
Au niveau de la vue Servers, vérifier que l'attribut **State** du serveur Tomcat est bien positionné à **Started** et que l'attribut **Status** est à **Synchronized** (cet attribut signifie que toutes les applications ajoutées sont bien configurées).

Enfin arrive le moment de lancer l'application: ouvrir un navigateur et taper:



Ajout d'une page JSP

1) Faites un clic droit sur le dossier **WebContent** et **New → JSP**
2) Dans **File name**, entrer le nom de la page JSP (exemple ***mapage.jsp***)
3) Faites **Next** et sélectionnez **New JSP File (html)**
4) Finish
5) Exécuter après remplissage du corps (body) de votre JSP.



Déploiement d'une DWP sous Tomcat

Pour déployer une application web de type ***Dynamic Web Project*** (DWP)sous Tomcat, il suffit tout simplement de l'archive sous forme d'un fichier WAR et de le déposer sous le répertoire Webapps de Tomcat.

Comment créer ce fichier WAR sous Eclipse:
Clic droit sur le nom du fichier projet (dans notre cas: MonAppli) → **Export** → **WAR file**.
C'est tout!!!

API Servlet

Présentation

Une servlet est un programme Java qui tourne sur la machine où est installé le serveur JEE, en tant qu' extension (des fonctionnalités) du serveur.

Elle reçoit une requête d'un client, effectue des traitements et lui renvoie le résultat

Elle est invoquée lorsque le navigateur client appelle l' URL associée à ce programme.

La liaison entre la servlet et le client peut être direct ou passer par l'intermédiaire d'un serveur HTTP – ce qui est plus courant- (grâce à l' URL saisie).

Une servlet fonctionne selon un modèle client/serveur ou requête/réponse, par conséquent tous les protocoles utilisant ce modèle pourront être utilisés (http, ftp, etc.)

Une servlet peut être invoquée plusieurs fois pour répondre à plusieurs requêtes simultanément.

Une servlet est écrite en Java, de ce fait elle en tire tous les avantages: portabilité, API abondante.

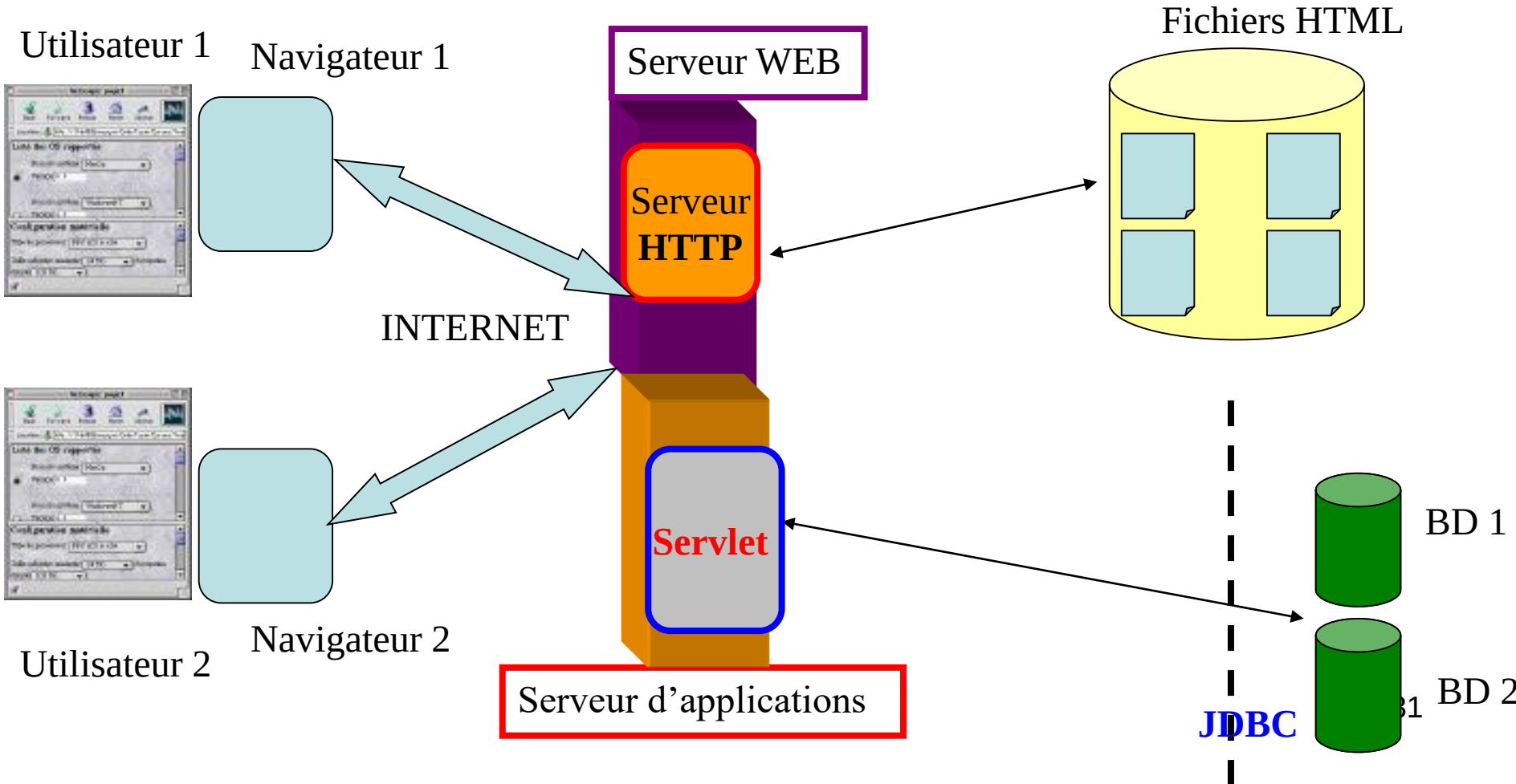
Une servlet se positionne dans une architecture trois tiers, dans le tiers du milieu, entre le client chargé de l'affichage et la source de données.

Si j'écris la servlet ***HelloWorld.java*** compilée en ***HelloWorld.class***, je peux décider que si j'appelle l'URL ***mapage*** c'est la servlet ***HelloWorld.class*** qui sera exécutée et le résultat s'affiche sur ma page de mon navigateur.

Vue générale

Les servlets sont aux serveurs ce que les applets sont aux browsers.

- **Programme Java qui sert d'extension à un serveur Web**
  - ***extension des fonctionnalités du serveur***
  - ***création et renvoi de pages HTML dont contenu le dynamique dépend de la nature de la requête du client.***



Fonctionnement des servlets (protocole HTTP)

Les servlets sont chargées et exécutées dans la JVM par un serveur d'applications. Je rappelle que ce serveur contient entre autre un moteur de servlets qui se charge de manager les servlet qu'il contient.

Pour exécuter une servlet, il suffit de saisir une URL qui désigne la servlet dans un navigateur:

1. Le serveur reçoit la requête HTTP qui nécessite une servlet
2. S'il s'agit de la première sollicitation de la servlet, le serveur l' instancie. Les servlets sont stockées sous forme de fichier **.class** dans un répertoire particulier du serveur qui dépend du serveur d'application utilisé. **La servlet reste en mémoire jusqu'à l'arrêt du serveur**. La servlet en mémoire peut être appelée par plusieurs threads lancés par le serveur pour chaque requête.
3. Le serveur crée un objet qui représente la requête HTTP et l'objet qui contiendra la réponse et les envoie à la servlet
4. La servlet crée dynamiquement la réponse sous forme de page HTML transmise via un flux dans l'objet contenant la réponse
5. Le serveur récupère la réponse et envoie la page html au client.

Comment effectuer le lien entre la servlet et l'URL http ?
→ Le fichier web.xml

Le fichier **web.xml** est le fichier le plus important d'une application JEE car il contient le paramétrage des servlets de l'application et est appelé ***descripteur de déploiement***.
Chaque servlet de l'application doit être décrite dans le fichier **web.xml**, et le lien de la servlet avec l'URL y figure également.

```
<!xml version="1.0"  encoding="UTF-8"?>

<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
   "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <servlet>
    <servlet-name> helloName </servlet-name>
    <servlet-class>  projet.servlets.HelloWorld</servlet-class>
</servlet>
            <servlet-mapping>
                <servlet-name> helloName </servlet-name>
                <url-pattern> /mapage < /url-pattern >
            </servlet-mapping>
</web-app>
```

Est nécessaire que si on appelle la servlet par une URL

La **déclaration** de la servlet se fait dans la balise XML **<servlet>:**

- **<servlet-name>** est le nom interne de la servlet, quelconque mais l'identifie de façon unique
- **<servlet-class>** est la classe Java associée à la servlet (qui doit dériver de la classe ***HttpServlet***)
  En général il s'agit du ***nom complet de la classe***.

Le **lien** entre la servlet et l'URL s'effectue grâce à l'élément **<servlet-mapping>:**

- **<servlet-name>** est le nom interne donné à la servlet identique à celui dans l'élément ***<servlet>***
- **<url-pattern>** est l'URL permettant de faire le lien avec la servlet.

A chaque ***servlet*** de l'application doit nécessairement correspondre un élément **<servlet>** dans le fichier **web.xml**, sinon la servlet n'existera pas pour le serveur d'application.

Remarque importante:

En pratique, lorsque vous développez des servlets, il faut prendre la peine **de ne jamais donner** pour **deux servlets différents** le même nom dans l'élément **<servlet-name>** des balises **<servlet>** et **<servlet-maping>.**
Cela entraînera des erreurs internes graves dans le conteneur **Tomcat qui l'empêcheront de démarrer normalement**.

Écriture de la servlet HelloWorld

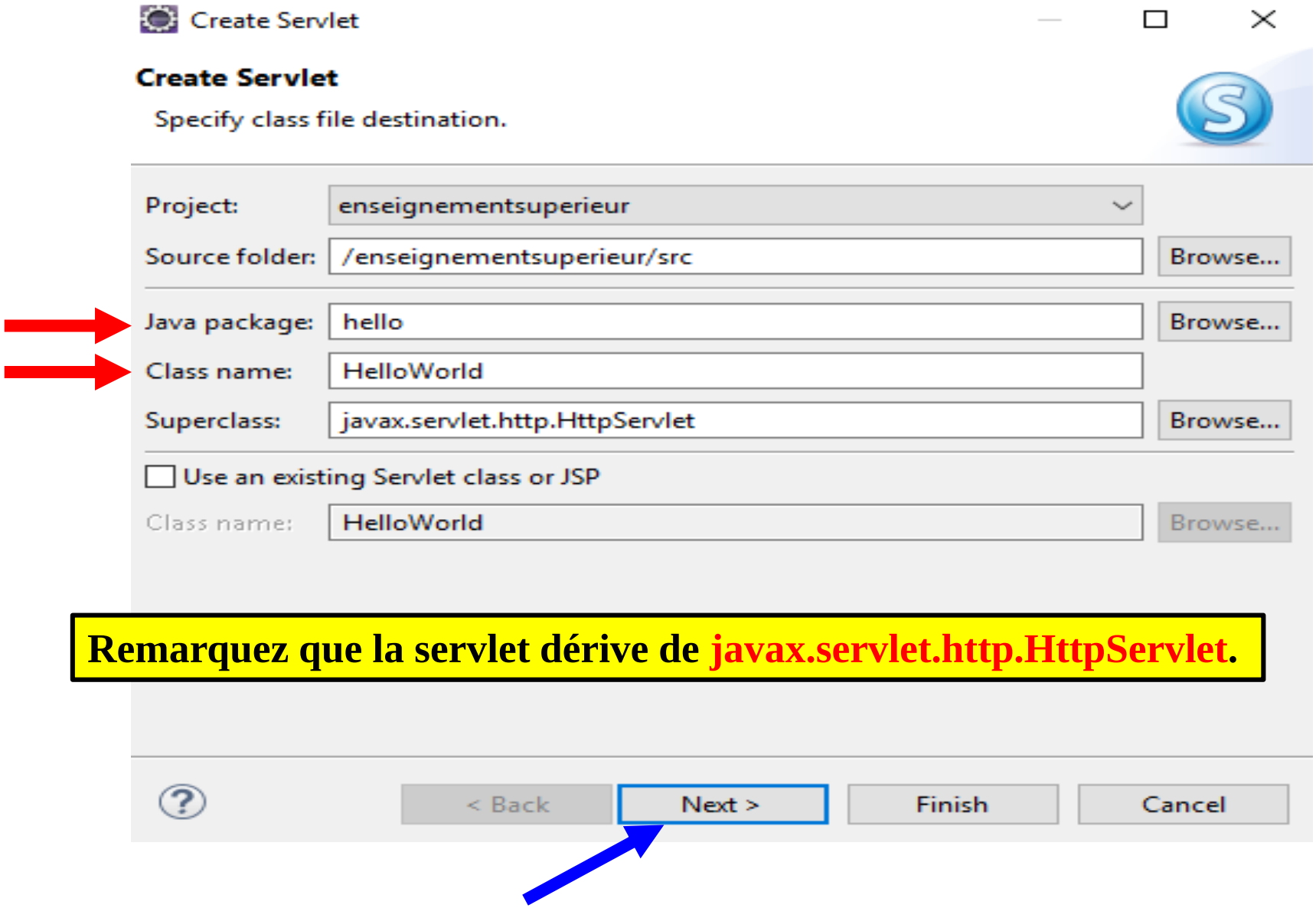
Nous avons choisi de travailler avec l' IDE Eclipse, vu sa *gratuité*, sa *robustesse* et son *extensibilité* grâce aux plugins. Pour une version classique d'Eclipse.

1. Je crée alors le projet que je nomme **servletcour**
2. Je crée la classe **HelloWorld** dans le package **joe.cours.servlets .** Cette classe dérive de la classe **HttpServlet**.

Dans ce cours on utilisera Eclipse-JEE qui ne nécessite pas cette configuration. On fera la configuration à partir du serveur d'application.

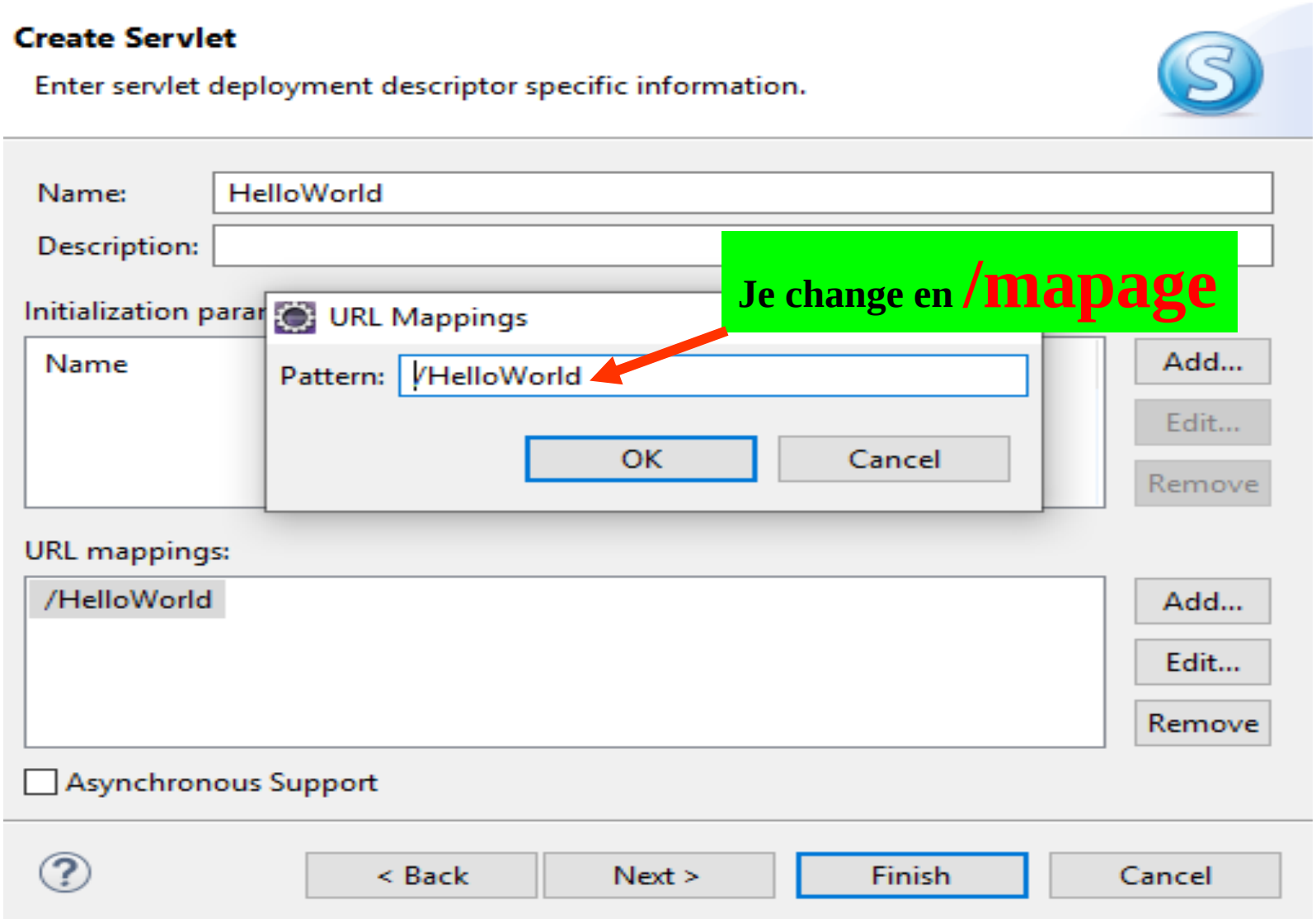
Création sous Eclipse de la Servlet HelloWorld

Clic droit sur le nom du DWP projet → New Servlet



Remarquez que la servlet dérive de javax.servlet.http.HttpServlet.

L'URL pattern de la servlet est par défaut le nom de la classe, vous pouvez le modifier en double-cliquant sur l'Url et modifier.



Create Servlet

Create Servlet

Specify modifiers, interfaces to implement, and method stubs to generate.

Modifiers: ☑ public ☐ abstract ☐ final

Interfaces:

Add...

Remove

Je prends en compte uniquement la méthode doGet.

Which method stubs would you like to create?

- ☐ Constructors from superclass
- ☑ Inherited abstract methods

☐ init	☐ destroy	☐ getServletConfig
☐ getServletInfo	☐ service	☑ doGet
☐ doPost	☐ doPut	☐ doDelete
☐ doHead	☐ doOptions	☐ doTrace

< Back | Next > | Finish | Cancel

Écriture de la servlet HelloWorld

```
package joe.cours.servlets;
import java.io .*;
import javax.servlet .*;
import javax.servlet.http .*;
@WebServlet("/mapage")
public class HelloWorld        extends HttpServlet        {

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

Compilation et exécution de la servlet HelloWorld

Dans les anciennes versions de Apache Tomcat (exemple v. 5), il fallait absolument configurer la servlet dans le descripteur de déploiement **web.xml**:

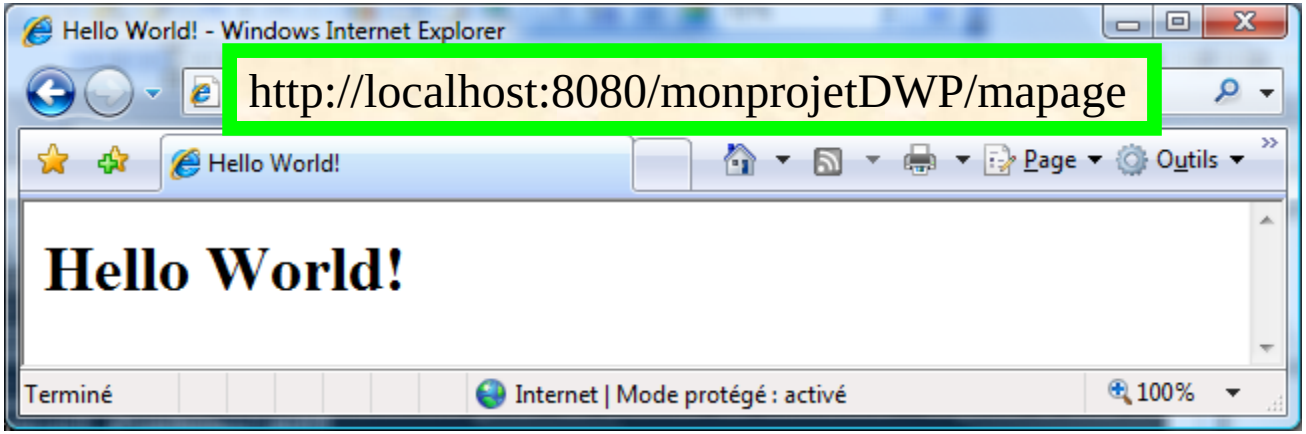
```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<servlet>
      <servlet-name> helloName </servlet-name>
      <servlet-class> joe.cours.servlets.HelloWorld </servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name> helloName </servlet-name>
      <url-pattern> /mapage </url-pattern>
   </servlet-mapping>
<!-- JOE HelloWorld servlet mappings end -->
```

Cette configuration est maintenant remplacée par l'annotation @WebServlet("/mapage").

4. Pour exécuter votre servlet, il suffit maintenant de saisir l'URL suivante sur votre navigateur : http://localhost:8080/monprojetDWP/mapage

Et la sortie donne ceci:

Il est évident que ceci ne peut se faire que si le serveur JEE tourne correctement.



Détails du Contenu de la servlet HelloWorld

Une servlet est une classe qui implémente ***l'interface* javax.servlet.Servlet** qui est l'interface de **base** pour générer des ***servlets indépendantes de tout protocole***.
Le fait d'implémenter cette interface oblige de redéfinir toutes ses méthodes.
C'est pourquoi Java a introduit la classe **GenericServlet** qui implémente **Servlet** et permet également de produire des servlets génériques (indépendantes de tout protocole).
Donc on peut dire aussi *qu'une servlet est une classe dérivée* de **GenericServlet**.

Mais de façon générale, votre classe étendra la classe **javax.servlet.http.HttpServlet** (elle-même basée sur **GenericServlet**).
La classe **HttpServlet** est définie pour gérer les requêtes HTTP dans tous leurs aspects (analyse des paramètres, gestion des cookies et support des sessions).

Le paquetage **javax.servlet** contient les classes et interfaces pour générer des servlets génériques indépendantes de tout protocole.
Le paquetage **javax.servlet.http** contient les classes pour développer des servlets qui reposent sur le protocole HTTP utilisé par les serveurs web.
Le paquetage **java.io** contient la classe **PrintWriter**, les autres classes de flux et la classe d'exception **IOException**.

La classe **HttpServlet** possède plusieurs méthodes que vous pouvez redéfinir pour implémenter certaines fonctionnalités. Les quatre méthodes les plus utilisées sont:

*/*gère une simple requête GET pour une page*/*
protected void doGet (HttpServletRequest req, HttpServletResponse resp) throws IOException

*/*gère les formulaires échangés avec le serveur par la méthode HTTP POST*/*
protected void **doPost** (HttpServletRequest req, HttpServletResponse resp) throws IOException

*/*appelée par le conteneur de servlets (Tomcat) lors de la création d'une instance de la servlet les informations sur le contexte de la servlet sont passées sous forme de paramètres sous la forme de l'objet ServletConfig. Toute opération de configuration importante (comme l' établissement d'une connexion à une base de données) s'effectue dans cette méthode.*
Elle sera appelée par le conteneur de servlet avant que celui-ci envoie la première requête à l'une des méthodes de gestion doGet ( ) ou doPost ( )/*
public void **init** (ServletConfig config) throws ServletException

*/*Est chargée d'envoyer les requêtes vers la méthode de gestion appropriée; c'est elle qui appelle les méthodes doGet ou doPost. Mais il est conseillé d'utiliser directement ces méthodes. */*
public void **service** (ServletRequest req, ServletResponse resp) throws IOException, ServletException

Dans notre exemple, nous avons défini la méthode doGet ( ), toutes les autres sont là mais elles sont héritées de la classe de base ***HttpServlet***.

Les servlets ne renvoient pas uniquement du code HTML; elles peuvent produire tout type de données, comme les images, le son, ou même des applets Java.
Cependant la majorité renvoie du HTML.
Donc il faut indiquer le type de données renvoyées en utilisant le type MIME des données.
Pour du HTML le type MIME est **text /html**. Voici comment l'indiquer:

response.setContentType (text/html) ;
Maintenant l'instruction **PrintWriter = response.getWriter ( );** renvoie un flux (ici de type texte) pour envoyer la réponse au client. Avec d'autres types de données (binaires ou caractères), différents types de classes de flux ***Writer*** et ***Stream*** sont nécessaires.

Les dernières instructions sont toutes des instructions **println ( )** qui envoient directement du texte à la réponse.

Remarque:
Une manière de lancer notre servlet sans passer par la configuration d'un fichier descripteur de déploiement est de copier votre servlet dans un ***contexte déjà enregistré***.
On peut faire ceci:
<$CATALINA_HOME>/webapps/ servlets-examples/WEB-INF/classes /joe/cours/servlets/ HelloWorld.class

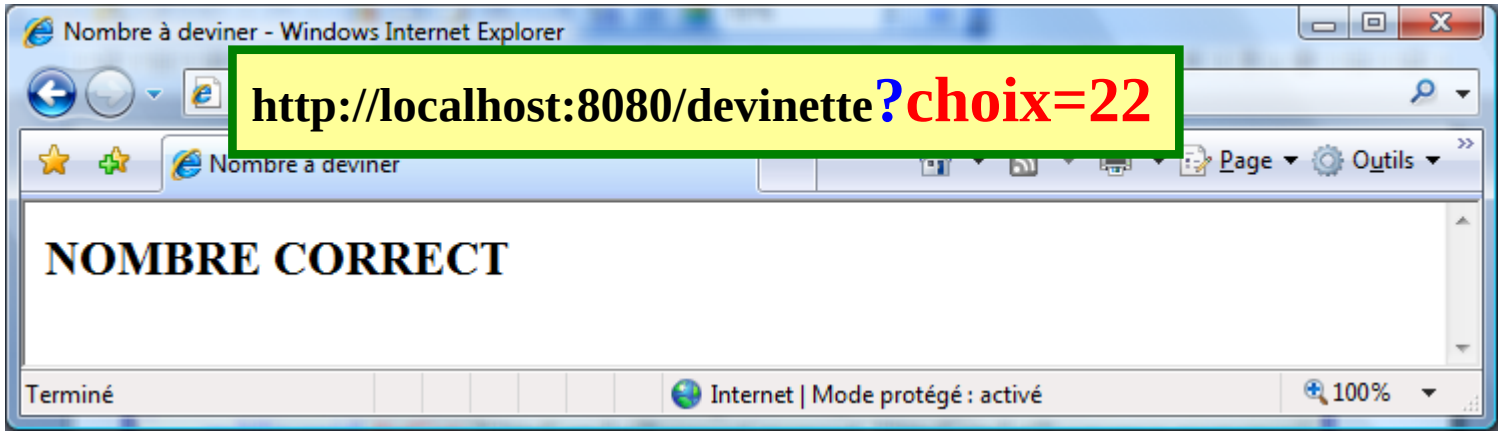
Et sur navigateur tapez http://localhost:8080/servlets-examples/servlet/ joe.cours.servlets.HelloWorld

Deuxième servlet DevinetteServlet

Dans ce deuxième exemple, nous voulons montrer la différence entre **doPost** et **doGet**.

```
import java.io .*;   import javax.servlet .*;  import javax.servlet.http .*;
public class DevinetteServlet extends HttpServlet{
      public void doGet (HttpServletRequest request, HttpServletResponse response)
                                                    throws IOException, ServletException {
{  response.setContentType ("text/html");
     PrintWriter out = response.getWriter ( );
     out.println ("<html>");
     out.println ("<head>");
     out.println ("<title>Nombre à deviner</title>");
     out.println ("</head>");
     out.println ("<body>");
           int number =22;
           String tatonne = request.getParameter ("choix");
            int saisie = new Integer (tatonne).intValue ( );
 if (saisie > number)              out.println ("<H2>NOMBRE TROP ELEVE</H2>");
 if (saisie < number)              out.println ("<H2>NOMBRE TROP BAS</H2>");
 if (saisie = = number)             out.println ("<H2>NOMBRE CORRECT</H2>");
    out.println ("</body>"); out.println ("</html>");
    }          }
```

Avec la méthode **doGet**, les ***paramètres*** de la méthodes apparaissent dans l'URL de lancement de la servlet.
Ici on a ***un seul paramètre*** récupéré dans le code de la servlet grâce à la méthode ***getParameter (String paramName).***
Si vous avez plusieurs paramètres séparez les par une esperluette (&).



Au lieu d'utiliser la méthode **doGet**, le développeur peut choisir d'utiliser plutôt la méthode **doPost**. Dans ce cas, pour appelez la servlet vous devez disposer d'un formulaire HTML.
Les formulaires sont le système de base pour envoyer des informations à un serveur.
Ce formulaire devra être placé dans le répertoire racine de votre application web.
L'un des intérêts de cette façon est de cacher les paramètres à l'utilisateur.

Voici le formulaire qui vous permettra d'appeler votre servlet.

```
<html>
        <head>
                <title>Formulaire pour ma servlet DevinetteServlet </title>
        </head>
<body>
        <h1> Formulaire de devinette de nombre </h1>
        <form method ="post"  action = " /devinette" >
                <p> Entrer un nombre: <input type ="text" name = "choix"> </p>
                <p> <input type ="submit"  name = "Envoyer" value = "Envoyer"> </p>
        </form>
</body>


</html>
```

DevinettePost.html

NB: ce fichier est placé à la ***racine de mon application web***:

C:\tomcat5.0\webapps\ROOT\DevinettePost.html

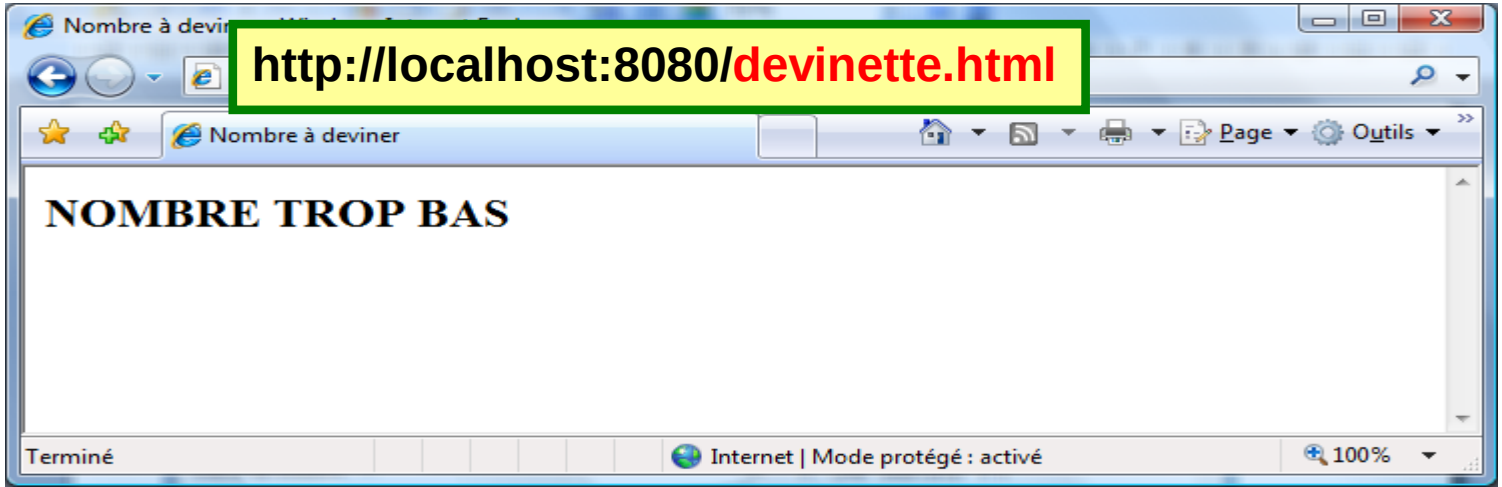
En général, vous utiliserez une application web différente de ROOT comme l'application web *joecontexte*. Dans ce cas mettez action= " /joecontexte/devinette"

Pour le l'appel de la servlet dans mon navigateur:

1. Je saisis 11

2. Je valide

3. j'obtiens ceci



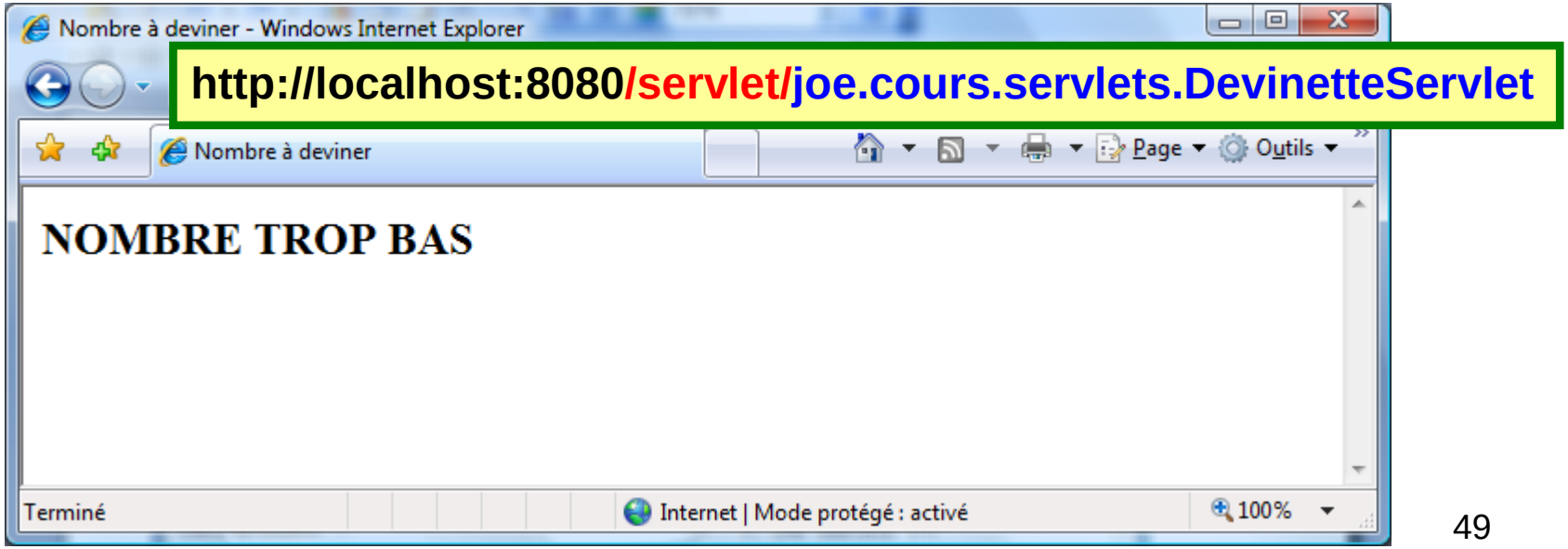
Remarque

Au niveau de votre formulaire c'est à dire le fichier ***DevinetteServlet***,
vous pouvez remplacer:

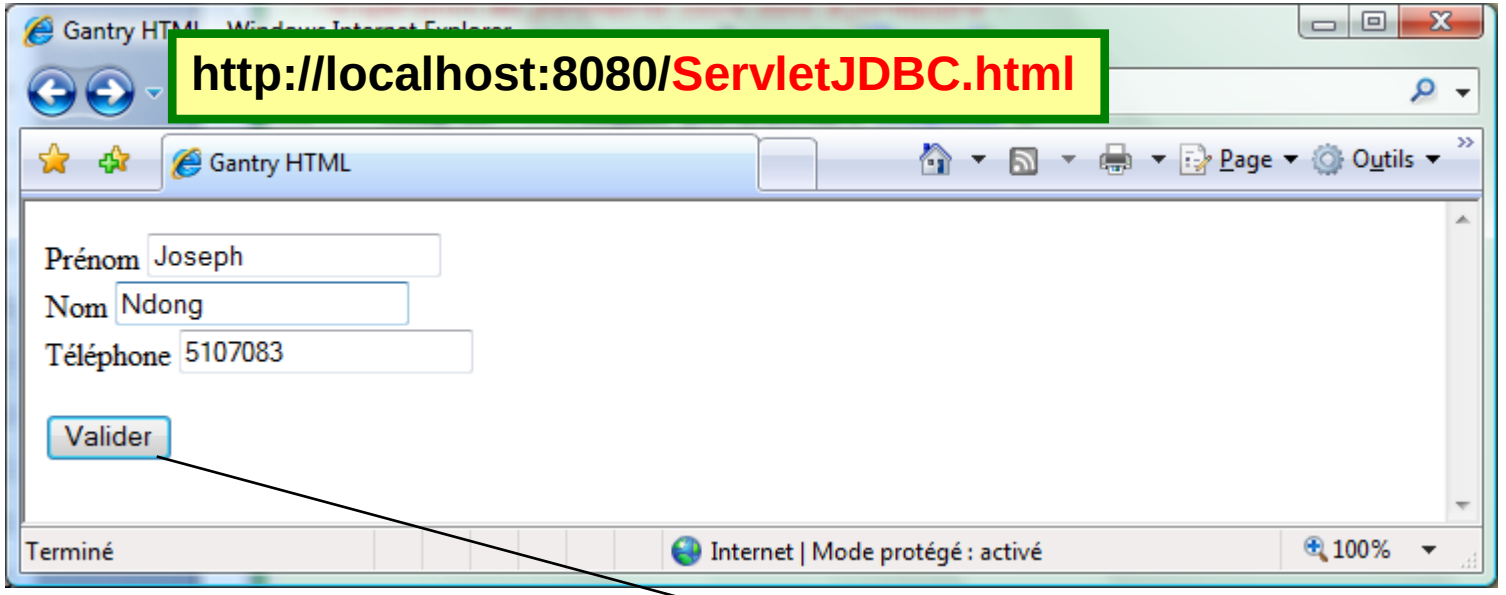
action = **" /devinette"** par

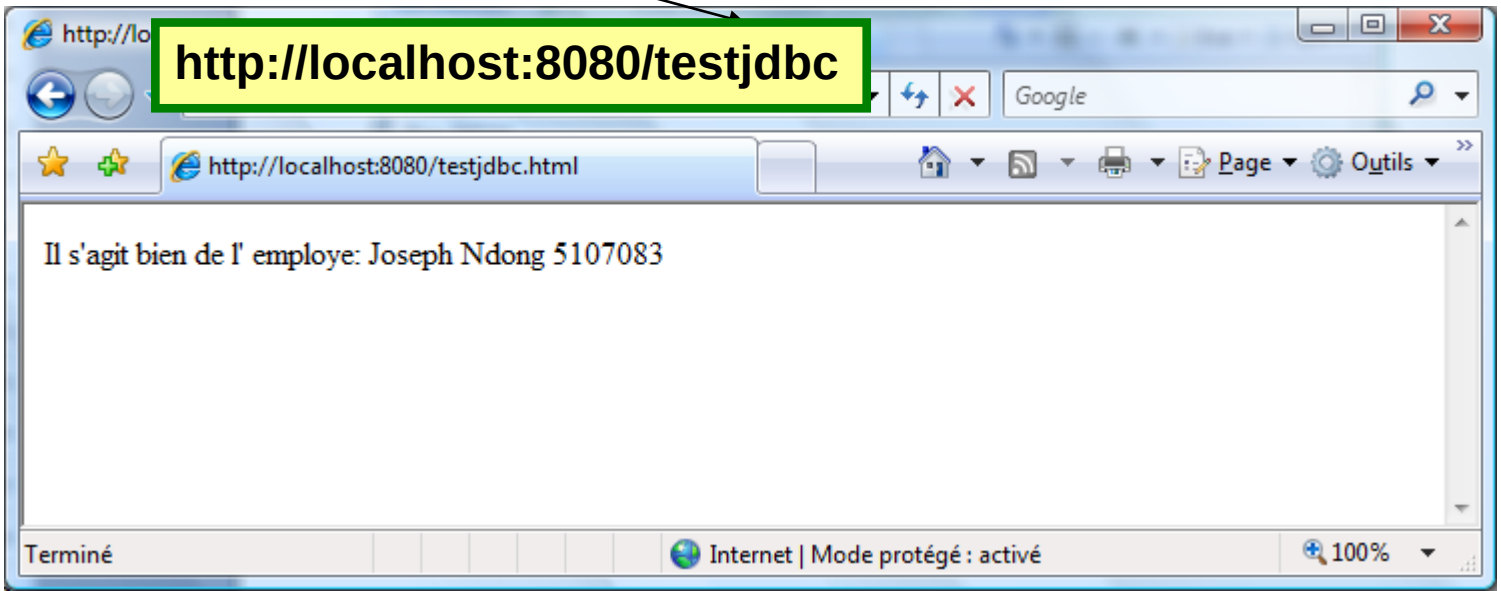
action = **"servlet/joe.cours.servlets.DevinetteServlet "**

Dans ce cas vous aurez à taper l' URL suivante:



Exécution et appel de la servlet





L'API servlets :classes et interfaces

L'API servlet est une extension du JDK de base, elle regroupe des classes et interfaces contenues dans des packages préfixés par « javax ».

- ***javax.servlet*** contient les classes et interfaces pour produire des servlets génériques indépendantes de tout protocole
- ***javax.servlet.http*** permet de développer des servlets basées sur le protocole HTTP utilisé par les serveurs web.

Il existe quelque principes fondamentaux sur le fonctionnement général d'une servlet qu'il est de savoir et de retenir:

Une servlet reçoit une requête provenant ***soit d'un navigateur***, ***soit d'une autre servlet***, effectue **un** traitement puis fournit en sortie ***soit une réponse vers le navigateur***, soit elle envoie (fait un ***forward***) ***vers une autre servlet*** qui continue le traitement.

La requête provenant d'un navigateur peut avoir trois origines différentes:

- Une URL dont on a tapé l'adresse dans la barre d'adresse du navigateur
  ex: http://nomServeur/cheminApplicationWeb/cheminServletWeb
- Un lien sur lequel on a cliqué dans une page HTML
- Un clic sur un bouton de formulaire HTML (bouton de type ***submit***)

Les servlets sont assujetties à des flux incessants d' arrivée de requêtes provenant d'utilisateurs divers et chacun d'eux peut à volonté invoquer la même servlet plusieurs fois.

C'est pourquoi les notions de ***session*** et ***d'application*** ont été créées pour faciliter la tâche du développeur.

- **session** : c'est toute l'activité mise en oeuvre par le ***même*** utilisateur pendant un temps donné.
  La servlet peut invoquée plusieurs fois pendant cet intervalle de temps (paramétrable).
  Il y 'a autant de sessions qu'il y a d'utilisateurs. Et les servlets se partagent la session de chaque utilisateur.
- **application**: concerne toute l'activité mise en œuvre par **tous** les utilisateurs.
  Il existe une **seule** application **commune** à toutes les servlets mises en œuvre.

Classes et interfaces du package javax.servlet

Les interfaces

- ***RequestDispatcher***: permet d'effectuer un forward vers une autre servlet, une JSP,… afin de lui donner le contrôle,
- ***Servlet***: définition de base d'une servlet,
- ***ServletConfig***: définition d'un objet pour configurer la servlet
- ***ServletContext***: pour l'obtention des informations sur le contexte d'exécution de la servlet,
- ***ServletRequest***: définition d'un objet contenant la requête du client
- ***ServletResponse***: définition d'un objet contenant la réponse renvoyée au client
- ***SingleThreadModel***: permet de définir une servlet qui ne peut répondre qu'à une seule requête à la fois

Les classes

- ***GenericServlet***: pour produire des servlets indépendantes de tout protocole
- ***ServletInputStream***: flux permettant la lecture de données de la requête du client
- ***ServletOutputStream***: flux pour l'envoie de réponse au client

Les exceptions

- ***ServletException***: exception générale en cas de problème durant l'exécution de la servlet
- ***UnavailableException***: exception levée si la servlet est indisponible.

Classes et interfaces du package javax.servlet.http

Ce package contient les classes et interfaces intéressantes pour développer des servlets utilisant le protocole HTTP.

Les interfaces

- ***HttpServletRequest***: hérite de ServletRequest et définit un objet contenant la requête du client selon le protocole HTTP,
- ***HttpServletResponse***: hérite de ServletResponse et définit un objet contenant la réponse renvoyée au client selon HTTP,
- ***HttpSession***: permet de définir un objet représentant une session

Les classes

- ***Cookie***: représente un cookie (données sauvées par le browser sur le poste client) selon le protocole HTTP,
- ***HttpServlet***: hérite de GenericServlet: classe définissant une servlet utilisant HTTP renvoyée au client selon HTTP,
- ***HttpUtils***: classe proposant des méthodes statiques pour la production de servlet HTTP.

Méthodes de l'interface de base javax.servlet.Servlet

Une servlet est une classe qui implémente l'interface Servlet, laquelle définit 5 méthodes qui permettent au conteneur web de dialoguer avec la servlet. ***Elle encapsule les méthodes nécessaires à la communication entre le conteneur et la servlet***.

public void **init** (ServletConfig config) throws ServletException
public void **service** (ServletRequest req, ServletResponse resp) throws IOException, ServletException
*/*renvoie l'objet ServletConfig passé à la méthode init */*
public **ServletConfig getServletConfig** ( ) throws ServletException
*/*appelée lors de la destruction de la servlet. Elle permet de libérer proprement certaines ressources (fichiers, base de données ...). C'est le serveur qui appelle cette méthode. Cette méthode ne s'exécute qu'une seule fois au cours du cycle de vie de la servlet*/*
public void **destroy** ( ) throws ServletException
*/*renvoie des informations sur la servlet*/*
public String **getServletInfo** ( ) throws ServletException

Cycle de vie d'une servlet

Les méthodes init ( ), service ( ) et destroy ( ) assurent le cycle de vie de la servlet:

- ***init ( ):*** appelé par le serveur juste après l'instanciation de la servlet.
- ***service ( ):*** appelée pour le traitement d'une requête, mais ne peut être invoquée tant que la méthode ***init ( )*** n'a pas terminée. Cette méthode appelle ***doGet ( )*** ou ***doPost ( )***
- ***destroy ( )*** est appelée juste avant que le serveur ne détruise la servlet, cela permet de libérer des ressources allouées dans la méthode ***init ( )*** tel qu'un fichier ou une connexion vers une base de données

En résumé:

1. ***Une seule*** fois **init ( )**
2. ***Une ou plusieurs*** fois **service ( )**: **doGet ( )** ou **doPost ( )**
3. ***Une seule*** fois **destroy ( )**

Méthodes de la classe de base javax.servlet.http.HttpServlet

Comme déjà indiqué plus haut, cette classe gère l'arrivée des requêtes clientes.
Les requêtes peuvent être de deux types: **GET** ou **POST**, selon la méthode utilisée pour transmettre les paramètres de formulaires associés à la requête.
Les paramètres d'une requête correspondent aux valeurs saisies dans les **formulaires HTML** et qui transmises à la servlet pour traitement lorsque l'utilisateur clique sur le **bouton de validation du formulaire**.

❖ **Méthode GET**: les paramètres sont transmis dans l'URL d'appel de la servlet, à la suite

ex: **http://localhost:8080/test/premier?prenom=Joseph&nom=Ndong**

les paramètres correspondent aux champs nom et prenom dont les valeurs respectives sont "*Joseph*" et "*Ndong*«

L'inconvénient de cette méthode est que l'utilisateur voit les paramètres en clair.

Par contre elle est utile pour le programmeur qui peut ainsi vérifier si les données sont correctes lors de la transmission,

❖ Méthode **POST**: les paramètres sont transmis dans l'en-tête de la requête HTTP et sont donc invisible dans la barre d'adresse du navigateur

ex: http://localhost:8080/test/premier

Le choix de **GET** ou **POST** ***ne s'effectue que dans les formulaires HTML*** (avec l'attribut ***method*** de l'élément ***<form>***). Partout ailleurs la méthode est **GET** (lorsqu'on clique sur sur un lien, ou lorsqu'on saisit l'URL d'appel de la servlet dans le navigateur)

L'activité principale de la servlet consiste au traitement associé à un GET ou à un POST.

La classe ***HttpServlet*** contient les méthodes:

- **void doGet (request, response)**
- **void doPost (request, response)**
- **void init ( )**
- **void destroy ( )**
- **ServletContext getServletContext ( ) utilisée pour avoir accès à l'application globale**

Méthodes de l'interface javax.servlet.http.HttpServletRequest

Elle encapsule la requête reçue et contient les informations qui permettent à la servlet de faire son traitement.

REQUETE → - paramètres de formulaire
- attributs de requête
- cookies
- session

Voyons de plus près les informations auxquelles donne accès à la requête:

✓ **les paramètres éventuels saisis dans un formulaire HTML** (si la servlet est invoquée à travers un formulaire). Pour récupérer ces paramètres de formulaire, utilisez ma méthode ***public String getParameter (String nomParamètre).***

Vous utiliserez la méthode ***String [ ] getParameterValues (String nom)*** si un paramètre à plusieurs valeurs et la méthode ***Enumeration getParameterNames ( )*** pour avoir le nom de tous les paramètres.

✓ **les attributs éventuels associés à la requête**: ces attributs doivent être positionnés par le développeur. Ils servent à transmettre des informations entre servlets par l'intermédiaire des requêtes.

Un attribut possède un **nom**, et correspond en *mémoire à un objet de classe dérivée* de ***Object***.

Pour positionner un attribut, utilisez la méthode ***void setAttribute (String nom, Object objet)*** et pour le récupérer servez vous de la méthode ***Object getAttribute (String nom)***.

ATTENTION: les *attributs* sont bien différents des *paramètres*.

- ✓ **les cookies éventuels associés à l'ordinateur de l'utilisateur**: les cookies sont des données stockées sur le disque dur de l'utilisateur, qui permettent en général de conserver des données propres à cet utilisateur (par exemple son e-mail pour qu'il n'ait pas à le ressaisir à chaque fois qu'il se connecte à ce site).
  Les cookies sont propres à chaque site.
  Un nouveau cookie est stocké lorsque la servlet renvoie une réponse au navigateur par ***HttpServletResponse.addCookie ( )*** et est récupéré par ***HttpServletRequest.getCookies ( )***. [***Cookies [ ] getCookies ( )***]
- ✓ **la session de l'utilisateur**: il y a toujours une session ouverte pour chaque utilisateur. Elle est récupérée sous forme d'objet par la méthode ***HttpServletRequest.getSession*** ( ) [***HttpSession getSession ( )***].

Remarque :il n'existe pas de méthode ***setAttribute ( )***! Un paramètre est positionné de façon automatique par la validation d'un formulaire et non par le programmeur qui ne peut que le récupérer.

Vous pouvez utilisez aussi les méthodes:
- ***String getRemoteAddr ( )*** pour connaître l'adresse IP du client qui a envoyé la requête.
- ***RequestDispatcher getRequestDispatcher (String path)*** : pour obtenir un objet *RequestDispatcher* qui permettra de faire un *forward* vers la servlet indiquée dans *path*.

Nous donnerons à la suite un exemple d'utilisation de ces méthodes.

Méthodes de l'interface javax.servlet.http.HttpServletResponse

Cette interface permet de fournir la réponse vers le navigateur effectuant une requête. Cette réponse va permettre essentiellement d'écrire du texte sur l'écran du navigateur.

Pour ce faire, récupérer d'abord le flux de sortie qui permet d'écrire vers le client par :
PrintWriter out = response.getWriter ( )
Et pour écrire faire des **out.print (chaîne)**.
Il faut aussi indiquer le type de format utilisé pour le texte envoyé (c'est ce qu'on appelle le type MIME), généralement il s'agit du HTML:
response.setContentType ("text/html")

Méthodes de l'interface javax.servlet.RequestDispatcher

Un objet RequestDispatcher s'obtient grâce à l'appel de la méthode ***HttpServletRequest.getRequestDispatcher ( )***.

Exemple: si la servlet ***Servlet1*** est mappée sur ***index***, on peut faire:

RequestDispatcher rd = request.getRequestDispatcher ("/index"). 1

L'interface ***RequestDispatcher*** contient deux méthodes: ***forward ( )*** et ***include ( )***.

- ***void forward (request, response)***: permet de donner le contrôle à la servlet indiquée dans 1

  ATTENTION: le flux de sortie produit par la servlet lors de ***getWriter( ).print ( )*** est supprimé, ce qui signifie que tout ce qui avait été envoyé en réponse vers le navigateur est perdu. *Seul le flux de sortie de la servlet vers laquelle on* ***forward ( )*** *sera pris en compte*.

- ***void include (request, response)***:permet de faire une inclusion dynamique d'une servlet: le contrôle de la servlet est donné à la servlet indiquée dans le **getRequestDispatcher (/servlet1/servlet2)** puis revient à la première ***servlet1*** lorsque ***servlet2*** est terminée (c'est comme un appel de sous programme).

Contrairement à *forward ( )* les flux de sortie se cumulent dans l'ordre où les servlets sont exécutées.

Méthodes de l'interface javax.servlet.http.HttpSession

Un objet de cette interface est fournie par la méthode ***getSession ( )*** vue précédemment.
Cet objet **session** correspond à l'utilisateur courant pour ce site.

La notion **d'utilisateur** est différent selon les navigateurs.
Internet Explorer considère que deux sessions de navigation ouvertes sur le même ordinateur sont deux sessions différentes (sauf si la deuxième session est ouverte à partir du même navigateur par Fichier → Nouveau→ Fenetre).
Quant à ***Nescape***, toutes les fenêtres de navigation ouvertes sur le même ordinateur font partie de la même session.
Mais quelque soit le navigateur, deux sessions ouvertes sur deux ordinateurs sont forcément différents puisque les utilisateurs sont différents.
Les informations communes à la session d'un utilisateur sont appelées des ***attributs***.
Il y a trois types d'attributs: de *requête*, de *session* et *d'application*.
La différence entre les attributs de *requête* et les attributs de *session* est la ***portée***.
Les attributs de requête ne sont valables que dans le contexte de la *requête* reçue (et transmissibles par ***forward ( )***), alors que les attributs de session sont valables dans toute la *session*, donc dans toutes les pages et servlets du site (pour un utilisateur donné!).

Durée de vie d’une session:

Elle est définie par le temps maximum autorisé entre deux requêtes au site (pour un utilisateur donné bien sûr).
Par exemple si l’utilisateur n’accède pas au site pendant 15 minutes, sa session est terminée. Ses attributs de session ne sont pas accessibles, mais de nouveaux attributs (portant le même nom) seront créés lors de la nouvelle session qui démarrera à sa prochaine requête sur le site.

void setAttribute (String nom, Object objet): pour positionner un attribut qui sera visible dans toute la session. On dit que l'attribut a une portée session.

Object getAttribute (String nom): pour récupérer la valeur d'un attribut positionné dans la session

Enumeration getAttributeNames ( ): pour récupérer la liste de tous les attributs positionnés dans la session.

void removeAttribute (String nom): suppression de l'attribut indiqué dans la session

Méthodes concernant la durée de vie d'une session:

void setMaxInactiveInterval (int nbSec): permet d'indiquer le nombre de secondes maximum autorisé entre deux requêtes de cet utilisateur. A l'issue de cet écoulement de temps, s'il n'y a pas eu de requêtes de cet utilisateur, sa session est clôturée.

Pour indiquer qu'une session ne termine jamais: ***setMaxInactiveInterval (-1):***

Remarque: *il n'y a aucun moyen de savoir qu'un utilisateur s'est déconnecté (en fermant son navigateur par exemple). On ne peut constater que ce dernier n'a plus effectué de requêtes au site depuis x secondes, et en déduire que sa session est terminée, même si son navigateur reste toujours ouvert.*

D’autres méthodes intéressantes de la classe ***HttpServlet*** sont données ci-dessous:

String getId ( ): identificateur de la session de l’utilisateur

long getCreationTime ( ): obtenir l’heure de la création de cette session (en millisecondes depuis le 1er Janvier 1970 à minuit).

long getLastAccessTime ( ): permet d’obtenir l’heure de la dernière requête effectuée pour cette session (en millisecondes depuis le 1er Janvier 1970 à minuit).

ServletContext getServletContext ( ): utilisé pour avoir accès à l’application globale

Concernant l'interface ServletContext, elle permet d'accéder à l'application dans sa globalité indépendamment des requêtes et des sessions.
On s'en sert pour définir des variables qui ont une portée globale (générale) pour tout le site.
Ces variables sont des attributs de type application et sont visibles pour tous les utilisateurs quelles que soient les sessions.
Si un attribut d'application est créé à un moment donné, il devient accessible pour tous les utilisateurs, jusqu'à ce que l'un d'eux éventuellement le détruise.
S'il est modifié par un utilisateur, il l'est pour tous les utilisateurs.
On peut obtenir un objet ServletContext de deux manières:
- par **HttpServlet.gerServletContext ( )**
- par **HttpSession.getServletContext ( ).**

La manière la plus utilisée est **HttpServlet.gerServletContext ( )** car on est toujours dans une servlet lorsqu'on veut accéder à ***getServletContext ( ).***
Gestion des paramètres d'initialisation des servlets:
L'interface ServletContext peut servir à lire des paramètres d'initialisation communs à toutes les servlets de l'application. Ces paramètres sont stockés dans le descripteur de déploiement ***web.xml***, et concernent des données générales à lapplication.
Ces paramètres d'initialisations sont lus grâces aux méthodes: ***getInitParameter (nomParam)*** ou ***getInitParameterNames ( ).***
NB: *Pour obtenir la liste des fichiers et répertoires de l'application sur le serveur:*
Set getRessourcePaths (path)
Un exemple de servlet manipulant des paramètres d'initialisation a été donné avec l'exemple *ServletJDBC*.

Exercices

Exercice 1: *calcul de factorielles de nombres entiers.*

Création d'une servlet pour le calcul de factorielle.
Demande à l'utilisateur de saisir un nombre positif sur un champ de texte et renvoie le factorielle de nombre par validation d'un bouton VALIDER du formulaire.

Exercice 2: *gestions de plusieurs boutons de formulaires et JDBC*

On dispose d'une base de données Mysql ***Employe*** contenant une table
Personnel (Matle, prénom, Nom, Sexe, adresse, pays).
Créer une servlet qui permet soit de sauver de nouveaux employés dans la base soit de modifier les données concernant un employé soit d'afficher sur une table tous les employés.
On se servira d'un formulaire de saisie avec trois boutons.

Exercice 3: *JDBC et transfert de contrôle à un autre servlet.*

On dispose d'une base de données Mysql ***Banque*** contenant une table
Compte (NumCompte, prénom, Nom, Sexe, adresse, pays, solde, decouvert).
Créer une servlet qui demande à l'utilisateur de saisir le numéro de compte d'un client, de faire un forward vers une deuxième servlet qui se chargera de vérifier si ce compte existe et d'afficher le solde et le decouvert du compte.
Si le compte n'existe, pas, elle l'indique aussi.
On se sert aussi de formulaire.

Exercice 4: *utilisation des attributs de sessions.*

Créer deux servlets ***ServletSession_1*** et ***ServletSession_2*** mappées respectivement sur ***session1.html*** et ***session2.html*** .
Créer un attribut de session dans ***ServletSessio_1*** et utiliser un lien vers ***session2.html*** pour montrer que cet attribut de session est aussi visible dans ***ServletSession_2*** .
L'attribut de session sera saisi au moyen d'un formulaire HTML et correspond au nom d'une personne.
NB: *pour bien voir l'effet, ouvrez deux sessions sur Internet Explorer et Mozilla Firefox et lancez dans chaque navigateur l'application en saisissant un nom différent.*
Vous pouvez cliquer plusieurs fois sur le bouton précédent.
Vous devrez constater que chaque session a son propre nom de personne.

Exercice 5: *utilisation des attributs d'application.*

Reprendre l'exercice précédent dans deux servlets que vous nommerez ***ServletAppli_1*** et ***ServletAppli_2*** mappées respectivement sur ***appli1.html*** et ***appli2.html***.
Et faîtes que l'attribut de session devienne ici un attribut d'application.
NB: *ici aussi pour bien voir l'effet, ouvrez deux sessions sur Internet Explorer et Mozilla Firefox et lancez dans chaque navigateur l'application en saisissant un nom différent.*
Vous pouvez cliquer plusieurs fois sur le bouton précédent.
Vous devrez constater que ce nom est le même dans chaque session.

Exemple de forward entre deux servlets (1/4)

Dans cette exemple, nous montrons comment faire un forward d'une servlet vers une autre. La première servlet gère un formulaire de saisie de personnels administratifs et la seconde se chargera de faire l'enregistrement dans la base de données et l'affichage de tous les personnels.
C'est le rôle de la première servlet de faire un forward vers la seconde servlet une fois les informations saisies.
Nous appellerons les servlets ***ServletForward_1*** et ***ServletForward_2***.

Nous utiliserons une classe ***Personne*** qui nous servira à construire des personnels que nous enverrons à la deuxième servlet

Exemple de forward entre deux servlets (2/4)

```
public class Personne {
        private String mat, prenom , nom, tel;
        public Personne (String mat,String prenom, String nom,String tel){
                this.mat = mat;
                this.prenom = prenom;
                this.nom = nom;
                this.tel = tel;
        }
public String getMat ( ){
        return mat;
}
public String getPrenom ( ){
        return prenom;
}
public String getNom ( ){
        return nom;
}
public String getTel ( ){
        return tel;
}
}
```

Exemple de forward entre deux servlets (3/4)

```
public class ServletForward_1 extends HttpServlet {

protected void doGet (HttpServletRequest request,HttpServletResponse response)
                                                    throws ServletException, IOException {
response.setContentType ("text/html");                    //nécessaire pour Servletforward_2
/*recuperation des parametres saisis dans le formulaire*/
String matricule = request.getParameter ("Matricule");
String prenom = request.getParameter ("Prenom");
String nom = request.getParameter ("Nom");
String tel = request.getParameter ("Telephone");

Personne p = new Personne (matricule,prenom, nom,tel);

request.setAttribute ("personnel", p);

javax.servlet.RequestDispatcher disp = request.getRequestDispatcher("/forward_2");
disp.forward ( request,response);
```

Exemple de forward entre deux servlets (4/4)

```
public class ServletForward_2 extends HttpServlet {
private java.sql.Connection con;
        protected void doGet (HttpServletRequest request,HttpServletResponse response)
                throws ServletException, IOException {
                PrintWriter out = response.getWriter ( );
                response.setContentType("text/html");
                ResultSet rs = null;
                Statement st = null;
    // récupération du paramètre positionné dans ServletForward_1
try {
        Personne pers = (Personne) request.getAttribute ("personnel");
st = con.createStatement();
st.executeUpdate ("insert into Personnel values('"+pers.getMat()+"','"+pers.getPrenom()+"','"
+pers.getNom()+"','"+pers.getTel()+"')");
rs = st.executeQuery("select * from Personnel");
out.print ( "<h1>Liste de tous les personnels</h1>");
out.print ("<table border =2>");
out.print ( "<th>Prénom</th>");
out.print ( "<th>Nom</th>");
out.print ( "<th>Téléphone</th>");
```

```
while (rs.next ( )){
String pr   = rs.getString ("prenom");
String nm = rs.getString ("nom");
String tl    = rs.getString ("telephone");
                out.print ( "<tr>");
                out.print ( "<td>"+pr+"</td>");
                out.print ( "<td>"+nm+"</td>");
                out.print ( "<td>"+tl+"</td>");
                out.print ( "</tr>");
}
out.print( "</table>");
}
catch (Exception e){out.println ("erreur 1");out.println (e.getMessage());}

}
```

Pour lancer l'application et appeler la seconde servlet

```
<html>
        <head>
                <title> Transfert de servlet à servlet </title>
        </head>

<body>
<form  action="/forward_1" method= "GET">

                Matricule           <input type = "text" name = "Matricule"></br>
                Prénom              <input type = "text" name = "Prenom"></br>
                Nom                 <input type = "text" name = "Nom"></br>
                Téléphone           <input type = "text" name = "Telephone"></br></br>
                        <input type = "submit" name = "bouton" value = "Valider">
</form>

</body>

</html>
```

testforward_1.html

Exemple de lancement

Gantry HTML - Windows Internet Explorer

http://localhost:8080/testforward_1.html

Gantry HTML

Page Outils

Matricule 751

Prénom Mamadou

Nom Ndiaye

Téléphone 4532321

Valider

Terminé | Internet | Mode protégé : activé | 100%

http://localhost:8080/forward_1.html?Matricule=751&Prenom=Mamadou&Nom=Ndiaye&Telephone=4532321&...

http://localhost:8080/forward_1?Matricule=751&Prenom=Mamadou&Nom=Ndiaye&Telephone=4532321&bouton=Valider

Liste de tous les personnels

Prénom	Nom	Téléphone
Joseph	Ndong	5107083
Natacha	Melemoko	5780996
Serge	Badji	3457786

Terminé | Internet | Mode protégé : activé | 100%

Remarque sur cet exemple:

On remarque que les paramètres saisis dans le formulaire apparaissent en clair dans la barre d'adresse du navigateur. Cela est dû au fait que dans les deux servlets, nous avons utilisé la méthode **doGet** et dans le formulaire HTML, nous avons indiqué que la méthode de traitement est **GET**.

Si vous ne souhaitez pas que ces paramètres apparaissent, utilisez plutôt la méthode **doPost** dans les deux servlets et indiquez aussi dans le formulaire (***testforward.html***) que la méthode utilisée est **POST**.

Autre façon de rediriger une requête d'une servlet à une autre: sendRedirect

On se donne deux servlets, **RedirectServlet** et **ValideUserServlet**.
On se donne un formulaire HTML *sendform.html* où l'utilisateur tape son login et son password.
La servlet **RedirectServlet** vérifie si l'utilisateur a fourni un login et un password justes.
Au cas échéant, elle redirige la requête à la servlet **ValidUserServlet** qui affiche alors un message de bienvenue à cet utilisateur sinon **RedirectServlet** demande au formulaire d'afficher *« vous n'êtes pas un utilisateur autorisé »*.

```
<html>
<head>
<title>Insert title here</title>
</head>
<body>
<form method="POST" action="/monappli/redirect">
   <p>Login:        <input type="text" name="username" size="20"></p>
   <p>Password:     <input type="text" name="password" size="20"></p>
   <input type="submit" value="Submit" name="Envoyer"></p>
</form>
</body>
</html>
```

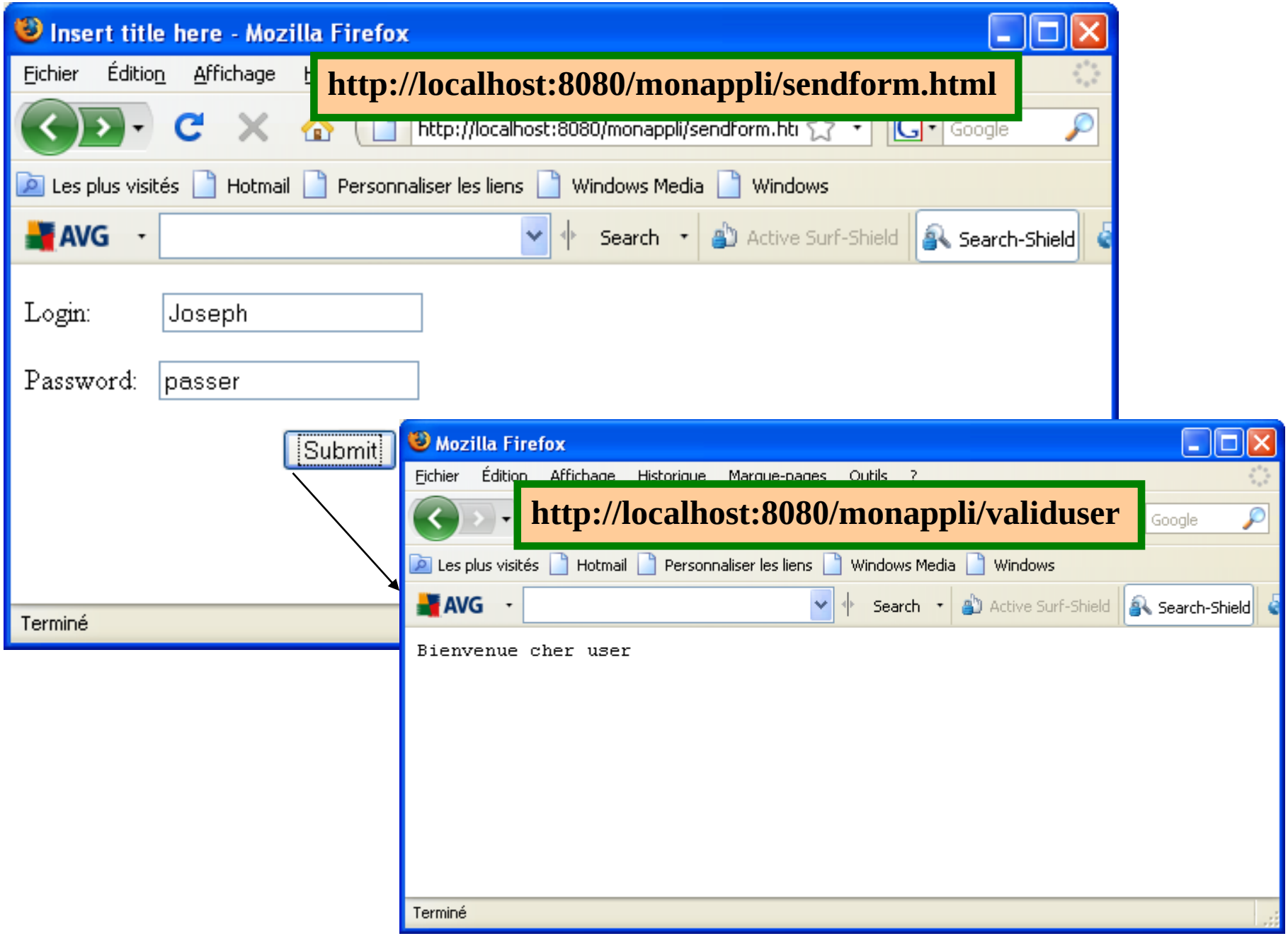
sendform.html

```
package servlets.joe;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
public class ValideUserServlet extends HttpServlet {
            private static final long serialVersionUID = 1L;
protected void doGet (HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
                                                PrintWriter pw = response.getWriter ( );
                                                 pw.println ("Bienvenue cher user ");

            }
}
```

ValiderUserServlet.java

```
package servlets.joe;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class RedirectServlet extends HttpServlet {
protected void doPost (HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
response.setContentType ("text/html");
PrintWriter pw = response.getWriter ( );
String name = request.getParameter("username");
String password = request.getParameter("password");
if (name.equals("Joseph")&& password.equals("passer")){
            response.sendRedirect("/monappli/validuser");
   }
else{pw.println ("Utilisateur non autorise!");
}
}
}
```

RedirectServlet.java


Insert title here - Mozilla Firefox
Fichier Édition Affichage
http://localhost:8080/monappli/sendform.html
http://localhost:8080/monappli/sendform.htı
Google
Les plus visités Hotmail Personnaliser les liens Windows Media Windows
AVG
Search
Active Surf-Shield
Search-Shield
Login:
Joseph
Password:
passer
Submit
Terminé
Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
http://localhost:8080/monappli/validuser
Google
Les plus visités Hotmail Personnaliser les liens Windows Media Windows
AVG
Search
Active Surf-Shield
Search-Shield
Bienvenue cher user
Terminé

Exercice

Reprendre l'exemple précédent en tenant compte des remarques suivantes:

On suppose que tous les utilisateurs autorisés à se connecter au site sont inscrits dans un ficher ***users.txt*** qui ressemble à ceci.

```
#login     #password
Joseph      passer
Pierre      logpass
Amy         logamy
….
```

Lorsqu'un utilisateur se connecte au site, on vérifié la validité du login et du mot de passe dans ce fichier. Si les informations données sont correctes, on répond par un message du type *« Bienvenue Pierre sur note site »,* sinon on répond par un message *« vous n'êtes par un utilisateur autorisé ».*

Paramétrage avancé

Il est possible d'ajouter un certain nombre de paramètres dans le fichier **server.xml**.
Ces paramètres vont influencer sur les différentes caractéristiques du pool de connexions.

Le nombre de connexions

Trois paramètres permettent d'influencer la gestion du nombre de connexions dans le pool :

· **maxActive** : le nombre maximum de connexions qui peuvent être allouées depuis le pool. Sa valeur par défaut est 8, une valeur de 0 indique "sans limite".
Il faut être sûr que la base de données autorisera ce nombre de connexions.

· **maxIdle** : le nombre maximum de connexions inactives qui peuvent être dans le pool. Sa valeur par défaut est de 8, une valeur de 0 indique "sans limite"

· **maxWait** : le temps d'attente maximum en millisecondes pour l'obtention d'une connexion. Passé ce délai, une exception est levée. Sa valeur par défaut est -1 qui correspond à un temps infini.

JSP
(Java Server Page)

Une page *JSP* est une page *HTML* qui peut contenir du code java.
Dans ce chapitre, nous allons voir comment écrire des pages *JSP* et déplacer la logique métier dans des ***classes externes*** au lieu de les inclure dans les pages *JSP*.

Tomcat ***compile les fichiers JSP dans des Servlets*** lorsqu'ils sont demandés pour la *première fois* (c'est-à-dire lorsque le tout premier utilisateur y accède) et exécute le code des servlets compilées.

Remarque importante :

Lorsqu'une JSP est compilée en servlet, cette servlet (donc le fichier **.class** résultant) est stocké dans le répertoire **work** de **Tomcat**.
Consultez ce répertoire pour voir comment sont constitués ces fichiers *.class*.

Une première JSP

Lorsqu'on écrit une page JSP, c'est du texte brut, contenant du HTML et du java.
Ce code java est constitué de balises spéciales donnant des instructions au compilateur JSP (***Jasper*** pour *Tomcat*).
La première fois qu'un fichier JSP est sollicité, celui-ci est compilé en une classe servlet.
Cette servlet est alors chargée et fonctionne comme toute autre servlet.

Eclipse ganymède permet de créer facilement des JSPs.

Créez un nouveau projet (*DWP*) nommé ***modjsp***.
Ajoutez à ce projet une page JSP ***MaJSP.jsp*** .
Elle peut ressembler au code ci-dessous. Nous déclarons dans cet exemple deux variables entières et effectuons la somme et l'affichage.

Notes importantes: les fichiers JSP doivent être placés ***à la racine de l'application WEB*** (au même endroit que les fichiers HTML si ils existent).
Vous n'avez pas besoin de modifier le fichier ***web.xml*** comme pour les servlets.

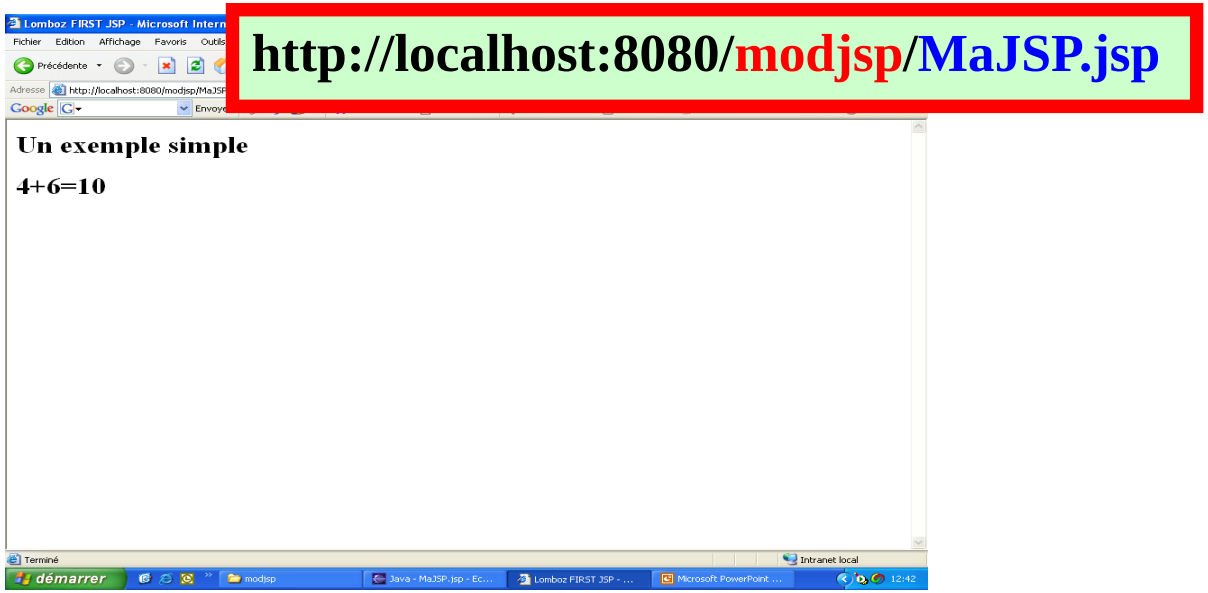
Une première JSP

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
        <head>
                <title> Lomboz FIRST JSP</title>
        </head>

<%
  int a = 4;
  int b = 6;
%>
<h1>Un exemple simple<h>
<p>
<% out.println ("Addition de deux entiers :"); %>
<%= a%>+<%= b%>=<%= a+b %>
</p>
</body>
</html>
```

Une première JSP

Pour tester:

NB: à chaque modification du code de votre JSP, faîtes un redéploiement.



Remarques sur l'exemple

Une page JSP commence toujours par la directive page: **<%@ page language="java" %>**
Le code java doit se trouver entre les balises **<% et %>**
La page JSP sera compilée en servlet mais contrairement aux servlets, vous n'avez pas besoin de la déclarer dans un fichier ***web.xml*** .

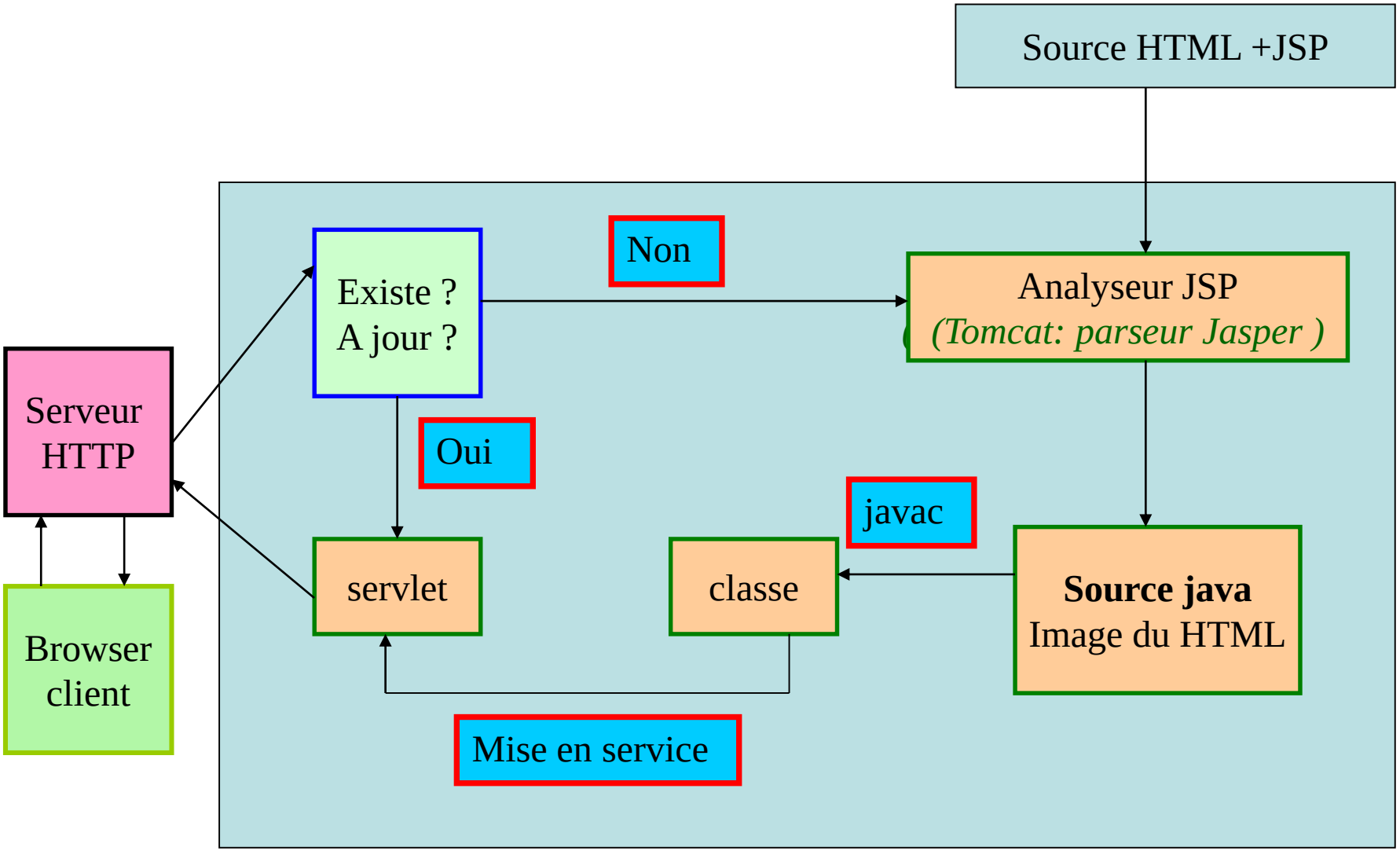
Sachez qu'il existe des variables prédéfinies utilisables directement dans la page, comme ***out*** et d'autres qu'on verra par la suite.
Dans cet exemple, on pourra écrire : **out.println ("exemple de jsp ");**

Rappelez vous que le but des JSPs est de dissocier HTML et code Java, c'est pourquoi l'usage des ***out.println ( )*** n'est pas approprié. On verra comment réaliser cette dissociation convenablement.

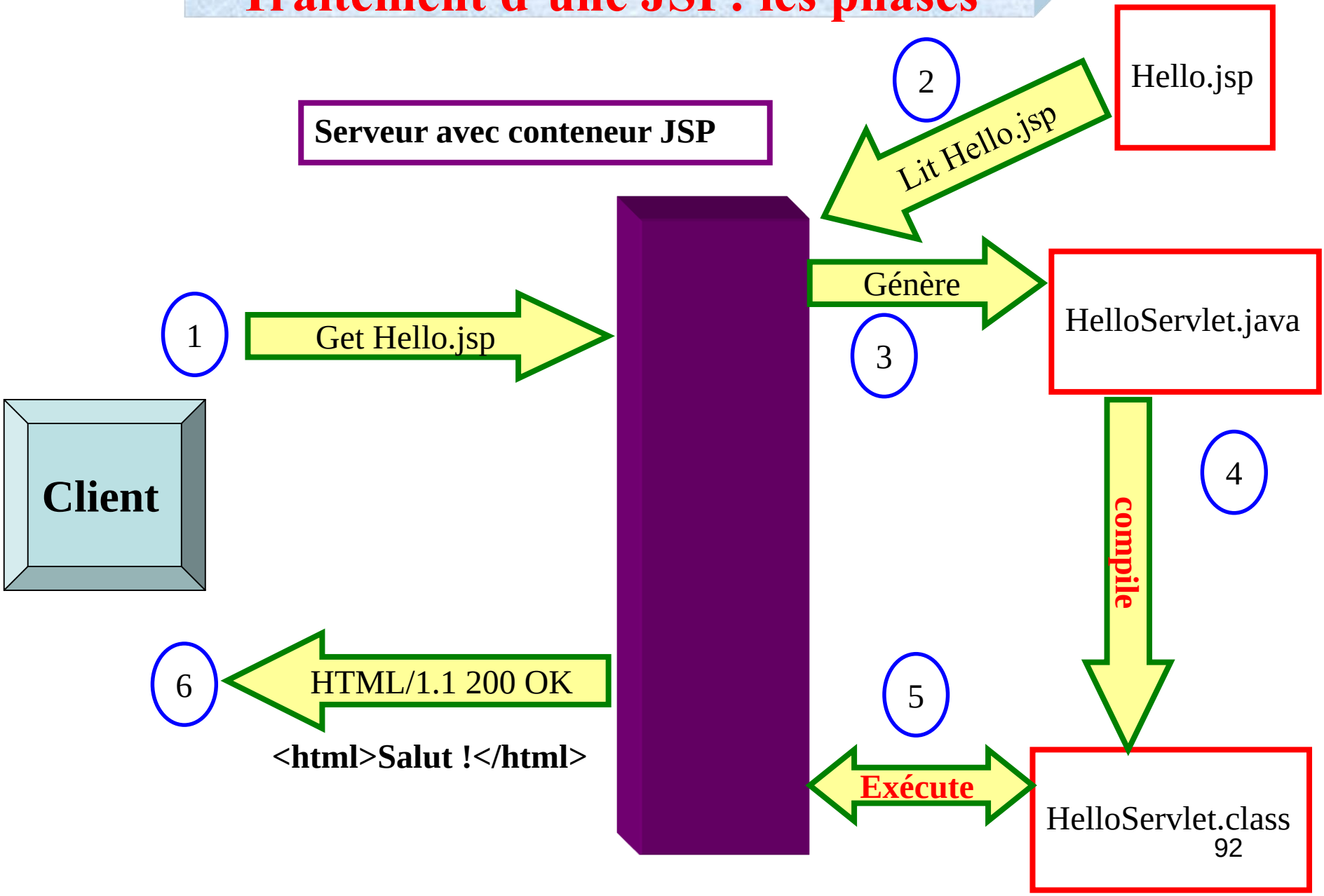
Si vous voulez mettre des commentaires, sachez qu'il en existe deux types**:**

- **<%-- commentaire JSP --%>** invisible dans le navigateur
- **<!– commentaire HTML -->** visible dans le navigateur (si vous faites ***Affichage/Source***)

Cycle de vie d'une JSP



Traitement d'une JSP: les phases



Cycle de vie d'une JSP

Génération de source java et compilation

Lorsque le moteur de JSP reçoit pour la première fois une requête d'un fichier JSP, le processeur JSP analyse la page et convertit les balises JSP en code Java. Il enrobe ensuite les instructions HTML dans du code Java. Le code java généré est ensuite compilé en servlet. Le cycle de vie de la page JSP devient similaire au cycle de vie d'une servlet.

Traitement de la requête

Lorsque que la page JSP est compilée en servlet, le moteur de servlet crée une instance de la servlet et appelle la méthode « **service ( )** » en réponse à la demande.
A chaque réception de requête, le moteur effectue une vérification afin de déterminer si le fichier JSP a été modifié depuis la dernière compilation. S'il a été modifié, le traitement de génération de source java et de compilation est à nouveau effectué. L'instance de servlet nouvellement chargée reçoit et traite la demande client.

Arrêt

Lorsque le moteur de servlet n'a plus besoin de la servlet ou qu'une nouvelle instance de la servlet est en cours de rechargement, il appelle la méthode « **destroy ( )** ». Comme pour les servlets, la machine virtuelle java (JVM) procède ensuite à une récupération des ressources mémoires qui avaient été allouées.

Différentes parties d'une JSP

Une page JSP est constitué de plusieurs éléments: le code HTML, les directives, les commentaires, les scripts Java et les actions.

1) Le **code HTML** est introduit et doit être facile à modifier pour faire apparaître les parties dynamiques de la page.

2) les **directives**:

Elles permettent de faire des import de package, de définir des pages d'erreur et les informations de sessions des pages.

- **page**: permet d'indiquer le langage utilisé dans la page (**java**) et aussi de préciser les instruction d' **import** à effectuer pour les classes java nécessaires

```
<%@ page language="java"
        import = " java.util.ArrayList ",
                 " java.net.Socket " %>
```

- **include:** permet d'inclure le fichier indiqué : **<%@ include file="index.jsp" %>**

include est surtout utilisé dans le cas de parties de code HTML répétitives entre les pages jsp.

- **taglib**: permet d'utiliser des librairies de tags personnalisés.

```
<%@ taglib prefix = "abc" uri = "path"%>
```

Différentes parties d'une JSP

3) **Les scripts Java**: ce sont des morceaux de code java qui sont insérés dans le code HTML de la page. Un script commence par **<%** et se termine par **%>.**
un script peut utiliser des **variables prédéfinies**, qui vont permettre d'accéder aux mêmes éléments qu'une servlet:

- **request**: de *HttpServletRequest*
- **response** de la classe *HttpServletResponse*
- **session** : de *HttpSession*
- **application**: de *ServletContext*
- **out**: de *PrintWriter*
- **page**: de *Object*
- **config**: de *ServletConfig*
- **pageContext**: de *javax.servlet.jsp.PageContext*
- **exception**: de *Throwable*.

4) **Les actions**: les actions sont écrites en XML et permettent de condenser en une ligne un traitement qui serait long à écrire à l'aide de script java.
il existe deux types d'actions: **standards** et **personnalisées**.

Balises de scripting d'éléments

Avant de voir les différentes actions, regardons d'abord les balises de script java qu'on peut utiliser dans une page JSP.

Bloc de déclaration de données <%! … %> :

Ces balises permettent de déclarer des variables et des méthodes d'instances à utiliser dans toute la page. Les balises de scripting ne génèrent aucun caractère dans le fichier HTML de sortie.

Exemple :

```
<% !
        private int nombreAcces = 0 ;
        private int incrementerNombreAcces ( ) {return  nombreAcces++ ;}
%>
```

Affichage des données <%= … %> :

Ces balises doivent contenir une et une seule expression valide et complète du langage. Elles sont utilisées pour intégrer des valeurs dans le code HTML. L'expression est évaluée et convertie en chaîne de caractères avant d'être affichée sur le navigateur.

Exemple :

```
Bonjour. Il est <% = new java.util.Date ( ) %>
```

Balises de scripting d’éléments

Scriptlets <% … %>

Les scriptlets sont des blocs de code java. Ils peuvent contenir un ensemble d'instructions et de déclarations. Les variables qui y sont déclarées ne sont valables que dans la partie ultérieure de la page JSP à la différence des blocs de déclarations.

Ce tag ne peut pas contenir de tags HTML ou JSP. Pour écrire ces tags, il faut fermer le tag du scriptlet, mettre le tag HTML ou JSP puis de nouveau commencer un tag de scriptlet pour continuer le code.

```
<%
String nom = request.getParameter("nom") ;
out.println ("Nom de l’utilisateur : " + nom) ;
%>
```

Les actions standards

Les actions standards sont celles de base, accessibles à toutes les pages jsp car définies dans l' API JEE pour les pages JSP. Les éléments XML correspondant commencent par le préfixe **jsp**.

Les actions personnalisées viennent enrichir les actions standards en offrant de nouvelles possibilités. On les utilise grâce aux librairies de tags personnalisés déclarés dans ***taglib***.

Voyons d'abord les actions standards.
Les actions personnalisées seront vues quand nous aborderons les librairies de tags.

Les actions standards

<jsp:useBean /> : permet de ***créer ou d'importer*** des attributs de requête, de session et d'application. L'attribut créé ou importé peut ainsi être utilisé dans la page JSP, comme s'il correspondait à une variable de la classe indiquée dans ***usebean***.

voici la syntaxe: **<jsp:useBean id = "nomAttribut" class = "package.Classe"**
scope = "portéeAttribut" />

voici sa signification: si l'attribut de nom *nomAttribut* existe dans la portée indiquée alors il est importé dans la page et on peut l'utiliser, sinon il est créé (de classe *package.Classe*) avec la portée indiquée et on peut aussi l'utiliser.

NB: la classe déclarée ne doit pas disposer d'aucun constructeur, en dehors du constructeur par défaut sans paramètres.

portéeAttribut peut prendre les valeurs suivantes:
- *request* pour les attributs de portée requêtes
- *session* pour les attributs de portée session
- *application* pour les attributs de portée application
- *page* pour les attributs de portée page.

Voici un exemple :

Les actions standards

```
<jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "request"  />
```

Signifie: si la page JSP a dans la portée ***request*** l'attribut de nom ***personne*** de la classe *Personne*, alors on peut l'utiliser, sinon on le crée pour l'utiliser par la suite.

Cette action correspondrait au script java suivant:

```
<% Personne personne = (Personne) request.getAttribute (" personne ");
    if (personne == null)
        {
            personne = new Personne ( );
             request.setAttribute (" personne ",personne);
         }
%>
```

Remarques :
Pour l'action ***useBean***, la classe utilisée n'a pas à être un **JavaBean**.
Mais si vous utilisez les actions ***setProperty*** et ***getProperty*** en liaison avec useBean, la classe doit être nécessairement un JavaBean.
Un **JavaBean** est une classe java possédant éventuellement un constructeur **sans** paramètres qui contient des propriétés (champs) et des *méthodes* et tous les champs sont accessibles par des ***getter*** et ***setter***. La classe doit implémenter ***java.io.Serializable*** .

<jsp:setProperty />: permet **d'initialiser** une ou plusieurs propriétés d'un *JavaBean*, *grâces aux paramètres présents dans la requête appelant la page*.

Exemple:

```
<jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "session"  />
<jsp:setProperty  name = " personne " property = " nom " />
```

Signification:
jsp:useBean: si on utilise l'attribut *personne* de type **Personne** pour la *session*
jsp:setProperty: utilise cet attribut et suppose que c'est un *JavaBean* pour initialiser sa propriété **nom** avec le paramètre ***nom*** de la requête reçue. *Si ce paramètre nom n'a pas été reçu dans la requête, la propriété n'est pas initialisée.*
En d'autres termes si comme si vous aviez écrit ceci:

```
<jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "session"  />
<% if (request.getParameter (" nom ") != null)
        personne.setNom ( request.getParameter (" nom ")); %>
```

Si on voulait initialiser toutes les propriétés du JavaBean d'un seul coup, avec les paramètres passés dans la requête, on ferait:

```
<jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "session"  />
<jsp:setProperty  name = " personne " property = " * " />
```

L'usage de ***jsp:setProperty*** est seulement intéressant si la liste des paramètres à initialiser est longue ou évolutive, sinon elle est facilement remplaçable par ***request.getParameter* ( )**.

Les actions standards

< jsp:getProperty />:permet d'afficher la propriété indiquée d'un JavaBean

Exemple:

```
< jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "session"  />
<jsp:getProperty  name = " personne " property = " nom " />
```

Signification:

jsp:useBean: si on utilise l'attribut *personne* de type **Personne** pour la *session*

jsp:getProperty: utilise cet attribut et suppose que c'est un *JavaBean* pour afficher sa propriété **nom** .

En d'autres termes si comme si vous aviez écrit ceci:

```
< jsp:useBean id = "personne"   class = "joe.cours.jsp.Personne"
                                scope = "session"  />
<% = personne.getNom ( ) %>
```

NB: l'utilisation de *jsp:getProperty* oblige dans useBean que l'attribut soit un JavaBean (donc à avoir des méthodes ***get ( )*** et ***set ( )*** sur chacune de ses propriétés) ce qui n'est pas nécessaire lorsqu'on utilise ***<% = personne.nom %>***.

Les actions standards

< jsp:forward />: cette action permet de transférer le contrôle à une autre JSP ou à une servlet.
Voici la syntaxe:

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

Ceci équivaut à:

```
<% RequestDispatcher disp = request.getRequestDispatcher (" verspage.jsp " );
    disp.forward (request, response);
%>.
```

Voici aussi une façon d'écrire l'action forward:

```
< jsp:forward page = " verspage.jsp ">
< jsp:param  name = " prénom " value = " joseph " >
< jsp:param  name = " nom " value = " ndong " >
< jsp:forward>
```

Cela signifie de transférer le contrôle à *verspage.jsp* tout en ajoutant en paramètres de la requête les paramètres indiqués par ***jsp:param***.
Et ces paramètres de formulaire peuvent être affichés dans *verspage.jsp* par le script suivant:

```
<% = request.getParameter (" prénom ") %> <br>
<% = request.getParameter (" nom ") %> <br>
```

Remarque: il faut faire **ATTENTION** sur l'utilisation de cette action car:

-Comme pour les servlets, le flux de sortie de ***pageinit.jsp*** (qui fait le forward) est annulé au profit de ***verspage.jsp*** (vers laquelle on forward). Cela signifie donc que *pageinit.jsp*, avec tout son code HTML ne sera pas prise en compte (ce qui n'est pas du tout intéressant). Il n'est pas intéressant d'écrire une JSP dont le contenu ne sera jamais vu (affiché dans le navigateur), sauf si cette page ne contient que des script java, sans code HTML.

Et dans ce cas précis mieux veut utiliser une servlet et non une JSP.

< jsp:include />: cette action permet de transférer le contrôle à une autre JSP ou à une servlet. Contrairement à l'action *forward*, ici, on **revient à la page initiale** (d'où est partie le transfert) pour continuer l'exécution, une fois qu'on a terminé avec la page où l'on a fait le transfert.

Syntaxe:

```
< jsp:include page = " verspage.jsp " />
```

Ceci équivaut à:

```
<% RequestDispatcher disp = request.getRequestDispatcher (" verspage.jsp " );
    disp.include (request, response);
%>.
```

NB: il ne faut pas confondre cette action avec la directive <%@ **include** file "page.jsp" %>

La directive est exécutée **une seule fois** *au moment de la compilation de la page JSP*, tandis que l'action est exécutée autant de fois que la page est appelée et peut donc produire un résultat différent à chaque appel.

Gestion des erreurs

Il est possible avec les JSP d'intercepter des erreurs d'exécution et d'afficher une page d'erreur.

Page normale

```
<%-- Exemple de page de traitement JSP --% >
<%@ page language = "java" %>
<%@ page errorPage = "/jsp/ErrorPage.jsp" %>
<%@ page import  = "java.util.LinkedList" %>
```

Page d'erreur

```
<%-- Exemple de page d'erreurJSP --% >
<%@ page language = "java" %>
<%@ page isErrorPage = "  true" %>
```

Dans une page normale (non dédiée au traitement des erreurs), il faut déclarer la page de déroutement à l'aide de la directive suivante :

<%@ page errorPage="/jsp/ErrorPage.jsp" %>

Il ne sera donc pas nécessaire de mettre la directive ***<%@ page isErrorPage="false" %>*** (elle devient implicite)

Dans une page de traitement des erreurs, il faut indiquer qu'elle est la page d'erreur à l'aide de la directive :

<%@ page isErrorPage="true" %>**.** Pour cette page la directive *<%@ page errorPage="..." %>* est inutile et ne sera pas prise en compte.

Exemple :

```
<%-- Page d'erreur JSP pour l'application --%>
<%@ page language="java" %>
<%@ page isErrorPage="true" %>
<html>
<head><title>Page d'erreur JSP</title></head>
<body>
<div align="center"><h1>*** Erreur fatale ***</h1></div>
<br><br><br>Détails : <br><br>
<font color="red"><%= exception.toString ( ) %><br></font>
</body>
</html>
```

Les actions (ou tags) personnalisées: directive taglib

Les actions personnalisées permettent d'embarquer une logique applicative dans les pages JSPs à l'aide de balises de type HTML.

La directive **taglib** permet d'indiquer que la page utilise une bibliothèque de balises personnalisées, identifiée de manière non ambiguë par son URI et associée à un préfixe permettant d'indiquer pour chaque balise utilisée la bibliothèque dont elle provient.

Possibilité de définir ses propres tags basés sur XML :

- *tags personnalisés (custom tags)*
- *regroupés dans des bibliothèques de tags (taglibs)*

Objectifs:

- *Déporter dans des classes dédiées le code java contenu dans les scriptlets de la JSP et appeler ce code en utilisant des tags particuliers*
- *Améliorer la séparation des rôles :*
  - *+ page JSP : présentation – concepteur de pages Web*
  - *+ scriplets / code Java – développeur Java*

Tags personnalisés / Java Beans

- *"philosophie" similaire*
- *Java Beans : objets métier pour stocker et échanger des données*
- *Tag personnalisé : interagit directement avec environnement JSP dans lequel il s'exécute*

Les actions (ou tags) personnalisées: directive taglib

Que peuvent faire des tags personnalisés ?

- ✓ *Produire du contenu pour une page JSP*
- ✓ *Recevoir des paramètres envoyés à partir de la JSP qui les appelle*
- ✓ *Avoir un corps qu'ils peuvent manipuler.*
- ✓ *Possibilité d'imbriquer un tag personnalisé dans un autre*
- ✓ *avec un nombre d'imbrications illimité*
- ✓ *Accéder aux Java Beans définis dans la page JSP*
- ✓ *Introduire de nouveaux Java Beans*
- ✓ *Introduire de nouvelles variables de scripting*

Les actions (ou tags) personnalisées: directive taglib

Pour définir une bibliothèque de tags :

Chaque Tag personnalisé est implémenté à l'aide d'un objet Java (donc une classe à créer) (il s'agit d'un ***Gestionnaire de balise*** *appelé* ***Tag Handler****)*

*Un fichier de description de la bibliothèque (****TLD: Tag Library Descriptor****)*
ce fichier d*éfinit la syntaxe des tags (les règles d'écriture des tags qu'il faut respecter)*
– Nom, attributs ….
– Classe du Tag Handler associé
C'est un document XML qui doit donc être bien formé.

En pratique, sachez que une bibliothèque de balises se compose d'un ensemble de classes qui implémentent les interfaces ***javax.servlet.jsp.tagext.Tag*** *et* ***BodyTag****.et d'un descripteur de bibliothèque de balises ou ficher TLD. Ces balises personnalisées sont destinées à être utilisées dans des pages JSPs.*

Mais le moyen le plus de créer une bibliothèque de balise sera de créer une sous classe de la classe ***TagSupport*** *ou* ***BodyTagSupport****.*

Création d'une action (tag) personnalisée.

Supposons que l'on veuille créer une action (nommée **hello**) qui gère l'instruction d'affichage **out.println (« bonjour ») ;** cette instruction devrait normalement se situer à l'intérieur d'une page JSP.

Voici la procédure à suivre pas à pas :

1. Création d'une classe associée à l'action

la création de cette classe est nécessaire pour que l'action fonctionne.

Plusieurs classes et interfaces (de l' API) sont disponibles pour créer des tags.

Utilisons celle-ci: **TagSupport**

```
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;
public class TagClass  extends TagSupport{
public int doStartTag ( )  throws JspException{
JspWriter out = pageContext.getOut ( );
try{ out.println ("salut le monde");} // ce que fera l'action
catch (java.io.IOException e){ }
return SKIP_BODY;}}
```

Compatibilité entre les serveurs JEE.

Le fichier **.tld** peut provoquer certaines incompatibilités entre les serveurs si vous en utilisez plusieurs:
Selon les serveurs certains éléments du fichier **.tld** peuvent être obligatoires ou non:

- La directive <!**DOCTYPE**> est obligatoire pour Tomcat et JBoss et non pour ***JRun***
- les éléments <***tlibversion***> et <***jspversion***> sont obligatoires pour **Tomcat** et **JBoss** et non pour ***JRun***
- l'élément <***shortname***> n'est obligatoire que pour ***JBoss*** même si le corps peut être vide

*Pour ne pas rencontrer des problèmes de compatibilité, prenez la peine d'écrire le fichier .**tld** en prenant en compte les contraintes les plus élevés.*

Création d'une action (tag) personnalisée.

2. Création du fichier .tld

dans cette deuxième phase il faut créer un fichier dont le rôle est de faire la ***correspondance*** entre le **nom de la classe** et le **nom de l'action** dans les pages JSP. ce fichier (que je nomme ici **deftag.tld**) sera stocké dans un répertoire: **/WEB-INF/tlds** sous l'application web (créer explicitement ***tlds*** s'il n'existe pas).

```
<?xml version ="1.0" encoding ="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems,
Inc.//DTD JSP Tag Library 1.1//EN" "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
        <tlibversion>1.0</tlibversion>
        <jspversion>1.1</jspversion>
        <shortname>test</shortname>
        <tag>
                <name>hello</name>
                <tagclass>joseph.TagClass</tagclass>
                <bodycontent>empty</bodycontent>
        </tag>
</taglib>
```

Création d'une action (tag) personnalisée.

3. Dans cette dernière partie, on crée une page JSP
dans cette JSP, on va utiliser le tag ainsi crée. Ici on lie la page JSP et le fichier .tld par une directive appelée **taglib**.
le **préfixe** que nous utilisons librement devra être utilisé pour **toutes les actions** se trouvant dans le fichier **.tld**. Dans notre exemple, nous avons une seule action.

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/tlds/deftag.tld" prefix="mytag" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Lomboz JSP</title>
</head>
<body bgcolor="#FFFFFF">

J'appelle le tag ici : <mytag:hello />
</body>
</html>
```

Sortie → **J'appelle le tag ici :** salut le monde

Exemple de JSP : un compteur avec une initialisation et une destruction

```
<%@ page language="java" contentType="text/html" %>
<%@ page import="java.util.Date" %>
<%!
        int global_counter = 0;
        Date start_date;
public void jspInit ( ) {
                        start_date = new Date ( );
                    }
public void jspDestroy ( ) {
            ServletContext context = getServletConfig ( ).getServletContext ( );
            context.log ("test.jsp a été visitée " + global_counter + "fois entre le"
                        + start_date + " et le " + (new Date ( )));
             }
%>
<html>
        <head><title>Page avec un compteur</title></head>
<body bgcolor="white">
Cette page a été visitée : <%= ++global_counter %> fois depuis le <%= start_date %>.
</body></html>
```

Quelques exemples de JSPs
mettant en œuvre la théorie
sur les différents éléments constituant les JSPs.

Exemple 1: le but ici est de récupérer les données saisies dans un formulaire HTML et de les afficher dans une page JSP.

joejsp.jsp

```
<%@ page language ="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html> <head> <title>Lomboz JSP</title> </head>
<body>
<h1> Affichage des parametres saisis</h1>
Le nom saisi est: <% = request.getParameter ( "nom")%></br>
Le prénom saisi est: <% = request.getParameter ( "prenom")%></br>
Le téléphone saisi est: <% = request.getParameter ( "telephone")%></br>
</body>
</html>
```

testjsp.html

```
<html>
<head> <title> Formulaire HTML</title></head>
<body >
<form action = "joejsp.jsp" method ="get">
Nom                <input type = "text" name = "nom"></br>
Prenom             <input type = "text" name = "prenom"></br>
Telephone          <input type = "text" name = "telephone"></br></br>
            <input type ="submit" name = "bouton" value = "Valider">
</form> </body>
</html>
```

Exemple 2: ici on reprend l'exemple précédent mais le formulaire HTML et l'affichage sont inscrits dans la page JSP.

joejsp.jsp

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html> <head><title> Lomboz  JSP</title> </head> <body>
<h1> Formulaire saisie</h1>
<form action = "joejsp.jsp" method ="get">
Nom                    <input type="text" name="nom"></br>
Prenom                 <input type="text" name="prenom"></br>
Telephone              <input type="text" name="telephone"></br></br>
                 <input type="submit" name="bouton" value="Valider">
</form>
<% if (request.getParameter ("nom")!=null)
   {
 %>
Le nom saisi est:          <%=request.getParameter ( "nom")%></br>
Le prénom saisi est:       <%=request.getParameter ( "prenom")%></br>
Le téléphone saisi est:    <%=request.getParameter ( "telephone")%></br>
<%
   }
 %>
</body> </html>
```

Exemple 3: dans cet exemple on tient compte dans le formulaire de la présence d'un RadioButton. Et pour réaliser cette tâche, on utilise une classe Personne.

```
package joejsp_2;
public class Personne {
        public String nom ="";
        public String prenom ="";
        public String telephone ="";
        public String sexe = "masculin";
        public String isMasculin ( ){
                if  (sexe.equals ( "masculin"))
                return "checked";
                return "";
        }
        public String isFeminin ( ){
                        if (sexe.equals ( "feminin"))
                        return "checked";
                        return "";
                }
}
```

Personne.java

Exemple 3 (suite)

```
<%@ page language="java" import="joejsp_2.Personne" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
<title>Lomboz  JSP</title>
</head>
<h1>Formulaire saisie</h1>
<%--cree un objet Personne pour memoriser les infos saisies--%>
<% Personne personne = new Personne ( );%>
<% if (request.getParameter("nom")!=null)
   {

personne.nom          = request.getParameter ("nom");
personne.prenom       = request.getParameter ("prenom");
personne.telephone    = request.getParameter ("telephone");
personne.sexe         = request.getParameter ("sexe");
   }
 %>
```

```
<form action="jspRadioButton.jsp" method ="get">
Nom :    <input type="text" name="nom"  value= "<%=personne.nom%>" /></br>
Prenom :<input type="text" name="prenom" value= "<%=personne.prenom%>" /></br>
Telephone :<input type="text" name="telephone" value= "<%=personne.telephone%>"
/></br></br>
Sexe   :Masculin   <input type="radio" name="sexe"
                                    value= "masculin"<%=personne.isMasculin()%> />
       Feminin    <input type="radio" name="sexe"
                                    value= "feminin"<%=personne.isFeminin()%> /></br>
   <input type="submit" name="bouton" value="Valider" /></br></br>
</form>
 <% if (request.getParameter ("nom")!=null)
   {
 %>
Le nom saisi est: <%=request.getParameter( "nom")%></br>
Le prénom saisi est: <%=request.getParameter( "prenom")%></br>
Le téléphone saisi est: <%=request.getParameter( "telephone")%></br>
La personne est de sexe: <%=request.getParameter( "sexe")%></br>
<%
   }
 %>
</body></html>
```

jspRadioButton.jsp

Exemple 3: on reprend l'exemple précèdent avec quelques modifications:
La classe Personne devient un JavaBean parce que on utilise useBean et setProperty

```
package joejsp_2;
public class PersonneBean implements java.io.Serializable {

        private String nom="“, prenom="“, telephone="“, sexe="masculin";
        public String isMasculin ( ){
                        if (sexe.equals( "masculin"))
                        return "checked";
                        return "";
                }
                public String isFeminin(){
                                if (sexe.equals( "feminin"))
                                return "checked";
                                return "";
                        }

                public String getNom() {
                        return nom;
                }
```

```
                    public String getPrenom ( ) {  return prenom; }

                    public String getSexe ( ) {return sexe; }

                    public String getTelephone ( ) {return telephone; }

                    public void setNom (String string) {nom = string; }
                    public void setPrenom (String string) { prenom = string;}
                    public void setSexe (String string) {sexe = string;              }
                    public void setTelephone (String string) {telephone = string;}
}
```

PersonneBean.java

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html><head><title>Lomboz  JSP</title></head>
                    <h1> Formulaire saisie </h1>
<body>
<%--cree un objet Personne pour memoriser les infos saisies:
l'objet sera automatiquement initialisé par les informations
contenues dans les parametres de la requete (grace au setProprety)--%>
<jsp:useBean id ="personne" class ="joejsp_2.PersonneBean" scope ="page" />
<jsp:setProperty name ="personne" property ="*" />
<form action= "jspBean.jsp" method ="get">
Nom :            <input type="text" name="nom"
                  value= "<%=personne.getNom ( )%>" /></br>
Prenom :         <input type="text" name="prenom"
                  value= "<%=personne.getPrenom ( )%>" /></br>
Telephone :       <input type="text" name="telephone"
                  value= "<%=personne.getTelephone ( )%>" /></br></br>
Sexe   :Masculin   <input type="radio" name="sexe"
              value= "masculin"<%=personne.isMasculin ( )%> />
      Feminin    <input type="radio" name="sexe"
              value= "feminin"<%=personne.isFeminin ( )%> /></br>
   <input type="submit" name="bouton" value="Valider" /></br></br>
</form>
```

```
<% if (request.getParameter ("nom")!=null)
  {
 %>
        Le nom saisi est: <%=request.getParameter( "nom")%></br>
        Le prénom saisi est: <%=request.getParameter( "prenom")%></br>
        Le téléphone saisi est: <%=request.getParameter( "telephone")%></br>
        La personne est de sexe: <%=request.getParameter( "sexe")%></br>
<%
  }
 %>
</body>
</html>
```

jspBean.jsp

Le design pattern MVC

Le pattern MVC est communément utilisé dans les applications Java/JEE pour réaliser la couche présentation des données aussi bien pour les applications Web que pour les clients lourds.

Lorsqu'il est utilisé dans le cadre de JEE, il s'appuie généralement sur l'API servlet et des technologies comme JSP/JSTL.

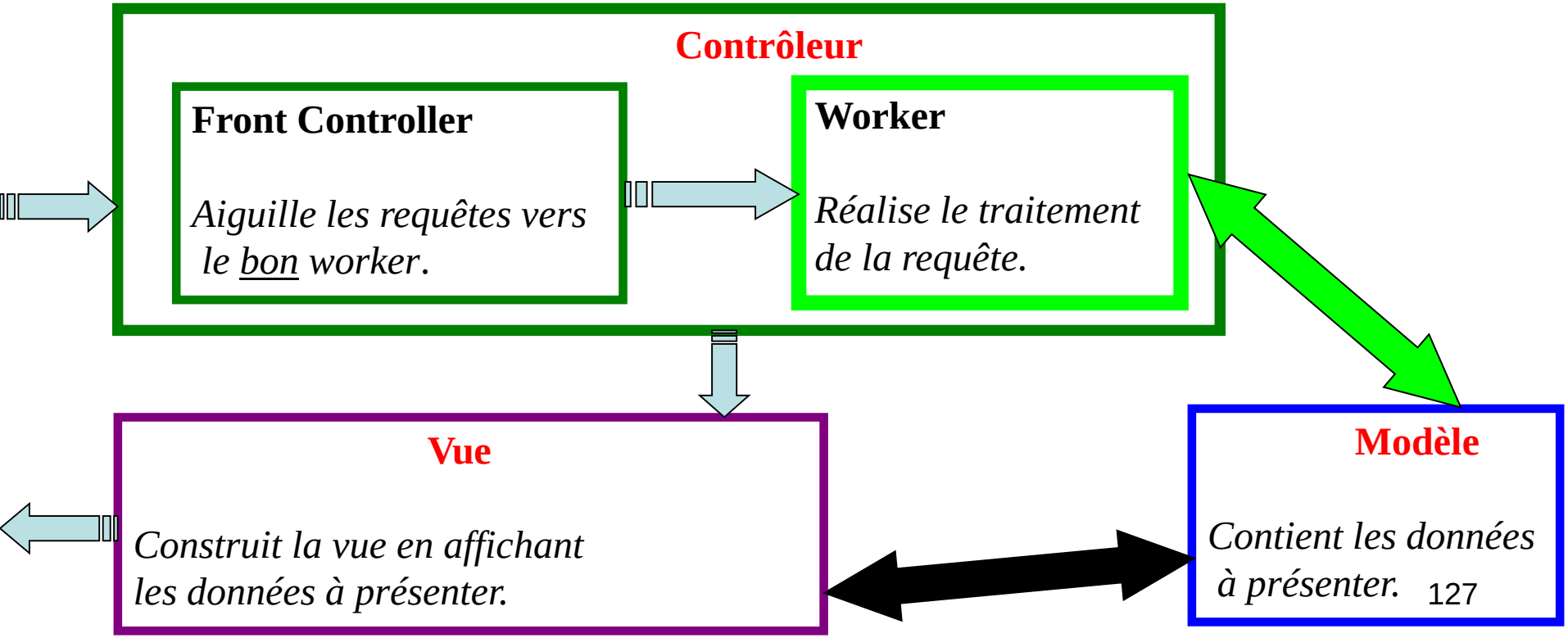
Le modèle de conception MVC met en œuvre trois composants: un **modèle**, une **vue**, un **contrôleur**.

- Le **modèle** contient les objets représentant les données. Ce sont des objets qui sont manipulés afin de produire les éléments qui seront présentés à l'utilisateur.
- La **vue** est la représentation (en général à l'écran) du modèle. La vue représente l'état courant du modèle.
- le **contrôleur** gère les interactions de l'utilisateur avec l'interface. Il manipule le modèle et les objets représentant les données.

Le design pattern MVC

Il existe deux types de pattern MVC, le pattern MVC de **type 1**, qui possède **un contrôleur par action**, et le pattern MVC dit de **type 2**, plus récent qui possède un **contrôleur unique**. Nous nous concentrerons sur ce dernier puisque c'est celui sur lequel que s'appuie **Struts**.

La figure suivante illustre les différentes entités du type 2 du pattern MVC ainsi que les interactions lors du traitement d'une requête:



Notes très importantes:

Le **modèle** est indépendant des API techniques et est constitué uniquement de **JavaBeans** (ou Beans) Java.

La **vue** peut être mise en oeuvre par différentes technologies comme les JSP/JSTL, XML pouvant générer différents types de formats.

Le **contrôleur** gère les interactions avec le client tout en déclenchant les traitements appropriés. Cette entité interagit directement avec les composants de la couche *service métier* et a pour responsabilité la ***récupération*** des données mises à la disposition dans le modèle.
Lors de la mise en œuvre du type 2 de ce pattern, cette partie se compose d'un ***point d'entrée unique pour toute l'application et de plusieurs entités de traitement***.
Ce point d'entrée unique traite la requête et dirige les traitements vers l'entité Worker appropriée.
Pour cette raison, l'entité **Worker** est habituellement appelée **contrôleur**.
Le **Front Controller** ou **« contrôleur façade »** est intégré au framework MVC, et seuls les *workers* sont spécifiques.

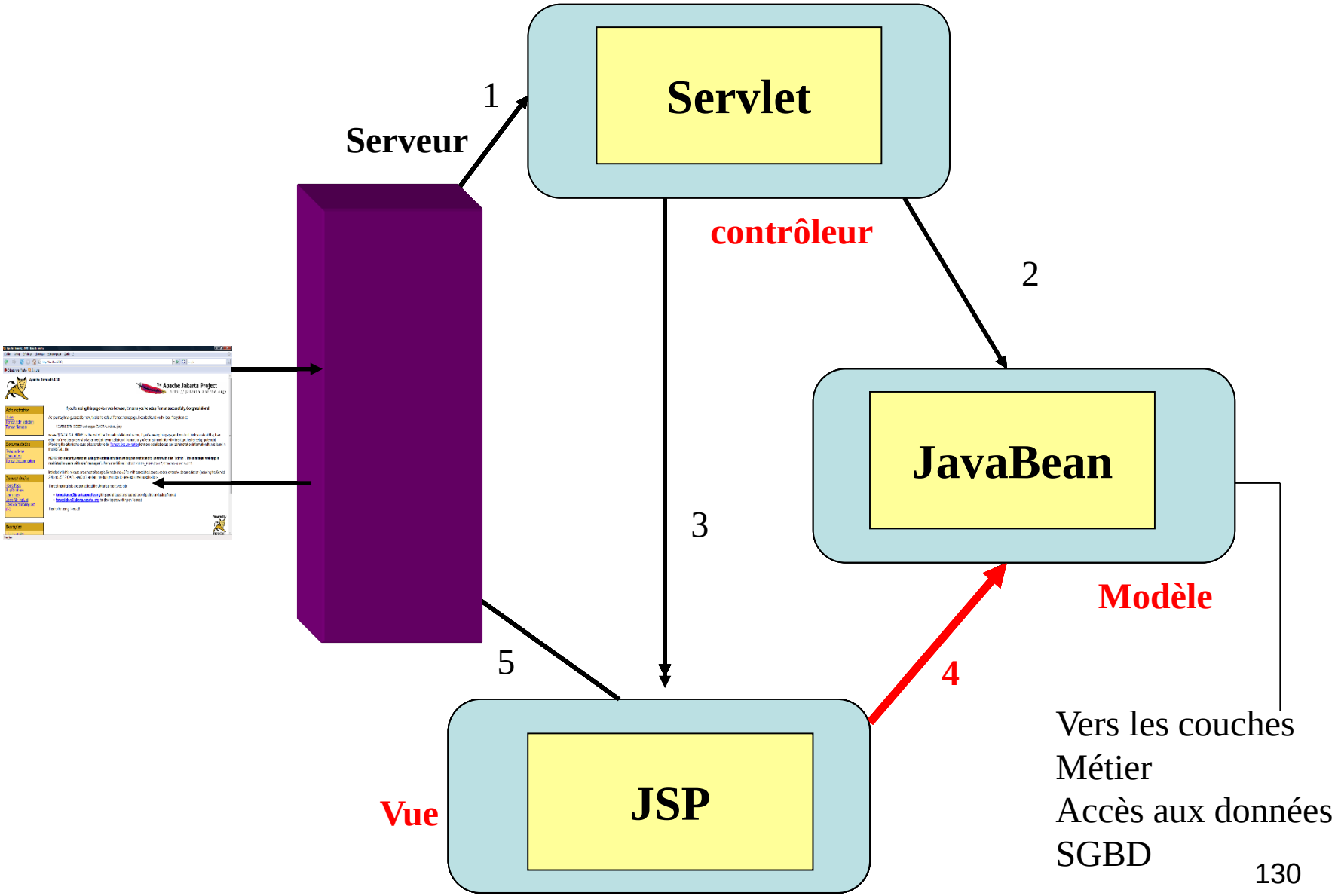
Un framework MVC implémente le ***contrôleur façade***, les mécanismes de gestion du modèle ainsi que ceux de sélection et de construction de la vue.
L'utilisateur d'un tel framework a en charge le développement et la configuration des *workers*.

Le design pattern MVC

L'utilisation de l'architecture MVC offre plusieurs avantages:

- **Fiabilité**: les couches présentation et transaction sont clairement séparées, ce qui permet de modifier l'aspect d'une application sans recompiler le modèle ni le contrôleur.
- **Réutilisabilité et adaptabilité**: l'architecture MVC autorise l'utilisation de plusieurs vues, accédant toutes au même programme sur le serveur, par exemple des navigateurs WEB (HTTP) ou des terminaux sans fil.
- **coûts de développement et d'évolution réduits**: l'architecture MVC permet de confier le développement et la maintenance des interfaces utilisateur à des programmeurs moins expérimentés.
- **Déploiement rapide**: les délais de développement peuvent être considérablement diminués car les développeurs du contrôleur (programmeurs java) se consacrent uniquement aux transactions tandis que les développeurs des vues (programmeurs HTML et JSP) ne sont concernés que par la présentation.
- **Maintenance facilitée**: la séparation entre la logique métier (*business logic*) et la présentation facilite la maintenance et l'évolution des applications Web basées sur Struts.

Collaboration de Servlets et JSP : architecture MVC



Voici les différentes étapes décrites dans le schéma précèdent :

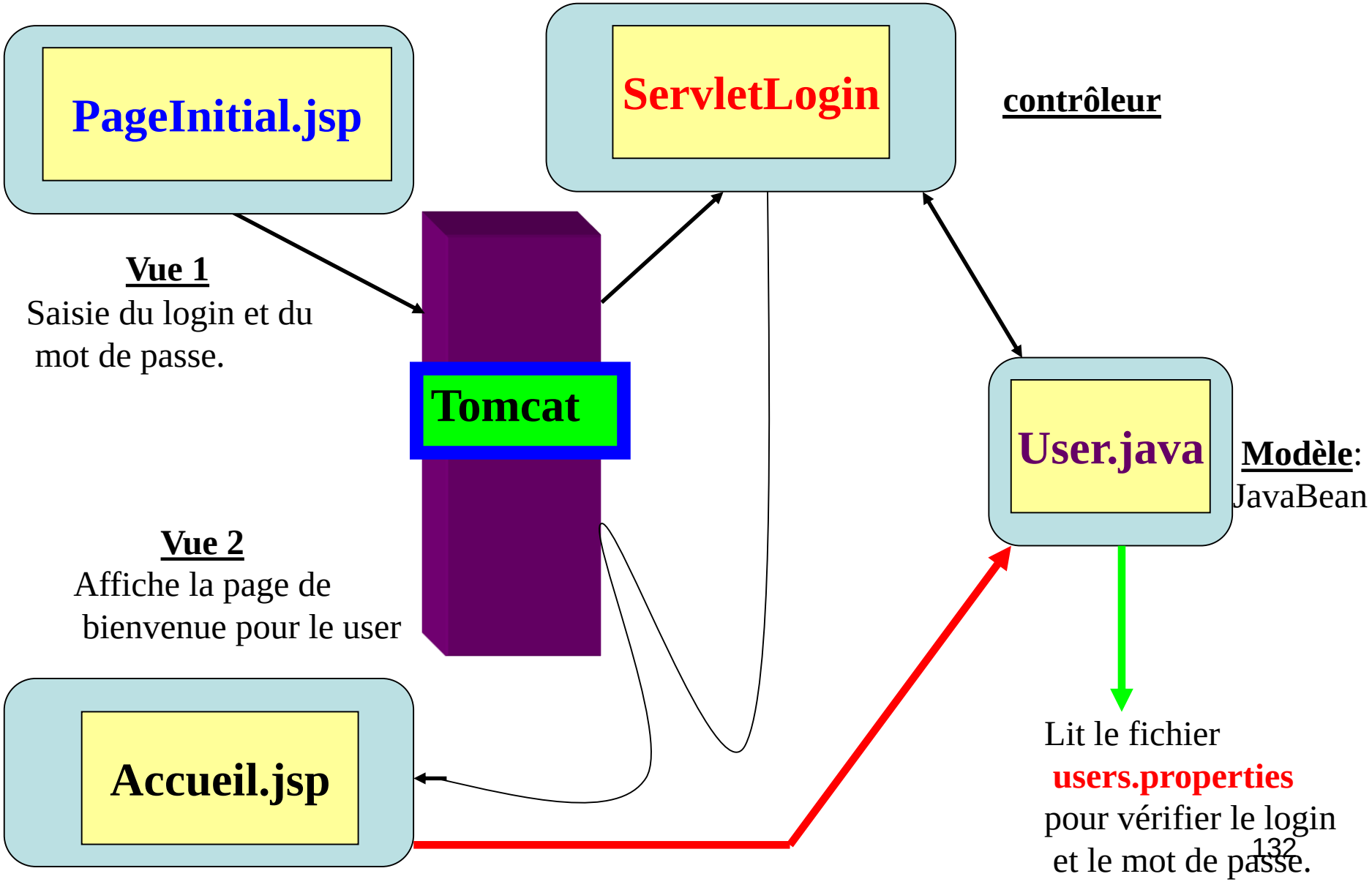
1. L'utilisateur émet une requête auprès du serveur en utilisant la méthode **POST** ou **GET**.
2. Le contrôleur ***accède*** aux informations demandées au travers d'un JavaBean.
3. Le contrôleur transmet le JavaBean à la page JSP chargée de la mise en forme du document.
4. La page JSP ***utilise*** les informations du JavaBean transmit.
5. La page JSP met en forme les informations et retourne la réponse à l'utilisateur.

Afin de mieux comprendre cette architecture et surtout la manière de la mettre en oeuvre, nous allons réaliser un premier exemple qui a pour objectif d'afficher une page d'accueil (***Accueil.jsp***) après que l'utilisateur se soit authentifié sur le site (***PageInitial.jsp***).

Commençons tout d'abord par le **JavaBean** (***User.java***).
Ce dernier acceptera les connexions des utilisateurs enregistrés dans le fichier « **users.properties** » dont voici le format.

```
# users.properties
#
# login = firstName ; lastName ; userId ; password
#
alwin = joseph; ndong ;124;a8b22
pat = Pierre;Diouf;584;ped11
```

Architecture de cette application :



```
package mvc.essai;
import java.util.Properties; import java.io.FileInputStream;
import java.util.StringTokenizer;

public class User implements java.io.Serializable {
private String firstName;
private String lastName;
private int userId;
public User ( ) { }
public String getFirstName ( ) { return firstName;     }
public String getLastName ( ) {  return lastName;      }
public int      getUserId       ( ) {  return userId;          }
public void connect (String user, String password) throws Exception {
// Contrôle des paramètres en entrée
if( user == "" || user == null || password == "" || password == null) {
                        throw new Exception ("Utilisateur ou mot de passe non renseigné !");
      }
```

User.java

```
else {
        // Lecture du fichier des utilisateurs users.properties
         Properties prop = new Properties ( );
         FileInputStream in = new FileInputStream ("F:/usersproperties/users.properties");
         prop.load (in);
String pUser = prop.getProperty (user);
if ( pUser  = = null)
throw new Exception ("Utilisateur inconnu");
StringTokenizer iUser = new StringTokenizer (pUser,";");
              String iFirstName = iUser.nextToken ( );
              String iLastName = iUser.nextToken ( );
              String iUserId = iUser.nextToken ( );
              String iPassword = iUser.nextToken ( );
if (iPassword.equalsIgnoreCase (password)) {
                                                this.firstName = iFirstName;
                                                this.lastName = iLastName;
                                                this.userId = Integer.parseInt (iUserId);
                                              }
else
                throw new Exception ("Mot de passe invalide");
   }
} }
```


User.java (suite)

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<HTML>
<BODY>
<H1>Identification</H1>
<FORM action="/webmvc/servlet/mvc.essai.ServletLogin" method="post">
User name : <INPUT TYPE="text" NAME="userName" value="pat"><br>
Password : <INPUT TYPE="password" NAME="password" value="ped11"><br>
<INPUT TYPE="SUBMIT" NAME="Submit" VALUE="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

Ou bien **/webmvc/logservlet**
webmvc est le nom du projet web et **logservlet** **est** défini dans *<url-pattern>* dans le fichier **web.xml**

PageInitial.jsp

Nous allons utiliser une page (***PageInitial.jsp***) qui dispose d'un formulaire d'identification. Le bouton de soumission du formulaire donnera la main à la servlet contrôleur (***ServletLogin***) qui aura la charge d'invoquer le Bean puis de router la requête vers une page d'accueil personnalisée (***Accueil.jsp***) qui affiche un message de bienvenue pour l'utilisateur authentifié.

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">

<jsp:useBean id="user" class="mvc.essai.User" scope="request"/>

<HTML>
<HEAD><TITLE>accueil</TITLE></HEAD>
<BODY>

Bonjour <%= user.getFirstName ( ) %> <%= user.getLastName ( ) %>

</BODY>
</HTML>
```

Accueil.jsp

```
package mvc.essai;
import java.io.IOException;                import javax.servlet.*;
import javax.servlet.ServletException; import javax.servlet.http.*;
import javax.servlet.http.HttpServlet;   import java.io.*;

public class ServletLogin extends HttpServlet {
private static final String CONTENT_TYPE = "text/html";
private User user;
/**Initialize global variables*/
public void init (ServletConfig config) throws ServletException {
                                                                super.init (config);
                                                              }
/**Process the HTTP Get request*/
public void doGet (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
                                                        doPost (request,response);
                                                      }
```

ServletLogin.java

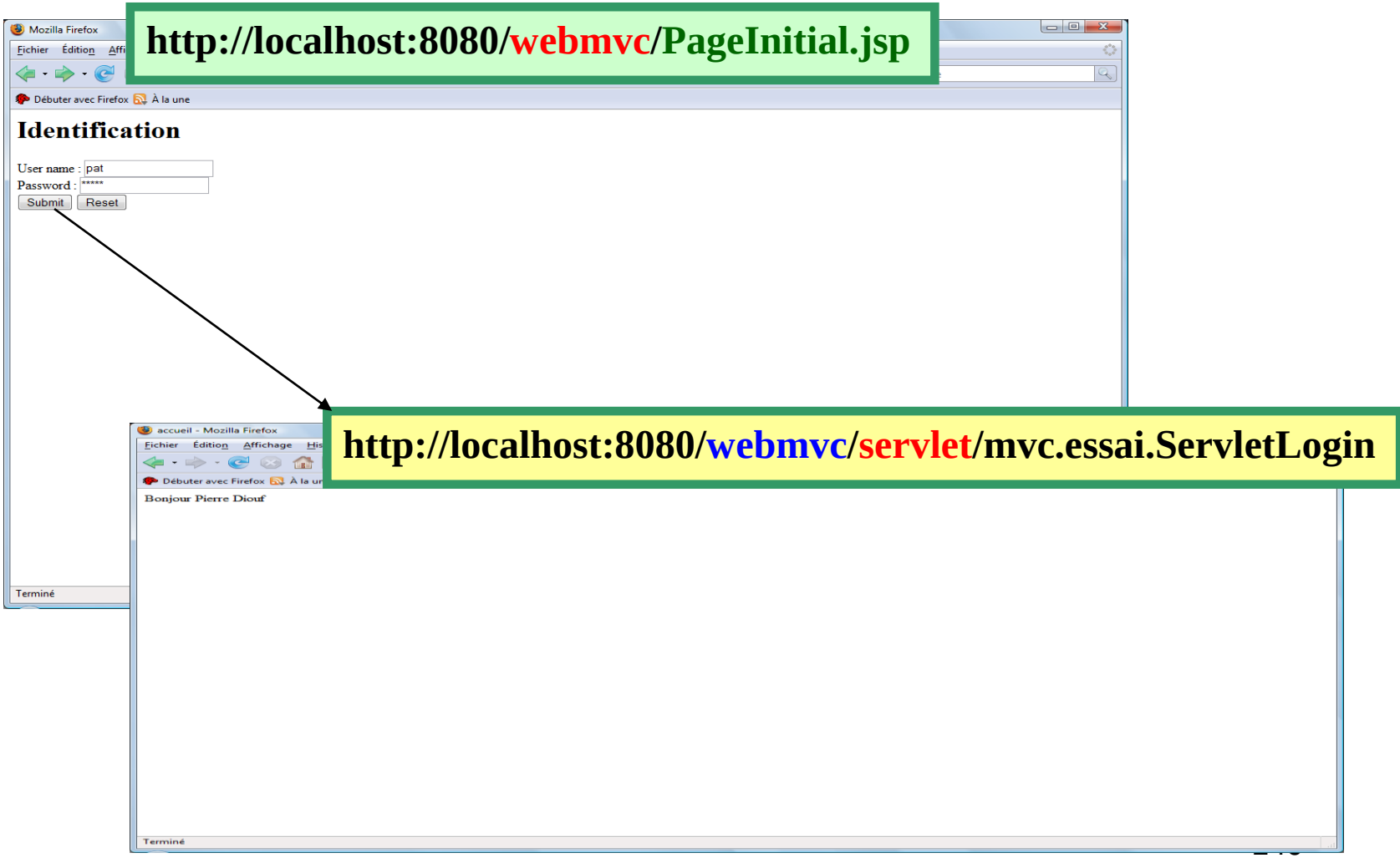
```
/**Process the HTTP Post request*/
public void doPost (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
// récupération des paramètres pUserName et pPassword du formulaire
String pUserName = request.getParameter ("userName");
String pPassword = request.getParameter ("password");

// on initialise le JavaBean (la servlet accède ici au JavaBean)
user = new User ( );
try {
// utilisation de la méthode de connexion 'connect' du JavaBean
user.connect (pUserName, pPassword);
// On transmet le JavaBean au context afin d'être récupéré (utilisé) par la page JSP
request.setAttribute ("user",user);
// On re-route l'appel vers la page d'accueil JSP
gotoPage ("/Accueil.jsp", request, response);
}
```

ServletLogin.java

```
catch (Exception e) {
                    response.setContentType (CONTENT_TYPE);
                    PrintWriter out = response.getWriter ( );
                    out.println ("<html>");
                    out.println ("<head><title>Erreur</title></head>");
                    out.println ("<body>");
                    out.println ("<p>Erreur : " + e.getMessage ( ) +"</p>");
                    out.println ("</body></html>");
    }
}
private void gotoPage (String adress, HttpServletRequest request,
                HttpServletResponse response) throws ServletException, IOException {
      RequestDispatcher dispatcher = getServletContext ( ).getRequestDispatcher(adress);
      dispatcher.forward (request,response);
  }
}
```

ServletLogin.java

Voici un exemple de sortie:



Architecture MVC (en règle générale)

- Le contrôleur est géré par les Servlets (pas de sortie dans la réponse)
- Le modèle est géré par les JavaBeans (ne connaît pas le contrôleur ni la vue)
- La vue est gérée par les JSP (pas de contrôleur)

Besoin de savoir communiquer entre des Servlets et des JSP

- Requête envoyée par le client (1)
- Communication entre le contrôleur et le modèle (2)
- Transmission à la page JSP des informations du JavaBean (3)
- La page JSP **utilise** les informations du JavaBean traduit (4)
- La vue est envoyée en réponse au client (5)

Panoplie d’outils

Nous allons à présent introduire une suite d’outils (frameworks) dédiés surtout à la technologie
JEE.
Ils permettent au développeur de bien cerner les étapes dans la mise en œuvre d’un projet JEE de bout en bout.

Framework STRUTS

Jakarta Struts est un projet open source soutenu par *Apache Software Foundation*. Il s'agit d'une implémentation coté serveur du design pattern **Modèle-Vue-Controleur** (*Model-View-Controller*) ou MVC.

Le projet Struts a été conçu dans le but de fournir un cadre de développement open source pour des applications Web permettant de séparer aisément la couche présentation des couches transactions/données.

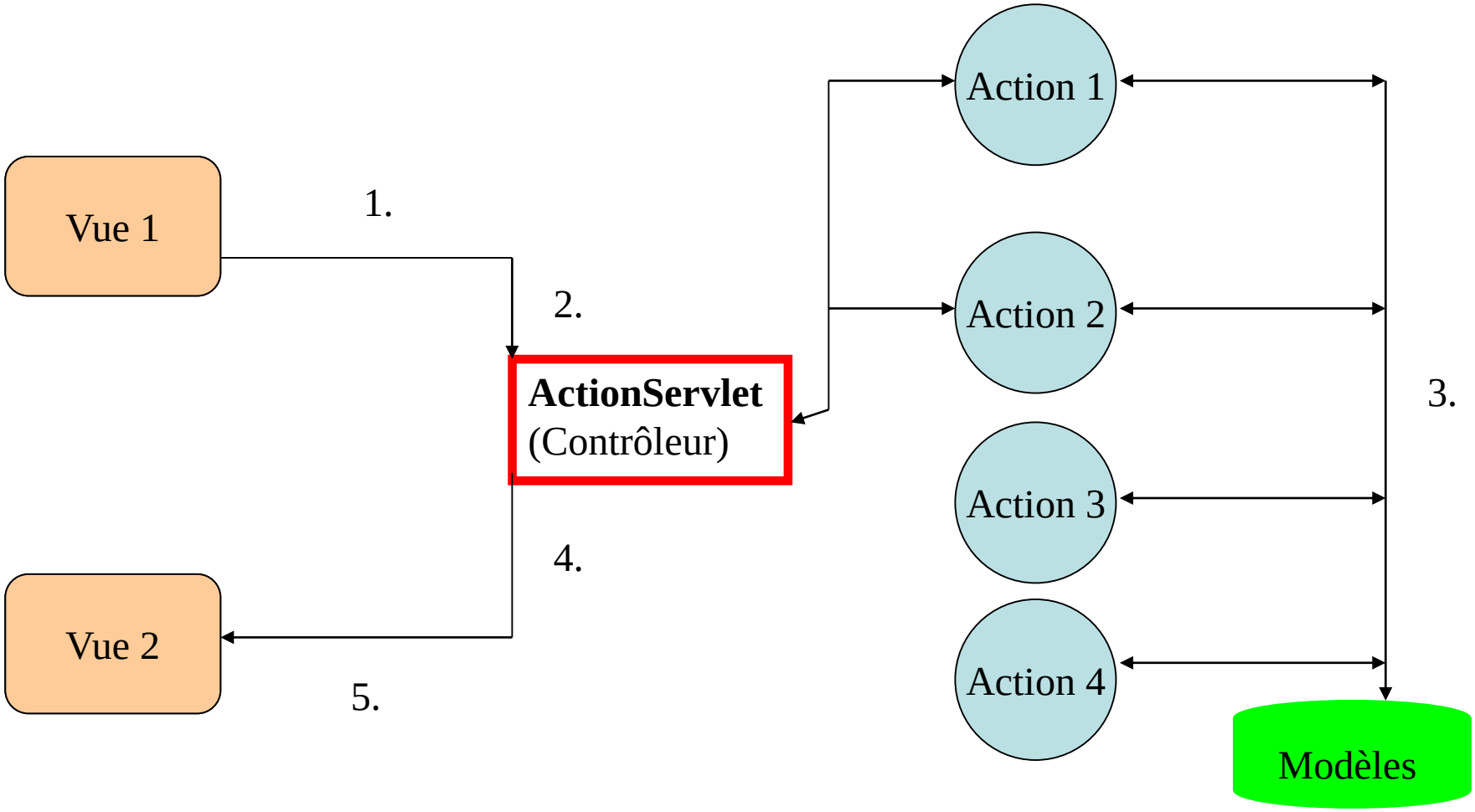
Dans cette présentation, j'utilise **Struts 1.3** et **Tomcat 5.5**.

Vous pouvez télécharger cette version de Struts à l'adresse http://jakarta.apache.org/.

Au niveau de Struts, les classes *ActionServlet*, *Action*, *ActionForm* et le fichier *struts-config.xml* gèrent le *Controller*.

L'implémentation MVC de Struts

Struts implémente l'architecture MVC de Struts côté serveur à l'aide d'une combinaison de pages JSP, de balises ou tags personnalisées et de servlets Java.



La figure montre le chemin par lequel une requête dans une application Struts. Le processus peut être divisé en cinq étapes fondamentales mettant en œuvre la servlet **ActionServlet** et les classes **Action**.

1. Une requête est envoyée depuis une page précédemment affichée.
2. La requête est reçue par la servlet ***ActionServlet***, qui fait fonction de contrôleur. Celle-ci recherche l'URI correspondant dans un fichier XML, et détermine le nom de la classe **Action** capable d'effectuer le traitement correspondant.
3. La classe Action effectue le traitement en utilisant les composants du modèle associé à l'application.
4. Lorsque la classe Action a terminé le traitement, elle rend le contrôle à la servlet ActionServlet tout en lui transmettant une clé indiquant le résultat. La servlet ActionServlet décide, en fonction de cette clé, si le résultat doit être transmis pour présentation.
5. La requête est terminée lorsque la servlet *ActionServlet* répond en transmettant celle-ci à la vue correspondant à la clé retournée. La vue affiche le résultat du traitement.

Le modèle:

Struts ne fournit aucun composant particulier pour le modèle.

La vue:

Chaque vue correspond à une page JSP pouvant contenir une combinaison quelconque de balises

Le contrôleur

Le contrôleur constitue le cœur d'une application Web Struts. Il est implémenté à 'aide d'une servlet nommée **org.apache.struts.action.ActionServlet** .
Cette servlet reçoit les requêtes des clients et les transmet à des classes **org.apache.struts.action.Action** définies par le développeur de l'application.
Le dispatching des requêtes est fait en fonction de l'URI de celles-ci. Lorsqu'une classe *Action* a terminé son traitement, elle retourne **une clé** à la servlet ***ActionServlet***. A l'aide de cette clé, la servlet détermine la *vue* qui sera employée pour présenter les résultats du traitement. La servlet ***ActionServlet*** est similaire à un composant de type *factory* qui crée des objets ***Action*** pour effectuer les traitements métier de l'application.

Installation de Struts

La dernière version de Struts est téléchargeable depuis **http://jakarta.apache.org**
Effectuer les étapes suivantes pour l'application que vous souhaitez déployer.
NB: *j 'ai utilisé ici la version* **struts-1.3.8**

1. Décompressez l'archive Struts sur votre disque dur.
2. Créez une application Web en utilisant la structure de répertoires décrites au chapitre sur Tomcat. Supposons ici que notre application Web s'appelle ***essaistruts***.
3. Copier tous les fichiers JAR (extraits de l'archive de Struts) dans le répertoire **/WEB-INF /lib** de votre application web.

4. On dispose d'un fichier **web.xml** *minimal* dans le répertoire **/WEB-INF** .

```
<?xml version="1.0" ?>

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

<web-app>

</web-app>
```

Installation de Struts

Créer un fichier de configuration **struts-config.xml** et copier le dans le répertoire **/WEB-INF** de votre application web**. Le fichier struts-config.xml** est le descripteur de déploiement des applications Struts. Les informations qu'il contient permettent d'assembler les différents composants de l'architecture MVC. Il est normalement placé dans le répertoire **/WEB-INF**

NB: dans son implémentation, Struts utilise deux notions principales, qui sont représentées par les classes **org.apache.struts.action.Action** , appelées actions (il s'agit des actions de l' utilisateur, et **org.apache.struts.action.ActionForm**, appelées **FormBeans** ou formulaires (ces derniers sont des JavaBeans standards).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
 Configuration 1.1//EN" "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

<message-resources
            parameter="eyrolles.ApplicationResources" />
</struts-config>
```

Le fichier strust-config.xml

Note: *The struts-config.xml file is the configuration file that defines the flow in Struts web applications. This configuration file defines:*

- ***data sources*** *(parameters needed to use JDBC objects),*
- ***form beans*** *(mappings/definitions for beans that hold information that has been submitted by the user from a web form),*
- ***forwards*** *(mappings that provide an easy way to send the user to a specified page depending on business logic) and*
- ***actions*** *(mapping to an object that handles business logic).*

Configuration de Struts

Struts se fonde sur une servlet très évoluée pour assurer sa fonction de contrôleur générique. Comme pour toute servlet, il faut la configurer dans le fichier **web.xml** .

```
(….)
<servlet>
        <servlet-name> action </servlet-name>
        <servlet-class> org.apache.struts.action.ActionServlet </servlet-class>
         <load-on-startup>1 </load-on-startup>
</servlet>
(….)
<servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
</servlet-mapping>
(….)
```

Convention de nommage arbitraire qui veut que toutes las *actions Struts* se terminent par l'extension **.do**
ex. http://localhost/example/test.do .

ActionServlet sera configurée dans le fichier **struts-config.xml** .

Première application Struts

Cette petite application consiste en un ensemble de pages JSP permettant à un utilisateur d'entrer un symbole d'une action cotée en bourse pour obtenir sa valeur.

Le fait qu'une application *Struts* soit modélisée selon le design ***pattern MVC*** nous permet de suivre un processus de développement standard. Ce processus commence par l'identification des vues de l'application, des contrôleurs manipulant ces vues et du modèle contenant la logique du système.

Voici l'ensemble des étapes principales à suivre:

1. Définition et création des vues qui représentent l'interface fonctionnelle de notre application et ajout de tous les **ActionForms** utilisés par celles-ci dans le fichier **struts-config.xml** .
2. Création des composants des contrôleurs de l'application
3. Définition des relations entre les vues et les contrôleurs dans **struts-config.xml**
4. Description des composants Struts dans le fichier **web.xml**
5. Lancement de l'application.

Création des vues

indexstruts.jsp

```
<%@ page language ="java" %>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<body>
<table width="500", border="0", cellspacing="0" cellpadding="0">
<tr>    <td> </td>  </tr>
<tr bgcolor="#000000">
  <td height="30" width="162">
  <div align="left">
                        <img
                              src="barreseule.gif"    width="162"   height="30">
  </div>
  </td>
  </tr>
  <tr>
  <td> </td>
  </tr>
</table>
```

URL à laquelle le formulaire sera envoyé.

```
<html:form action="/Lookup" >
                    <bean:define
                                id="symbol"
                                name="lookupForm"
                                type="joe.LookupForm"   />
  <table width="45%" border="0">
        <tr>
            <td>        Symbole:              </td>
            <td> <html:text property="symbol" />   </td>
        </tr>
        <tr>
            <td colspan="2" align="center">   <html:submit />     </td>
        </tr>
  </table>
  </html:form>
 </body>

</html>
```

indexstruts.jsp

```
<%@ page language="java" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<body>          <table width="500", border="0", cellspacing="0" cellpadding="0">
                 <tr>
                      <td> </td>
                 </tr>
<tr bgcolor="#000000">
                          <td height="30" width="162">
                                   <div align="left">
                                        <img
                                             src="barreseule.gif"  width="162"
                                                                        height="30">
                                   </div>
                          </td>
  </tr>
  <tr>
        <td> </td>
  </tr>
 <tr>
          <td>   Cours actuel:<%=request.getAttribute ("COURS")%>   </td>
  </tr>
</body></html>
```

cotation.jsp

La vue ***cotation.jsp*** est affichée lorsque la recherche du symbole d'une action a réussi.
Ici c'est une simple page JSP ne contenant aucune balise struts spécifique.

Il existe trois types de balises struts: **Bean**, **HTML** et **Logic**.
Nous avons utilisé les deux premiers dans la page ***indexstruts.jsp*** . C'est cette page que les utilisateurs verront lorsqu'il se connecteront à l'application. Cette page sert à saisir un symbole représenter une action et d'envoyer celui-ci à l'application.

Les balises de struts (comme **<html: form />**) encapsulent le traitement des formulaires Struts.
La balise **<bean:define />** définit le JavaBean utilisé (ie l' ***ActionForm***).
cette balise contient les attributs suivants:

id: spécifie le nom de l'attribut du bean

name: identifie la clé par laquelle sera référencée *l'ActionForm*. La valeur ici est *lookupForm* un *ActionForm* est un objet utilisé par Struts pour représenter les données du formulaire sous la forme d'un JavaBean.

type: cet attribut donne le nom qualifié de la classe bean utilisé pour cette requête.

<html:text /> correspond à **<input type='text' >,** sauf qu'ici cette balise est associée à un attribut **property** qui contient le nom d'un membre de la classe du bean *ActionForm*.

Lorsque l'utilisateur clique sur le bouton <u>submit</u>, l'objet *ActionForm* désigné par la balise ***<html:form />*** est créé et reçoit les données contenues dans les balises ***<html:text />.***
Il est ensuite enregistré dans la <u>*session*</u>.

Comment utiliser les balises Struts ?

Il faut ajouter deux entrées dans le fichier **web.xml** pour référencer respectivement les bibliothèques **/WEB-INF/tlds/struts-html.tld** et **/WEB-INF/tlds/struts-bean.tld**. Cela se fait en copiant ces lignes suivantes dans le descripteur de déploiement:

```
<jsp-config>
<taglib>
        <taglib-uri>/WEB-INF/tlds/struts-html.tld</taglib-uri>
        <taglib-location>/WEB-INF/tlds/struts-html.tld</taglib-location>
 </taglib>

<taglib>
   <taglib-uri>/WEB-INF/tlds/struts-bean.tld</taglib-uri>
   <taglib-location>/WEB-INF/tlds/struts-bean.tld</taglib-location>
 </taglib>
</jsp-config>
```

ATTENTION: il faut copier les fichiers **.tld** (extraits de l'archive de struts) dans un répertoire **tlds** (à créer) sous le répertoire **WEB-INF** de votre application web.

Commentaires dur **l'ActionForm** qui est notre seul *JavaBean*:

Cette classe *ActionForm* contient un seul membre correspondant au symbole entré dans le formulaire de la vue ***indexstruts.jsp***.
Il faut savoir que lorsqu'un formulaire **<html:form />** est envoyé au serveur, les membres de l' *ActionForm* reçoivent les valeurs provenant de la balise **<html:text />**: cela est réalisé grâce à la **réflexion** (ou *introspection*) en Java. C'est pourquoi à ce niveau, vous noterez que les *accesseurs* de l'**ActionForm** doivent nécessairement se conformer au standard **JavaBean**.

```
private String symbol = null;
public void setSymbol (String symbol)
public String getSymbol ( )
```

Ci après nous donnons la classe java représentant notre bean **LookupForm.java** .

```
package joe;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import javax.servlet.http.HttpServletRequest;

public class LookupForm extends ActionForm {

private String symbol = null;
public String getSymbol ( ) {
                    return symbol;
                }
public void setSymbol (String symbol) {
                    this.symbol=symbol;
                }
public void reset (ActionMapping mapping, HttpServletRequest request){
this.symbol = null;
  }
}
```

La méthode **reset** est spécifique aux beans ***ActionForm***, elle est appelée par struts à chaque requête utilisant l'objet **LookupForm**. *Son rôle est de réinitialiser les membres afin que l'objet puisse être placé dans un pool en vue d'une future utilisation*.

ATTENTION: avant de créer cette classe ajouter le fichier **struts-core-1.3.8.jar** dans le *classpath* du projet sous **Eclipse** pour pouvoir faire les importations des classes ***ActionForm*** et ***ActionMapping***.

Pour le déploiement du bean, une fois compilée, sa classe (ie *LookupForm.class*) doit être copiée dans le répertoire *classes* de l'application web (comme avec les servlets). Ensuite le fichier **struts-config.xml** doit contenir une entrée contenant:

```
<form-beans>
        <form-bean name="lookupForm" type="joe.LookupForm" />
</form-beans>
```

Cette entrée informe l'application Struts de la présence du bean ***LookupForm*** et de la manière dont il doit être référencé.

A présent voyons comment créer les composants du contrôleur.

Contrôleur:
Dans une application Struts le contrôleur est composé de deux composants, représentés par les classes **org.apache.struts.action.ActionServlet** et **org.apache.struts.action.Action** .

Vous noterez que la plupart des applications struts possèdent une seule implémentation de la classe **org.apache.struts.action.ActionServlet** et plusieurs implémentations de la classe **org.apache.struts.action.Action.**

ActionServlet gère les requêtes client et détermine laquelle des classes **Action** doit effectuer le traitement.
NB: il existe des cas complexes où il faut créer une implémentation spécifique de cette classe. Mais pour notre exemple l'implémentation par défaut convient parfaitement.

Pour ce qui est de la classe **Action**, elle doit être implantée pour chaque fonctionnalité de notre application *puisque c'est elle qui contient la logique spécifique à l'application*.

Dans cet exemple, il y a un seul traitement à faire: rechercher le cours de l'action correspondant au symbole entré par l'utilisateur.
Nous allons donc créer une classe **LookupAction** pour cela.

Vous noterez bien que la méthode la plus importante cette classe est **execute ( )**.

```
package joe;
import java.io.IOException; import javax.servlet.ServletException;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class LookupAction  extends Action  {

protected Double getCotation (String symbole){
 if (symbole.equalsIgnoreCase ("SUNW")) return new Double(25.00);
                                        return null;
}
public ActionForward  execute  (ActionMapping mapping, ActionForm form,
                                                        HttpServletRequest request,
HttpServletResponse response) throws IOException,ServletException{
Double cours = null;
//cible par défaut: clé employée pour déterminer la vue utilisée pour afficher le résultat
String cible = new String ("succes");
```

```
if (form != null){
                    //utilisation de LookupForm pour obtenir les parametres de la requete
                    // lorsque le formulaire sera envoyé, ActionServlet utilisera la
                    // réflexion pour initialiser les variables membres de LookupForm
                    LookupForm lookupForm = (LookupForm) form;
                    String symbol = lookupForm.getSymbol ( );
                    cours = getCotation (symbol);
                }

//cible en cas d'echec
if (cours == null){
                        cible = new String ("echec");
                    }
else request.setAttribute ("COURS",cours);
//transmission a la vue appropriée
return mapping.findForward (cible);
}
}
```

LookupAction.java

Commentaires:

La méthode **execute ( )** contient l'essentiel des fonctionnalités de **LookupAction**, elle doit être implémentée dans toutes les classes **Action**.
Cette méthode possède plusieurs paramètres:

-**ActionMapping**: contient les informations de déploiement d'un bean **Action** particulier. *Elle détermine où les résultats de l'action devront être envoyées une fois le traitement terminé*.

- **ActionForm**: représente le formulaire contenant les paramètres de la requête venant de la vue référencée par le bean Action. La référence passée à **LookupAction** pointe ici vers une instance de **LookupForm**.

- **HttpServletRequest**: référence à l'objet ***request*** de la requête HTTP en cours de traitement.

-**HttpServletResponse**: référence à l'objet ***response*** de la requête HTTP en cours de traitement.

Pour déployer cette classe, vous devrez ajouter les lignes suivantes au fichier de configuration de struts.

```
<action-mappings>
        <action path ="/Lookup"
                type ="joe.LookupAction"
                name ="lookupForm" >
                <forward name ="succes" path ="/cotation.jsp"    />
                <forward name ="echec" path ="/indexstruts.jsp" />
        </action>
</action-mappings>
```

Cette entrée contient les données qui seront stockées dans l'objet **ActionMapping** passé à la méthode **execute ( )** de **LookupForm**.
Elle définit tous les attributs nécessaires pour utiliser cette instance de **LookupAction**, dont une série de sous éléments <**forward** > représentant les différentes vues pouvant être employées pour afficher les résultats en fonction de la valeur de **cible**.

Passons maintenant au déploiement de notre application:

Certaines modifications sont à apporter au fichier ***web.xml***:
- l'informer de la présence de la servlet **ActionServlet**:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- La seconde modification à apporter au fichier ***web.xml***:

Ce mapping indique à l'application Web que lorsqu'une requête dont l'URL se termine par **.do** est reçue, celle-ci doit être traitée par la servlet nommée **action**

```
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

La troisième modification (non obligatoire) précise la page par défaut ouverte lors du lancement de l'application.

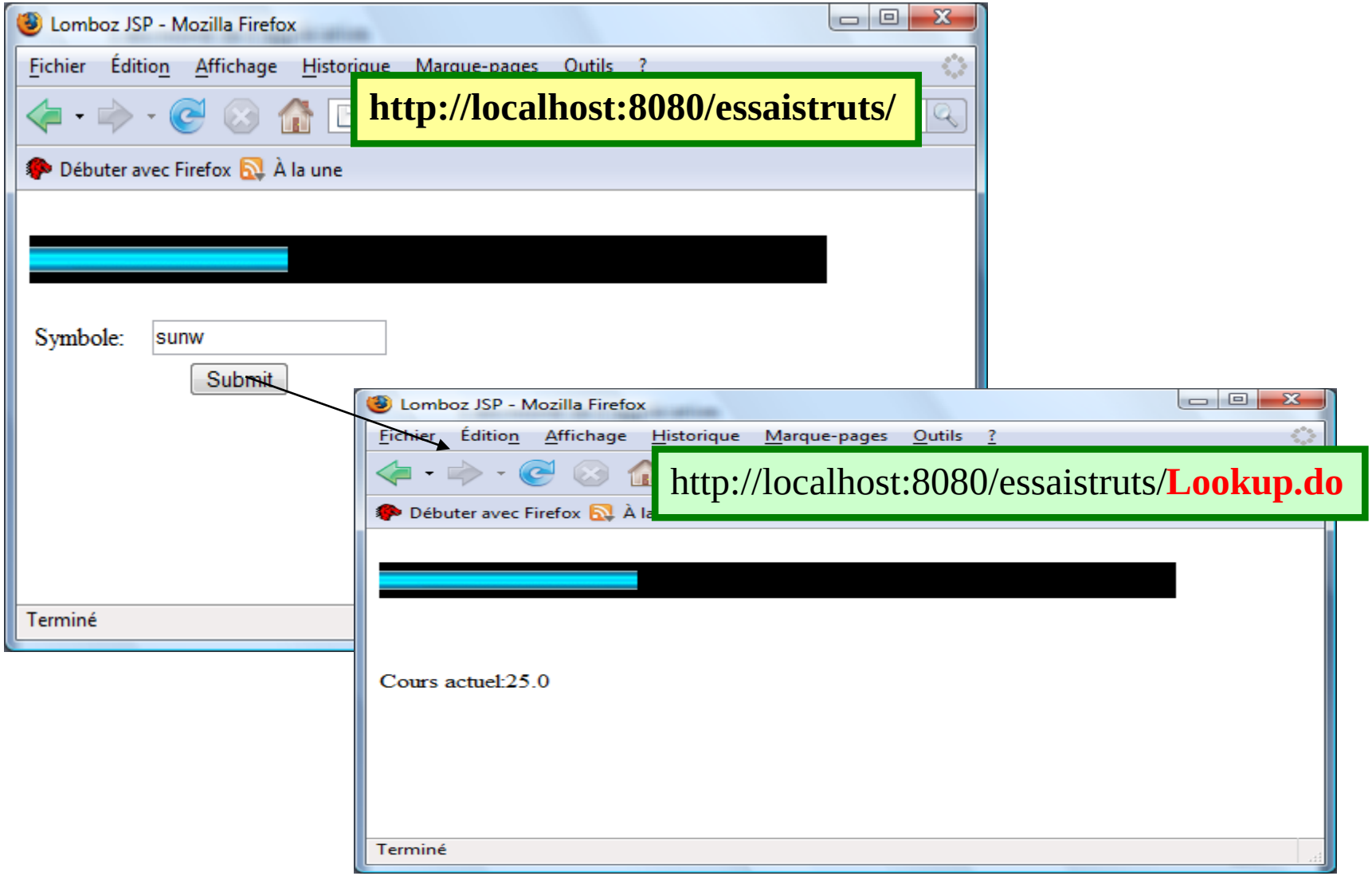
```
<welcome-file-list>
                    <welcome-file>indexstruts.jsp</welcome-file>
</welcome-file-list>
```

Voici à présent le fichier complet web.xml :

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN
"
 "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
                        <param-name>config</param-name>
                        <param-value>/WEB-INF/struts-config.xml</param-value>
         </init-param>
                <load-on-startup>1</load-on-startup>
 </servlet>
  <servlet-mapping>
                        <servlet-name>action</servlet-name>
                        <url-pattern>*.do</url-pattern>
  </servlet-mapping>
```

```
<jsp-config>
<taglib>
        <taglib-uri>/WEB-INF/tlds/struts-html.tld</taglib-uri>
        <taglib-location>/WEB-INF/tlds/struts-html.tld</taglib-location>
 </taglib>
<taglib>
        <taglib-uri>/WEB-INF/tlds/struts-bean.tld</taglib-uri>
        <taglib-location>/WEB-INF/tlds/struts-bean.tld</taglib-location>
 </taglib>
</jsp-config>

<welcome-file-list>
                        <welcome-file>indexstruts.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

Lancement de l'application



Différentes étapes dans le fonctionnement de l'application:

- on tape la valeur **sunw** dans la vue ***indexstruts.jsp*** présentée à l'utilisateur puis on clique sur le bouton submit,
- le clic sur submit provoque un appel à l'URL nommée par l'attribut action de **<html:form >** c'est-à-dire **Lookup**.

Lorsque le conteneur de servlet/JSP reçoit cette requête, il cherche dans *web.xml* un élément **<servlet-mapping >** possèdant un sous élément <*url-pattern*> se terminant par **.do**

- on trouve alors la servlet dont le nom est action : il s'agit de la servlet ***ActionServlet***
- ***ActionServlet*** traite la requête en accèdent à l'objet *LookupForm*, elle affecte à ses variables membres les données présentes dans la requête, puis place l'objet *LookupForm* dans la session
- ensuite ***ActionServlet*** recherche dans ***struts-config.xml*** une entrée <action-mapping> avec un élement **<path>** ègal à Lookup.
- ***ActionServlet*** crée alors une instance de *LookupAction* correspondant à la valeur de type,
- ***ActionServlet*** invoque *execute ( )* qui effectue le traitement puis appelle *findForward ( )* qui cherche un élément <**forward**> dont l'attribut name correspond à la valeur de cible

Elle renvoie un objet de type *ActionForward* contenant le résultat de la recherche ie *citation.jsp* (succès) ou *indexstruts.jsp* (échec)

LookupAction retourne alors l'objet *ActionForward* à *ActionServlet* qui transmet à son tour la requête à la vue sélectionnée pour la présentation.

L'internationalisation dans Struts

L'objectif de l'internationalisation (i18n) des applications est un moyen de configurer vos applications afin qu'elles puissent utiliser différentes langues (Français, Anglais, Italien …)
Struts comporte deux composants i18n.
Le premier est la classe **Message** qui fait référence à un groupe de ressources contenant les **chaînes de caractères localisées**. Cette classe doit être utilisée par le CONTROLEUR.
Le second composant i18n, qui doit être utilisé par les VUES, est une balise personnalisée ***<bean: message />*** pour représenter les chaînes de caractères gérées par le contrôleur.

Voici comment procéder pour gérer cette i18n:

Pour le contrôleur: on doit définir un ensemble de ressources

Il faut créer un ensemble de fichiers de propriétés Java, chacun contenant une paire **clé/valeur** pour chacun des messages que l'application est susceptible d' afficher dans la langue utilisée par le client.
Le **premier de ces fichiers** contient les paires **clé/valeur** pour le langage utilisé.

Ce fichier doit porter un nom de la forme ***nom*.properties** .
Ex: *ApplicationResources.properties.*

Voici un exemple de fichier que je vais utilisé pour internationaliser l'application ***essaistruts***

app.symbole=Symbole
app.cours=Cours Actuel

ApplicationResources.properties

Ici on indique à Struts que si le client utilise un **Locale** par défaut dont la langue est le français et si la clé ***app.symbole*** existe dans le fichier de ressources, la valeur **Symbole** remplacera chaque occurrence de la clé ***app.symbole*** . Même chose pour la clé **app.cours** .

Remarque fondamentale:

Il faut créer un fichier de ressources pour chaque langue utilisée. Ces fichiers doivent être nommés en suivant les mêmes conventions que le fichier par défaut.
Le code ISO de la langue utilisée devra être rajouté au nom du fichier:

Par exemple pour :

- le français: le fichier sera nommé **ApplicationResources_fr.properties**
- l'italien : on nommera le fichier **ApplicationResources_it.properties**

Vous trouverez les codes ISO des différentes langues à l'adresse:

www.old.ics.uci.edu/pub/ietf/http/related/iso639.txt

les fichiers de ressources sont chargés au démarrage de l'application.

Maintenant que vos fichiers de ressources sont créés il faut les déployer pour informer Struts de leur présence.

Déploiement des ressources:

Ajoutez le chemin d'accès aux fichiers dans le fichier **struts-config.xml** en insérant la ligne ci-dessous:

```
<message-resources
                parameter="joe.ApplicationResources" />
```

Ici on indique simplement que tous les fichiers de ressources se trouvent dans le répertoire **/src/joe** (où joe est votre package) de votre projet web dynamique.

NB: l'élément **<message-resources />** ne contient que le nom du fichier par défaut. Struts trouve les autres fichiers en ajoutant le code ISO de la langue en fonction du **Locale** du client.

Regardons à présent ce qu'il faut faire pour les VUES:
On utilise le second composant **<bean:message />**
Commencer par déployer la bibliothèques de balises dans le fichier **web.xml** (cf diapo 193)
Le fichier ***struts-bean.tld*** doit aussi être copié dans l'emplacement idéal comme nous l'avons expliqué dans les sections précédentes.

Au niveau de la page JSP qui doit utiliser la ressource, placez l'instruction

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
```

Maintenant dans le corps de votre JSP, remplacez les chaînes de caractères qui doivent être internationalisées par:

```
<bean:message key='app.symbole' />
```

Remarque: la balise **<bean:message />** possède neuf (9) attributs parmi lesquels on a utilisé l'attribut **key** qui est l'identifiant employé pour trouver un message stocké dans un fichier de ressources. Il est OBLIGATOIRE.

Suivez ces différentes étapes pour internationaliser l'application ***essaistruts***.

Traitement des erreurs dans Struts

Pour traiter les erreurs, on utilise principalement deux classes:

import org.apache.struts.action.**ActionErrors**;
import org.apache.struts.action.**ActionMessage**;

Chaque objet **ActionMessage** constitue un message d'erreur qui peut être stocké dans un fichier de ressources. Le message est stocké sous forme de clé.
La classe **ActionErrors** représente une collection d'instances **d'ActionMessage**.

Le message d'erreur est souvent créé par une instance **d'Action** ou **d' ActionForm** .

Regardons comment ajouter le traitement d'erreurs dans l'application ***essaistruts***.
d'abord comment gérer les erreurs dans **l'ActionForm** (c'est-à-dire dans **LookupForm**)

Commencer par modifier le fichier **struts-config.xml** comme ceci:

```
<action path ="/Lookup"
                    type ="eyrolles.LookupAction"
                    name ="lookupForm"
                      validate= " true "
                      input= " /indexstruts.jsp "  >
                    <forward name ="succes" path ="/cotation.jsp"    />
                    <forward name ="echec" path ="/indexstruts.jsp" />
          </action>
```

L'attribut ***validate*** indique que la validation doit être effectuée par Struts.
L'attribut ***input*** indique le composant d'où proviennent les données et vers lequel les erreurs doivent être retournées.

Maintenant ajouter la définition de la méthode ci-dessous dans la classe ***LookupForm***:

```
public ActionErrors validate (ActionMapping mapping,   HttpServletRequest request)
{
  ActionErrors errors = new ActionErrors (  );
  if (symbol == null || symbol.trim ( ).equals ("")) {
                errors.add ("symbol", new ActionMessage ("error.symbole"));
  }



  return errors;
  }
```

Il faudra nécessairement importer les deux classes suivantes:

```
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMessage;
```

Maintenant dans la page ***indexstruts.jsp***, rajoutez la balise **<html:errors />**

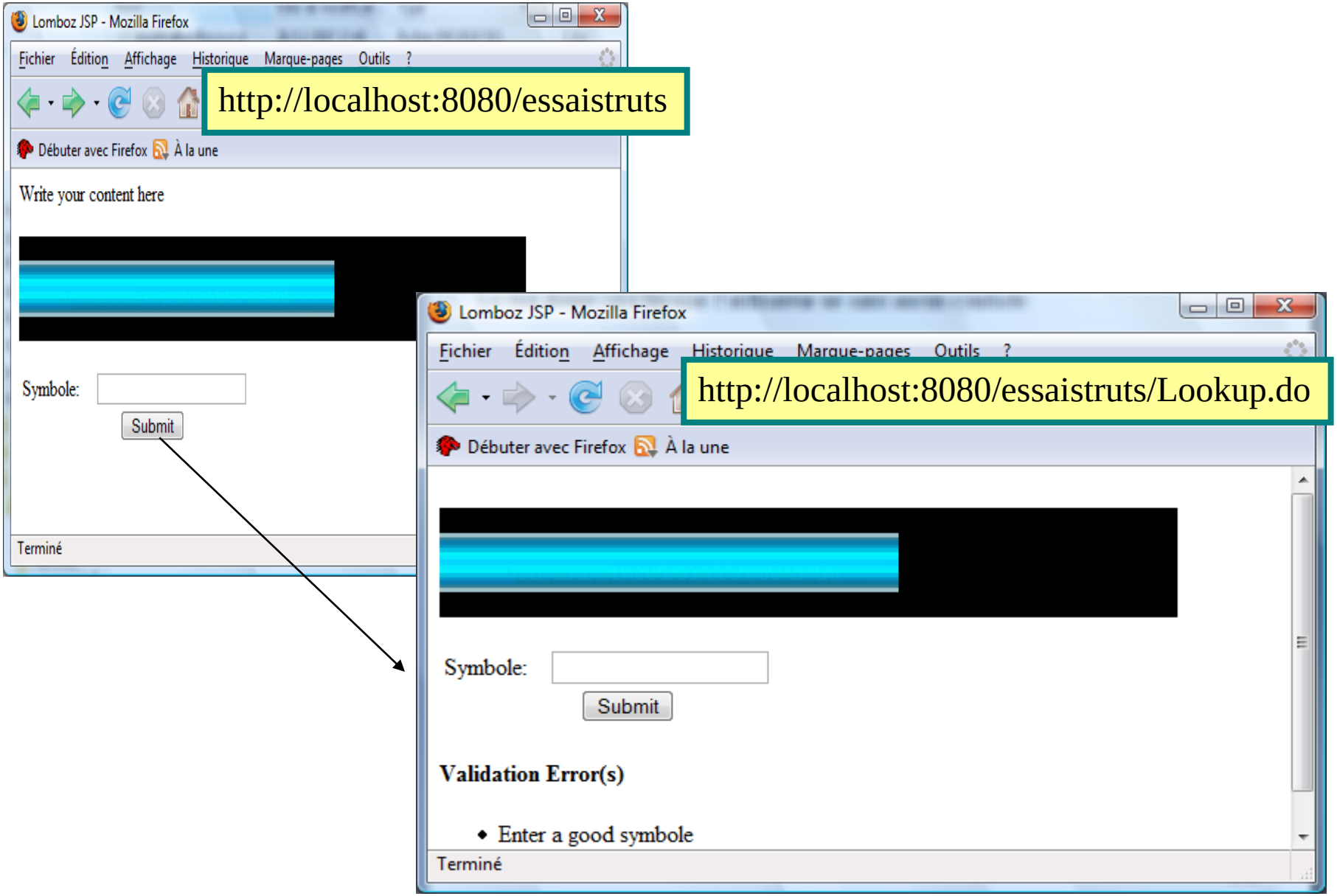
Cette balise permet d'afficher le message d'erreur dans la page.

Et en fin créer le fichier *ApplicationResource.properties* (s'il existe déjà mettez le simplement à jour). L'essentiel est qu'il doit contenir une ligne pour le message d'erreur que l'on veut gérer:

error.symbole=<li>Enter a good symbole

Ce fichier doit aussi être placé dans le classpath de l'application.

Le test donne ceci lorsque l'utilisateur ne saisi aucun symbole:



Maintenant regardons comment gérer les erreurs dans **l'Action** (concrètement dans la classe **LookupAction**).

Le paramétrage que nous avons réalisé dans les sections précédentes pour gérer les erreurs dans *l'ActionForm* demeure sauf que nous utilisons la méthode ***execute ( )*** au lieu de la la méthode ***validate ( )***.

Et voici comment créer cette méthode :

```
public ActionForward execute (ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)
                    throws IOException,ServletException{
Double cours =null;
String symbol=null;
//cible par defaut
String cible = new String ("succes");
 if (form!=null){
//utilisation de LookupForm pour obtenir les parametres de la requete
LookupForm lookupForm = (LookupForm)form;
 symbol = lookupForm.getSymbol ( );
cours = getCotation (symbol);
}
```

```
//cible en cas d'echec
if (cours = = null){
cible = new String ("echec");
 ActionErrors erreurs = new ActionErrors ( );
 erreurs.add ( ActionErrors.GLOBAL_MESSAGE,
                                        new ActionMessage("error.general","symbol"));
 //on signale les erreurs à la page d origine
 if (!erreurs.isEmpty ( ) )
 this.saveErrors(request,erreurs);

}
else request.setAttribute("COURS",cours);
//transmission a la vue appropriée
return mapping.findForward(cible);
}
```

On rajoute la ligne suivante dans le fichier de ressources *ApplicationResource.properties*:
error.general=<li>This symbol does not exist

Si l'utilisateur saisi un symbole qui n'existe pas : il reçoit le message:
This symbol does not

Lomboz JSP - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/essaistruts

Débuter avec Firefox À la une

Symbole: badsymbol

Submit

Terminé

Lomboz JSP - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/essaistruts/Lookup.do

Débuter avec Firefox À la une

Symbole: badsymbol

Submit

Validation Error(s)

- This symbol does not exist

Terminé

JavaBeans Struts: **DynaValidatorActionForm**

Dans l'exemple MVC 2 Struts, nous avons créé notre propre *JavaBean* **LookupForm.java** que nous avons utilisé après *réflexion (introspection)* dans notre classe *Action*, **LookupAction**.

Il est possible de ne pas créer la classe **LookupForm** et demander à Struts de gérer dynamiquement la ***création*** et ***l'initialisation*** des attributs de la classe de notre JavaBean. Cela est possible car Struts nous donne la classe **DynaValidatorActionForm** du package **org.apache.struts.validator** .

Cette classe permet de réduire considérablement le temps de développement.

Reprenons à présent l'exemple précédent en supprimant la classe **LookupForm** et en la remplaçant par la classe décrite ci-dessus.

Je commence par donner les deux pages JSPs ***indexstruts.jsp*** et ***cotation.jsp*** avec quelques modifications non obligatoires dans le code.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/tlds/struts-bean.tld" prefix="bean" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html> <head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Test Struts DynaValidatorActionForm</title>
</head>
<body>
<table width="30", border="0", cellspacing="0" cellpadding="0">
        <html:image src="barreseule.gif" border="30"/>
</table>
<html:form action="/dyna1">
   <table width="45%" border="0">
   <tr>     <td> Symbole: </td> <td> <html:text property="symbol" /></td>
  </tr>
              <tr>   <td colspan="2" align="center">
                     <html:submit value="Envoyer"/></td>
              </tr>
    </table>  </html:form>
</body> </html>
```

indexstruts.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tlds/struts-html.tld" prefix="html" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Affichage cours de la Bourse</title>
</head>
<body>
<table width="500", border="0", cellspacing="0" cellpadding="0">
 <html:image src="barreseule.gif" border="30"/>
 <tr>
            <td>   Cours actuel:<%=request.getAttribute ("COURS")%>   </td>
   </tr>
</table>
</body>
</html>
```

cotation.jsp

Voici comment nous allons maintenant écrire notre classe *Action*, **LookupAction**.

```
package joe;
import java.io.IOException; import javax.servlet.ServletException;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.validator.DynaValidatorActionForm ;
public class LookupAction extends Action {
protected Double getCotation (String symbole){
 if (symbole.equalsIgnoreCase ("SUNW")) return new Double(25.00);
                                                return null;
}
```

```
public ActionForward execute (ActionMapping mapping, ActionForm form,
                                                    HttpServletRequest request,
HttpServletResponse response) throws IOException,ServletException{
Double cours = null;
//cible par défaut: employé pour déterminer la vue utilisée pour afficher le résultat
String cible = new String ("succes");
if (form != null){
   //utilisation de DynaValidatorActionForm pour obtenir
   //les parametres de la requete lorsque le formulaire (indexstruts.jsp) sera envoyé
DynaValidatorActionForm lookupForm = (DynaValidatorActionForm) form;
String symbol = lookupForm.getString ("symbol");
   cours = getCotation (symbol);
}
//cible en cas d'echec
if (cours == null){
      cible = new String ("echec");
   }
else request.setAttribute ("COURS",cours);
//transmission a la vue(page jsp) appropriée
return mapping.findForward (cible);
}
}
```

LookupAction.java

Maintenant les configurations suivantes sont nécessaires:

1) Commençons par le fichier **struts-config.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
Configuration 1.1//EN«  "http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">
<struts-config>
<form-beans>
 <form-bean name="dyna" type="org.apache.struts.validator.DynaValidatorActionForm">
 <form-property name="symbol" type="java.lang.String" />
</form-bean>
</form-beans>
<action-mappings>
<action attribute="dyna"
          input="/indexstruts.jsp"
          name="dyna"
          path="/dyna1"
          scope="request"
          type="joe.LookupAction"
      >
<forward name="succes" path="/cotation.jsp" />
</action>
```

```
</action-mappings>
<message-resources parameter="joe.ApplicationResources" />
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
 /WEB-INF/validation.xml" />
</plug-in>
</struts-config>
```

struts-config.xml

Vous devez aussi disposer d'un fichier **validation.xml** à placer dans le répertoire **/WEB-INF** de votre application web.

Voici le contenu de ce fichier

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
"-//Apache Software Foundation//DTD Commons Validator Rules Configuration 1.0//EN"
"http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">
<form-validation>
<formset>
<!--Validation mapping for action /dyna1-->
<form name="/dyna1">
<field property="symbol" depends="required, symbol, minlength">
<arg0 key="dyna.symbol"/>
<arg1 key="${var:minlength}" resource ="false"/>
   <var>
    <var-name>minlength</var-name>
    <var-value>3</var-value>
   </var>
</field>
</form>
</formset>
</form-validation>
```

validation.xml

Vous devez aussi disposer d'un fichier **validation-rules.xml,** fourni par Struts, à placer dans le répertoire **/WEB-INF** de votre application web. Ce fichier contient beaucoup de règles fournies par Struts.

NB: Sachez que ce fichier est STANDARD, mais vous pouvez le modifier et ajouter de nouvelles règles.Vous trouverez ce fichier dans **commons-validator.jar** de l'archive de Struts téléchargée.

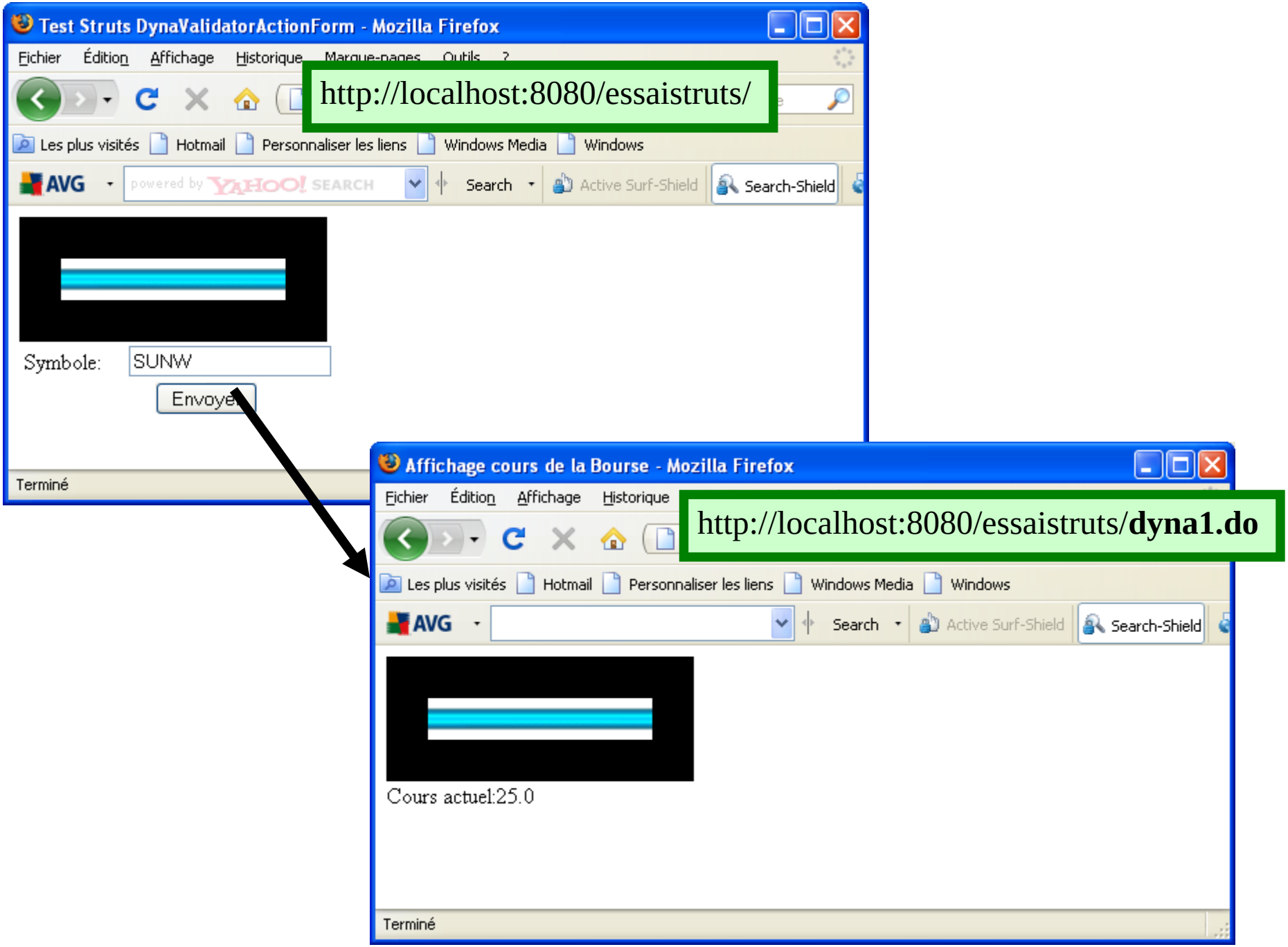
Le contenu de ce fichier n'est pas donné ici, il sera vu en cours.

Remarque: aucune configuration supplémentaire n'est nécessaire pour le fichier **web.xml**.

Et enfin voici le dernier fichier dont vous aurez besoin **ApplicationResources.properties**, à placer dans le répertoire source (dans le même emplacement que les *.java*) de votre projet web:

```
# Struts Validator Error Messages
   errors.required={0} is required.
   errors.minlength={0} can not be less than {1} characters.
# Filed names
   dyna.symbol=symbol
```

ApplicationResources.properties


Test Struts DynaValidatorActionForm - Mozilla Firefox
Fichier Édition Affichage Historique Marque-pages Outils ?
http://localhost:8080/essaistruts/
Les plus visités Hotmail Personnaliser les liens Windows Media Windows
AVG powered by YAHOO! SEARCH Search Active Surf-Shield Search-Shield
Symbole:
SUNW
Envoyer
Terminé
Affichage cours de la Bourse - Mozilla Firefox
Fichier Édition Affichage Historique
http://localhost:8080/essaistruts/dyna1.do
Les plus visités Hotmail Personnaliser les liens Windows Media Windows
AVG Search Active Surf-Shield Search-Shield
Cours actuel:25.0
Terminé

Rôle des différents fichiers de configuration précédents.

Le fichier **validation.xml** définit les règles de validation spécifiques qu'il faut appliquer aux JavaBeans (*form beans*).

<form name=**"/dyna1"**> : on indique ici la jsp qui sera validée (l'action qui sera appelée dyna1.do). Il coreresponds à la valeur de l'attribut path du pattern **<action-mappings >** du fichier ***struts-config.xml*.**

NB: On peut avoir un élément ***form*** pour chaque action créée**.**

L'élément

```
<field property ="symbol" depends="required, minlength">
        <arg0 key="dyna.symbol"/>
        <arg1 key="${var:minlength}" resource ="false“ />
        <var>
                <var-name>minlength</var-name>
                <var-value>3</var-value>
        </var>
</field>
```

L'élément ci-dessus indique comment le champ *symbol* doit être validé.

L'attribut **depends** liste les règles particulières que le champ *symbol* doit vérifié pour validation.

Ces règles doivent se trouver dans le fichier **validation-rules.xml** .

Lorsque la valeur « ***required*** » est référencée, voici le code correspondant (au code ci-dessus) dans le fichier **validation-rules.xml** :

```
<validator name="required"
classname="org.apache.struts.validator.FieldChecks"
method="validateRequired"
methodParams ="java.lang.Object,
                org.apache.commons.validator.ValidatorAction,
                org.apache.commons.validator.Field,
                org.apache.struts.action.ActionMessages,
                javax.servlet.http.HttpServletRequest"
msg="errors.required"/>
```

Note: l'attribut **msg** referre à l'attribut « **errors.required** » du fichier ***ApplicationResources.properties***, pour trouver le texte correspondant qu'il faut afficher au cas où il y aurait erreur.

Toujours dans le fichier ***validation.xml***, **arg0** remplaces **{0}** trouvé avec les attributs **errors.required** et **errors.minlength** (dans le fichier ***ApplicationResources.properties*** par le le texte de l'attribut **dyna.symbol** (ie ici *symbol*).

Le fichier **ApplicationResources.properties** contient les chaînes de caractères (String) utilisées dans votre application. Ainsi tous les messages d'erreurs par exemple peuvent être placées à ce niveau.
Ceci est capital dans la mesure où si on une centaine de pages JSPs, par exemple il n'est pas judicieux de placer ces messages directement dans les pages. Il est plutôt conseillé d'avoir un seul élément dans ce fichier qui référencent tous les messages. Et ainsi le jour où on aura besoin de modifier ces messages, au lieu de la faire dans toutes les pages, on modifie seulement un seul élément (attribut) de ce fichier.

Ce fichier peut être étendu pour supporté différentes langues (Français, Anglais, Italien, …) comme vu précédemment.

Gestion des validations de formulaires Struts

Une fois un formulaire saisi par un utilisateur, il faut qu'il soit validé un *minimum*. Cette validation peut se faire à deux endroits :

côté client : dans le butineur avec du code Javascript

côté serveur : dans le serveur d'application avec du code Java

La validation ***côté client*** permet d'éviter un aller-retour avec le serveur dans ***le cas où un champ du formulaire est invalide***. Par contre, il est impossible de s'assurer que le contrôle côté client a réellement été effectué. C'est pourquoi, il faut toujours réaliser les contrôles côté serveur.

Tout le monde s'accorde pour dire qu'il est souhaitable d'effectuer des contrôles côté client et côté serveur.

Struts propose plusieurs façon pour gérer ces contrôles :

1). la méthode ***validate ( )*** de la classe **ActionForm (**votre **JavaBean Struts)**

2) le plugin **Struts Validator**.

La première méthode est déjà vue (cf. diapo Traitement des erreurs dans Struts).
Voyons la deuxième méthode.

Struts Validator:

Struts Validator est un framework à part entière qui permet d'effectuer la validation de formulaire côté client et côté serveur par **simple configuration**. En effet, il faut configurer les règles de validations dans un fichier XML, ensuite *Struts Validator génère le code Java et le code Javascript à exécuter lors de la validation du formulaire*.

Il faut configurer les fichier **validation.xml** et **validation-rules.xml** .

Nous avons déjà vu ces deux fichiers.

ATTENTION: le fichier *validation-rules.xml* fournit par défaut par Struts ne supporte pas par défaut la validation automatique des formulaires. En effet cette validation est réalisée par la classe **org.apache.commons.validator.Validator** qui doit donc être présente dans ce fichier.

Donc il faut mettre **org.apache.commons.validator.Validator** entre les éléments *ActionMessages* et *HttpServletRequest* dans tous les blocs <validator ..../>

Exemple pour le bloc "required"

```
<validator name="required"
          classname="org.apache.struts.validator.FieldChecks"
          method="validateRequired"
          methodParams="java.lang.Object,
               org.apache.commons.validator.ValidatorAction,
               org.apache.commons.validator.Field,
               org.apache.struts.action.ActionMessages,
               org.apache.commons.validator.Validator,
               javax.servlet.http.HttpServletRequest"
            msg="errors.required"/>
```

Pour notre exemple le fichier ***validation.xml*** doit contenir ceci:

```
(…)
<form-validation>
<formset>
<!--Validation mapping for action /dyna1-->
<form name="/dyna1">
<field property="symbol" depends="required,minlength,maxlength">
<arg0 key="dyna.symbol"/>
<arg1 key="taille.symbol"/>
<arg2 name="minlength" key="${var:minlength}" resource="false"/>
        <var>
            <var-name>minlength</var-name>
            <var-value>3</var-value>
        </var>
<arg3 name="maxlength" key="${var:maxlength}" resource="false"/>
        <var>
            <var-name>maxlength</var-name>
            <var-value>5</var-value>
        </var>
</field>
</form></formset></form-validation>
```

La dernière modification à faire concerne le formulaire (pour nous la page indexstruts.jsp) qui contient le champ à vérifier.

Il faut insérer dans cette page, à l'endroit où on veut afficher le message d'erreurs de Validation, cette ligne:

```
<html:javascript formName="/dyna1" dynamicJavascript="true" staticJavascript="true" />
```

Vous devez disposer d'un fichier PROPERTIES (ex *ApplicationResources.properties*) dans lequel on met quelques messages concernant les champs à valider.

Voici un exemple de contenu de ce fichier:

```
errors.required     =   {0} est requis
errors.minlength    =   {1} doit depasser {2} caracteres
errors.maxlength    =   {1} ne doit pas  depasser {3} caracteres
taille.symbol  =  la taille du symbole
dyna.symbol  = le symbole
```

Si l'utilisateur valide le formulaire et que le champ est **vide** (non renseigné) voici le message affiché: **le symbole est requis**..

```
(….)
<struts-config>
<form-beans>
 <form-bean name="dyna" type="org.apache.struts.validator.DynaValidatorActionForm">
 <form-property name="symbol" type="java.lang.String" />
</form-bean>
</form-beans>
<action-mappings>
<action attribute="dyna"
      input="/indexstruts.jsp"
      name="dyna"
      path="/dyna1"
      scope="request"
      validate="true"
      type="eyrolles.LookupAction"  >
<forward name="succes" path="/cotation.jsp" />
<forward name="echec" path="/indexstruts.jsp" />
</action>
</action-mappings>
<message-resources parameter="ApplicationResources" />
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
/WEB-INF/validation.xml" />
</struts-config>
```

struts-config.xml

http://localhost:8080/essaistruts/dyna1.do

Test Struts DynaValidatorActionForm - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

Les plus visités Hotmail Personnaliser les liens Windows Media Windows

le symbole est requis

Symbole:

Envoyer

Client Side Validation for Address Form

The Address Form that validates the data on the client sid

JavaScript.

Test Struts DynaValidatorActionForm - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/essaistruts/dy

Les plus visités Hotmail Personnaliser les liens Windows Media Windows

AVG Search Active Surf-Shield Search

la taille du symbole ne doit pas depasser 5 caracteres

Symbole: azertyarg1

Envoyer

Client Side Validation for Address Form

The Address Form that validates the data on the client side using Stuts Validator

Terminé

Test Struts DynaValidatorActionForm - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils

http://localhost:8080/ess

Les plus visités Hotmail Personnaliser les liens Window

la taille du symbole doit depasser 3 caracteres

Symbole: su

Envoyer

Client Side Validation for Address Form

The Address Form that validates the data on the client side using Stuts Validator generated JavaScript.

Terminé

Gestion des erreurs avec **DynaValidatorActionForm**

Dans l'exemple précédent, je n'ai pas gérer les erreurs.
Je souhaite maintenant gérer deux contraintes sur le symbol.
- si l'utilisateur ne donne aucun symbole, je lui lance comme message d'erreur:
  symbol est nécessaire.
-s'il donne un symbole inexistant, le message sera: **symbol inexistant**.

Voici les quelques modifications à apporter:

On rajoute ces deux lignes dans le fichier ApplicationResources.properties

error.symbol.required = ce champ est obligatoire
error.symbol.notexist = ce champ n existe pas

L'élément **<action />** de notre fichier ***struts-config.xml*** devient

```
<action attribute="dyna"
        input="indexstruts.jsp"
        name="dyna"
        path="/dyna1"
        scope="request"
        validate="true"
        type="eyrolles.LookupAction"
       >
<forward name="succes" path="/cotation.jsp" />
<forward name="echec" path="/indexstruts.jsp" />
</action>
```

S'il n'y a pas d'erreurs

S'il y a des erreurs

Dans la page *indexstuts.jsp* où on veut faire la gestion d'erreurs on rajoute dans le *body*:

<font color = *"red"*> <html:errors/> </font>

Notre classe Action, LookupAction contient maintenant une méthode exécute comme ceci.

On importe deux classe supplémentaires

```
import org.apache.struts.action.ActionMessage;
import org.apache.struts.action.ActionMessages;
```

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
                                                    HttpServletRequest request,
HttpServletResponse response) throws IOException,ServletException{
Double cours = null;
String cible = new String ("succes");
DynaValidatorActionForm lookupForm;
ActionMessages errors = new ActionMessages ( );
String lesymbol = null;
if (form != null){
                    lookupForm = (DynaValidatorActionForm) form;
                    lesymbol = lookupForm.getString ("symbol");
   if (lesymbol.equals(""))
          errors.add ("symbol",new ActionMessage("error.symbol.required"));
   else {  cours = getCotation (lesymbol);
          if(cours == null)
                    errors.add("symbol",new ActionMessage("error.symbol.notexist"));
        }
   this.saveErrors(request, errors);
   }
if(!errors.isEmpty())  cible = new String ("echec");
else request.setAttribute ("COURS",cours);
return mapping.findForward (cible);
}
```

Test Struts DynaValidatorActionForm - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/essaistruts/dyna1.do

Les plus visités Hotmail Personnaliser les liens Windows Media Windows

AVG Search Active Surf-Shield Search-Shield

ce champ est obligatoire

Symbole:

Envoyer

Terminé

Test Struts DynaValidatorActionForm - Mozilla Firefox

Fichier Édition Affichage Historique Marque-pages Outils ?

http://localhost:8080/essaistruts/dyna1.do

Les plus visités Hotmail Personnaliser les liens Windows Media Windows

AVG Search Active Surf-Shield Search-Shield

ce champ n existe pas

Symbole: franc

Envoyer

Terminé

Exercice

Reprendre l’exemple du modèle MVC (diapo 180), en utilisant Struts.

Classe ActionServlet

Cette classe est le cœur des applications Struts. C'est le composant principal du contrôleur qui gère les requêtes des clients et détermine quelle classe ***Action*** doit traiter chacune d'elles.

Elle agit comme une ***factory*** pour la classe *Action*, c'est-à-dire qu'elle produit des instances spécifiques de cette classe en fonction des requêtes qu'elle reçoit.

La classe ***ActionServlet*** dérive de ***HttpServlet*** et implémente donc les méthodes *init ( ), doGet ( ) doPost ( ) et destroy ( ).*

Il faut savoir que son comportement spécifique commence avec la méthode ***process ( )*** *qui effectue le traitement des requêtes. Sa signature est:*

protected void process (HttpServletRequest request, HttpServletResponse response)

Lorsqu'une *ActionServlet* reçoit une requête, elle effectue différentes opérations que nous énumérons ci-dessous:

1. La méthode **doGet** ( ) ou **doPost** ( ) reçoit la requête et appelle la méthode **process** ( ).
2. La méthode **process** ( ) obtient le **RequestProcessor** courant et invoque sa méthode **process** ( ). C'est en fait la classe *RequestProcessor* qui contient la logique mise en œuvre par le controleur pour traiter chaque requête.
3. La méthode *process ( )* de *RequestProcessor* effectue le traitement. Elle recherche dans le fichier *struts-config.xml* l'élément <action> correspondant au chemin d'accès de la requête en comparant la valeur de l'attribut **action** de l'élément **<*html:form*** > de la page avec celle de l'attribut **path** des éléments <action> dans le fichier *struts-config.xml* .
4. Lorsque la méthode *process* ( ) trouve une correspondance, elle recherche un élément <***form-bean*** > dont l'attribut **name** correspond à celui de l'élément **action**.
5. La méthode *process ( )* est alors en mesure de connaître le nom qualifié de l' *ActionForm* indiqué par l'attribut **type** de l'élément <*form-bean* >. Elle peut donc obtenir une instance de cette classe a partir du pool et initialise ses variables membres en fonction des valeurs des paramètres de la requête.
6. Une fois l'ActionForm initialisé, *process* ( ) appelle la méthode ***validate*** ( ) de celui-ci qui teste la validité des données reçues.
7. A ce stade la méthode process ( ) est en possession de toutes les informations nécessaires et est en mesure d'effectuer le traitement proprement dit de la requête. Elle obtient du pool une instance *d'Action* correspondant à la classe indiquée par l'attribut type de l'élément <action> et appelle sa méthode ***execute*** ( )

8. Lorsque la méthode ***execute( )*** se termine elle retourne un objet de type *ActionForward* qui est utilisé pour déterminé la cible de la transmission en cours. La méthode ***process ( )*** reprend le contrôle des opérations et la requête est transmise à la cible.
9. ***l'ActionServlet*** a maintenant terminé le traitement et est prête à prendre en charge une nouvelle requête.

Pool de connexions JDBC dans Struts

Nous montrons ici l'usage des ***DataSource*** (sources de données) dans une application Struts.

On utilise l'interface ***javax.sql.DataSource*** . Vous aurez besoin de placer **jdbc2_0-stdext.jar** (qui est l'extension standard Java DataBase Connectivity 2.0) dans le classpath de votre application.
NB: si vous avez plusieurs applications, il faudra le placer dans le *classpath* de chacune d'elles. mais il est préférable de le placer une et une seule fois dans le répertoire **<CATALINA_HOME>/shared/lib**, ce qui vous évitera de placer une copie de ce fichier dans chaque classpath de chaque application.

Une DataSource est un objet décrit par l'extension JDBC 2.0 et utilisé pour représenter de manière générique un système de gestion de base de données.
Les sources de données décrites par JDBC 2.0 sont définies par l'interface javax.sql.DataSource.
Il s'agit d'un composant de type ***factory*** contenant des objets JDBC de type **Connection**.

Comme c'est une interface, vous devez fournit votre propre implémentation ou utiliser simplement les implémentations proposées par JDBC.
JDBC 2.0 propose trois implémentations qui sont **Basic**, **Pooled** et **Distributed**.

Basic: cette implémentation génère des objets **Connection** standards non distribués et n'utilisent pas de pool.

Pooled: génère une collection d'objets JDBC **Connection** gérés par un pool. Cette implémentation permet de réutiliser plusieurs fois la même connection.

Distributed: génère des objets **Connection** pouvant être utilisés par des transactions distribuées. de cette façon, une même transaction peut avoir accès à plusieurs serveurs de base de données.

Struts utilise une *DataSource* pour gérer les connexions avec une base de données. Cette ***DataSource*** comporte toutes les fonctionnalités nécessaires pour gérer un pool de connexions.

Nous donnons un exemple ci-après:

Dans l'exemple d'application Struts que nous avons donné, le symbole « SUNW » était saisi sur le formulaire et la valeur associée était disponible aussi dans l'application.
Maintenant, nous créons une base de données MySQL contenant tous les symboles avec leur valeur associée.
La base s'appelle **cotation** et contient une table nommée **cota**.

```
create table cota {
        symbol varchar (15) not null primary key,
        cours double not null
};
```

Comment utiliser une DataSource ?

Après avoir créer votre base de données, il faut intégrer la ***DataSource*** à Struts.
Vous pouvez le faire en utiliser les fonctionnalités de la classe ***ActionServlet***.

Il faut insérer l'élément <***data-sources***> suivant au fichier **struts-config.xml** .
NB: cet élément doit être situé avant les éléments <*form-beans*> et <*action-mapping*> .

```
<data-sources>
 <data-source>
   <set-property property="driverClass" value="org.gjt.mm.mysql.Driver" />
   <set-property property="url" value="jdbc:mysql://localhost/cotation" />
   <set-property property="maxCount" value="5" />
   <set-property property="minCount" value="1" />
   <set-property property="user" value="joe" />
<set-property property="password" value="ndong" />
</data-source>
</data-sources>
```

Framework SPRING

Spring est un framework open source rendant simple l'utilisation de JEE.
Spring est né du constat que l'utilisation des EJBs (Enterprise JavaBean) est très complexe et n'est adaptée que pour des projets de très grande envergure.

Il se base sur le concept de conteneur léger et s'appuie sur des concepts modernes :

- inversion de contrôle
- programmation orientée aspect (POA)

Il peut s'intégrer facilement avec d'autres frameworks tels que Struts, Hibernate, etc.

Spring propose donc une intégration des frameworks tiers les plus répandus.
A ce point Spring propose une architecture applicative générique capable de s'adapter aux choix technologiques.

Conteneur léger

La notion de conteneur EJB est une technologie lourde à mettre en œuvre, inadaptée pour les objets de faible granularité, intrusive dans le code et en partie dépendante du serveur d'applications, ne serait-ce qu'en terme de configuration.

Un conteneur léger adopte une autre démarche: fournir une infrastructure de gestion de l'ensemble des composants de l'application.
Ceci s'effectue:
- au travers de mécanismes de configuration comme les fichiers XML pour initialiser les composants
- au travers de la gestion du cycle de vie des composants
- au travers de la gestion des dépendances entre les composants

Les conteneurs légers sont indépendants des technologies JEE et peuvent fonctionner sur tout type de serveur d'application, voire sans eux en utilisant une simple JVM.
Ils rendent les applications qui les utilisent à la fois plus flexibles et mieux testables, car le couplage entre les composants est géré par le conteneur et non plus directement dans le code source de l'application.

Le support de la POA

La POA est un nouveau paradigme (ensemble de principes qui structurent la manière de modéliser les applications et en conséquence la façon de les développer) qui étend et complète les autres paradigmes de programmation structurale et POO.

Elle offre des mécanismes afin de mieux séparer les différentes préoccupations d'une application en introduisant une nouvelle dimension de modularisation : ***l'aspect***.

Elle se concentre sur les éléments ***transversaux***, c'est-à-dire ceux qui se trouvent **dupliqués** ou *utilisés dans un grand nombre d'entités comme les classes ou les méthodes*, sans pouvoir être centralisés au sein d'une entité unique avec les concepts classiques.
Ainsi grâce à la ***notion d'aspects*** qui capturent en leur sein les ***préoccupations transversales***, la séparation des préoccupations est nettement améliorée.

Modèles de conception (design pattern)

Une modélisation appropriée des applications nécessite souvent de passer par des moyens ***génériques*** (**modèles**) pour bien séparer les ***notions fonctionnelles*** manipulées par ces applications, leur ***gestion*** et leur ***persistance***.

Les fondements d'une bonne modélisation s'appuient sur deux piliers fondamentaux: *Séparation claire entre les notions fonctionnelles manipulées par l'application*, leur *gestion* et leur *persistance*. L'idée est d'améliorer la réutilisation et l'évolution de services autour des notions fonctionnelles. Il s'agit d'une architecture en couches:

- ***une couche modèle contenant les notions fonctionnelles***
- ***une couche service qui peut gérer les notions techniques***
- ***une couche pour la persistance des données.***

***Séparation claire entre les <u>interfaces</u> et les <u>implémentations</u>* pour rendre transparentes les modifications des implémentations qui n'impactent pas les interfaces.**

A partir de ces deux choix, nous pouvons choisir les modèles de conception suivants:

Modèles de conception (design pattern)

La **fabrique** (***factory***): ce modèle de conception permet d'associer une ***implémentation*** à une ***interface***. Si vous désirez rendre une application indépendante des implémentations, il est nécessaire de faire appel à une classe spécifique la fabrique qui saura instancier l'implémentation correspondant à une interface donnée.

Le **singleton**: ce modèle de conception définit des classes n'ayant ***qu'une seule instance*** pour l'ensemble de l'application.

L'objet d'accès aux données (**DAO**): ce modèle isole la problématique de l'accès aux données dans des classes prévues exclusivement à cet effet. Cela permet d'éviter la prolifération du code d'accès aux données au sein de l'application en le concentrant dans quelques classes spécialisées, plus faciles à gérer.

Inversion de contrôle
IoC (Inversion of Control)

Il s'agit du contrôle du flot d'exécution d'une application. Ce concept propose une gestion des dépendances entre objets et leur instanciation, notamment dans le cadre d'une dissociation entre des interfaces et leur implémentation.

Dans les applications complexes, souvent beaucoup d'objets fabriques et singletons sont nécessaires. Et l'application devrait être en mesure de maîtriser la création des objets dont elle a besoin et la gestion de leurs dépendances, ce qui assez fastidieux.

Un framework de création doit pouvoir prendre en charge cette problématique de manière générique. C'est en ce moment que l'inversion de contrôle entre ne jeu.

Le résultat attendu: disparution pure et simple des fabriques et de l'implémentation des Singletons.

Grâce au conteneur les mécanismes (explicites) de fabrication des objets et des dépendances devient générique.

Pour réaliser l'inversion de contrôle, Spring utilise le concept ***d'injection de dépendances*** pour rendre cette dernière transparente**.**

L'injection peut se faire:

- *par le contructeur,*
- *par les modifieurs ou mutateurs (setter)*
- *par des interfaces.*

Injection par modifieurs (**setter injection**): la propriété (**champ**) à modifier doit appartenir à un JavaBean

```
public class namebean
{
    String      name;

    public void setName (String a)
    {
      name = a;
    }
    public String getName ( )
    {
      return name;
    }
}
```

```
<bean id="bean1"   class="namebean">

  <property       name="name" >
    <value>ndong</value>
  </property>

</bean>
```

Injection par constructeur :

```
public class namebean
{
    String name;
    public namebean (String a)
    {
      name = a;
    }
}
```

```
<bean id="bean1"  class="namebean">
   <constructor-arg>
     <value> ndong </value>
   </constructor-arg>

</bean>
```

Exemple avec Spring

Avant de donner cet exemple, il faut installer le minimum requis pour pouvoir utiliser Spring
Vous pouvez téléchargez le framework spring contenant le fichier zip
spring-framework-1.2.9-with-dependencies.

Lien de téléchargement: **http://www.springframework.org**

Décompressez le fichier dans un répertoire de votre choix et copier le fichier ***spring.jar*** (trouvez ce fichier dans le répertoire dist de l'archive décompressé), dans le répertoire **lib** de votre application web.

Exemple avec Spring

Il s'agit dans cet exemple de soumettre un formulaire ***html*** contenant un paramètre qui est le nom d'une personne à une servlet qui répond par un message de bienvenue. Spring nous permet de créer dynamiquement une instance permettant à la servlet de fournir un message.

On crée une application web sous **Tomcat** appelée **webspring** .
Voici le code source du formulaire:

```
<html>
    <body>
        <form    method="post"    action = "/webspring/springserv">
            <input type = "text" name = "nompersonne">
            <input type = "submit">
        </form>
</body>
</html>
```

servletclient.jsp

Exemple avec Spring

Le descripteur de déploiement doit contenir une entrée indiquant notre servlet (*servletclient.java*).

```
<servlet>
        <servlet-name>springtest</servlet-name>
        <servlet-class>servletclient</servlet-class>
        <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
        <servlet-name>springtest</servlet-name>
        <url-pattern>/springserv</url-pattern>
</servlet-mapping>
```

Voici les classes java et fichier .**xml** dont nous avons besoin:

Exemple avec Spring

```
package joe;
public interface hello
{
    public String sayhello (String message);
}
```

hello.java

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id ="hello"    class="joe.helloimpl">
  <property name="greeting">
     <value>Good Morning!...</value>
  </property>
  </bean>
</beans>
```

hello.xml

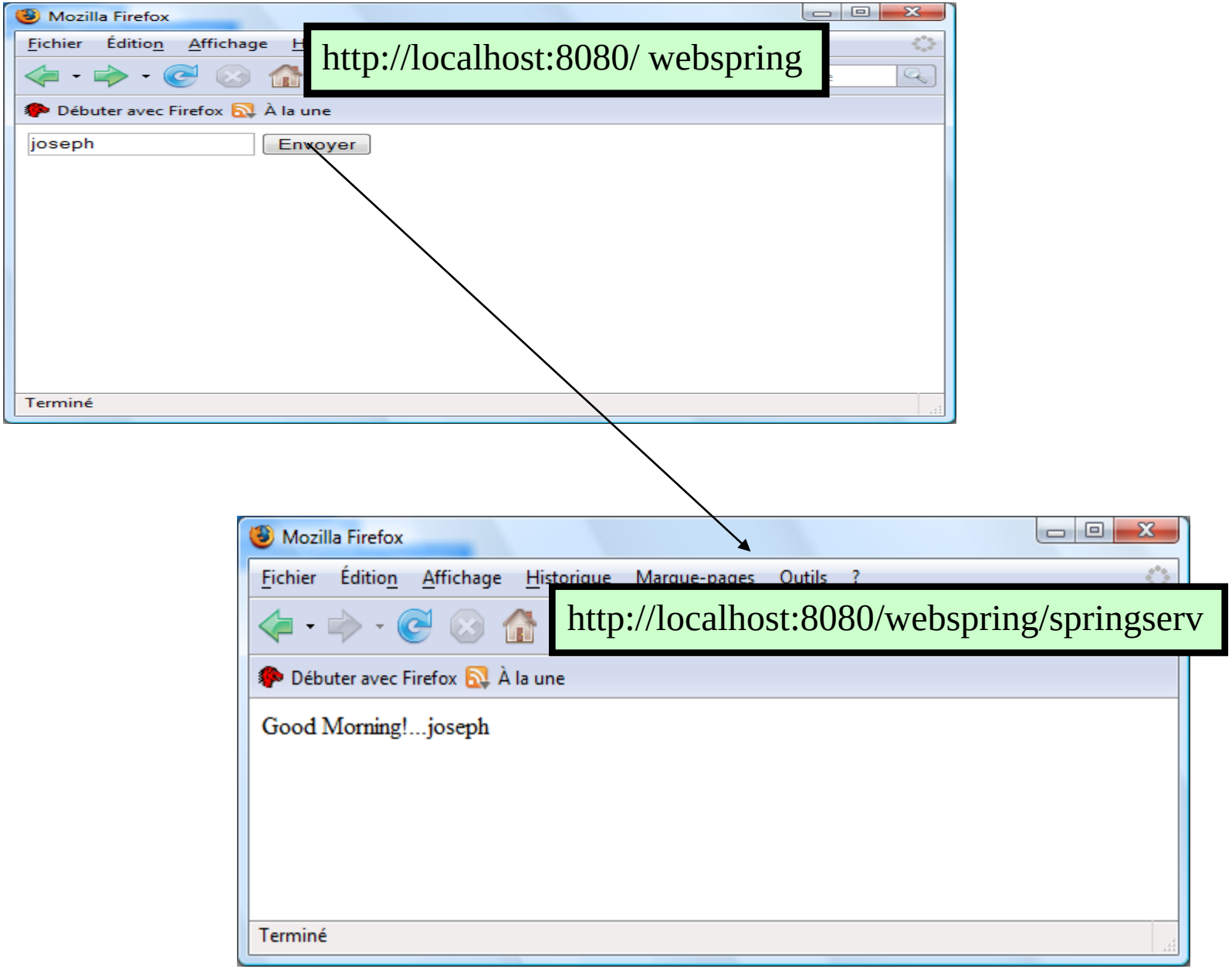
Il faut placer ce fichier dans le classpath de l'application.

```
package joe;

public class helloimpl implements hello
{
    private String greeting;

    public helloimpl ( )    {     }
    public helloimpl (String a)
    {
       greeting = a;
    }
    public String sayhello (String s)
    {
        return greeting + s;
    }
    public void setGreeting (String a)
    {
        greeting= a;
    }

}
```

helloimpl.java

```
package joe;
import java.io.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
import org.springframework.core.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class servletclient extends HttpServlet{
   public void doPost (HttpServletRequest req,HttpServletResponse resp)
            throws ServletException,IOException
   {
      resp.setContentType ("text/html");
      PrintWriter out = resp.getWriter ( );
      String a = req.getParameter ("nompersonne");
      try       {
                  Resource  res = new ClassPathResource ("/joe/hello.xml");
                  BeanFactory  factory = new XmlBeanFactory(res);
                  hello bean1 = (hello) factory.getBean ("hello");
                  String s = bean1.sayhello (a);
                 out.println (s);
      } catch (Exception e1){ }
   }
}
```

Servletclient.java


Mozilla Firefox
Fichier
Édition
Affichage
http://localhost:8080/ webspring
Débuter avec Firefox
À la une
joseph
Envoyer
Terminé
Mozilla Firefox
Fichier
Édition
Affichage
Historique
Marque-pages
Outils
?
http://localhost:8080/webspring/springserv
Débuter avec Firefox
À la une
Good Morning!...joseph
Terminé

Les apports de Spring (1/2)

Spring est une boîte à outils présentant beaucoup de modules utilisables au choix.

Spring est disponible sous deux formes, celle d'un jar unique et celle de plusieurs fichiers jar permettant de ne rajouter au projet que la partie que l'on souhaite utiliser
(i.e. Spring Core, Spring Remoting, etc...).

Ce framework fournit des services de *type fonctionnel* comme par exemple les transactions mais est également utile d'un point de *vue conceptuel* en améliorant la qualité du design.
Cela par l'utilisation *d'interface*.

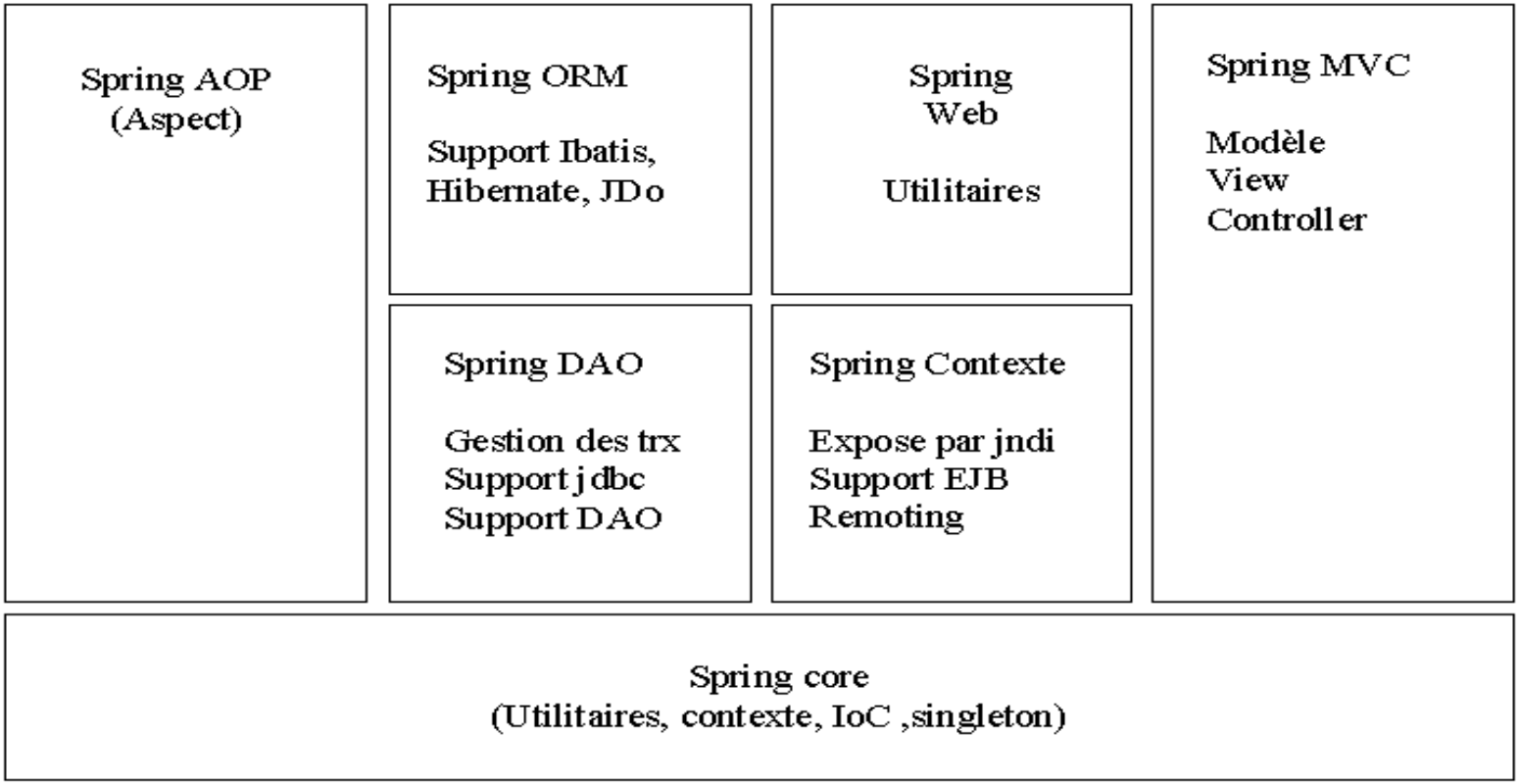
Voici une liste non exhaustive des services qu'offre Spring:

Les apports de Spring (2/2)

1 Découplage des composants. Moins d'interdépendances entre les différents modules.
2 Rendre plus aisés les tests des applications complexes c'est-à-dire des applications multicouches.
3 Diminuer la quantité de code par l'intégration de frameworks tiers directement dans Spring.
4 Permettre de mettre en oeuvre facilement la programmation orientée aspect.
5 Un système de transactions au niveau métier qui permet par exemple de faire du "two-phases-commit".
6 Un mécanisme de sécurité.
7 Pas de dépendances dans le code à l'api Spring lors l'utilisation de l'injection. Ce qui permet de remplacer une couche sans impacter les autres.
8 Une implémentation du design pattern MVC
9 Un support du protocole RMI. Tant au niveau serveur qu'au niveau du client.
10 Déployer et consommer des web-services très facilement.
11 Echanger des objets par le protocole http.

Architecture de Spring (1/2)

Ci-dessous un schéma représentant l'organisation des modules et fonctionnalités à l'intérieur du framework.



Spring est organisé en différents modules, reposant tous sur le module **Spring Core**;

Architecture de Spring (2/2)

1 **Spring Core :** implémente le concept *d'inversion de contrôle* (*injection de dépendance*). Il est également responsable de la gestion et de la configuration du conteneur.

2 **Spring Context :** Ce module étend Spring Core. Il fournit une sorte de base de données d'objets, permet de charger des ressources (telles que des fichiers de configuration) ou encore la propagation d'évènements et la création de contexte comme par exemple le support de Spring dans un conteneur de Servlet.

3 **Spring AOP :** Permet d'intégrer de la programmation orientée aspect.

4 **Spring DAO :** Ce module permet d'abstraire les accès à la base de données, d'éliminer le code redondant et également d'abstraire les messages d'erreur spécifiques à chaque vendeur. Il fournit en outre une gestion des transactions.

5 **Spring ORM :**intégration des frameworks de mapping Object/Relationnel tel que Hibernate, JDO ou iBatis avec Spring. La quantité de code économisé par ce package peut être très impressionnante (ouverture, fermeture de session, gestion des erreurs)

6 **Spring Web :** Ensemble d'utilitaires pour les applications web. Par exemple une servlet qui démarre le contexte (le conteneur) au démarrage d'une application web. Permet d'utiliser des requêtes http de type multipart. C'est ici que se fait l'intégration avec le framework Struts.

7 **Spring Web MVC :** Implémentation du modèle MVC.

Quand et comment utiliser Spring ?

Voici, plusieurs cas d'utilisation du framework Spring. Vous pouvez toujours trouvé l'architecture adéquate à mettre en œuvre pour l'application à implanter.

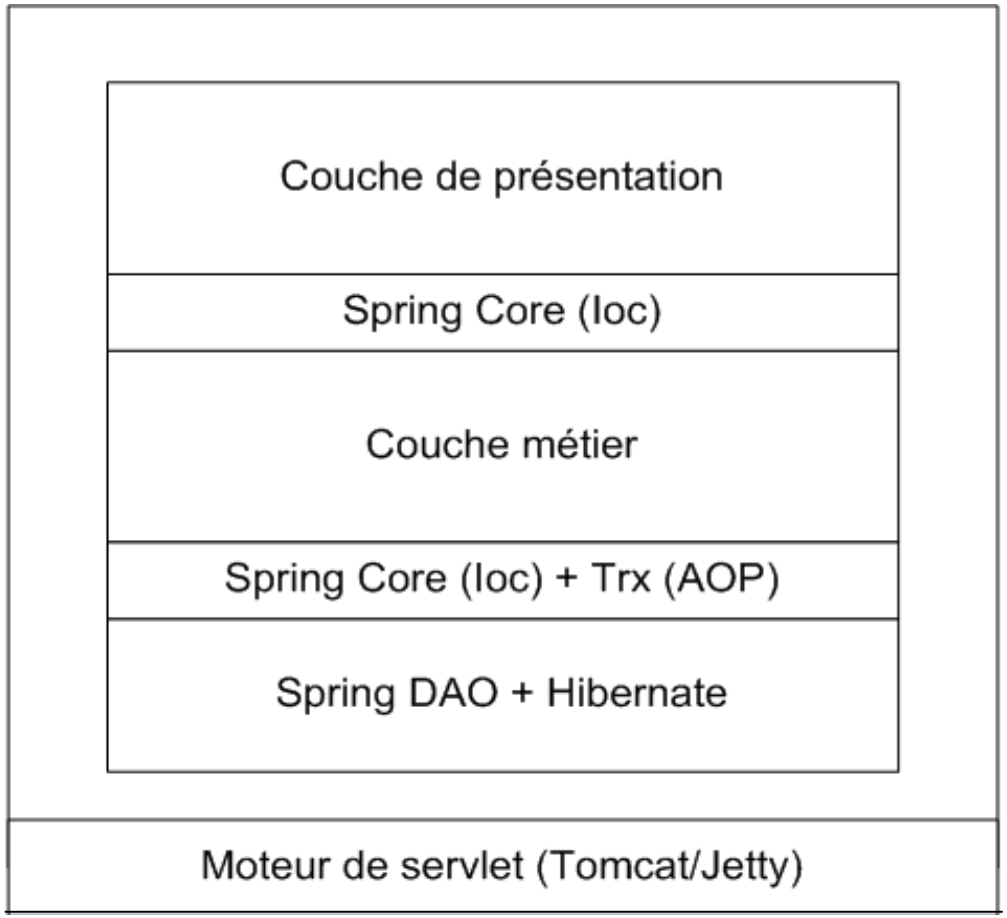
Une grande majorité des applications d'entreprise à l'heure actuelle sont basées sur une architecture multicouche (souvent trois).

- une couche de présentation (soit une application Internet ou un client lourd),
- une couche métier (POJO ou EJB)
- une couche d'accès aux données.

Voici quelques exemples d'architectures :

Exemples d'architectures

1 - Application Web Inter/Intranet



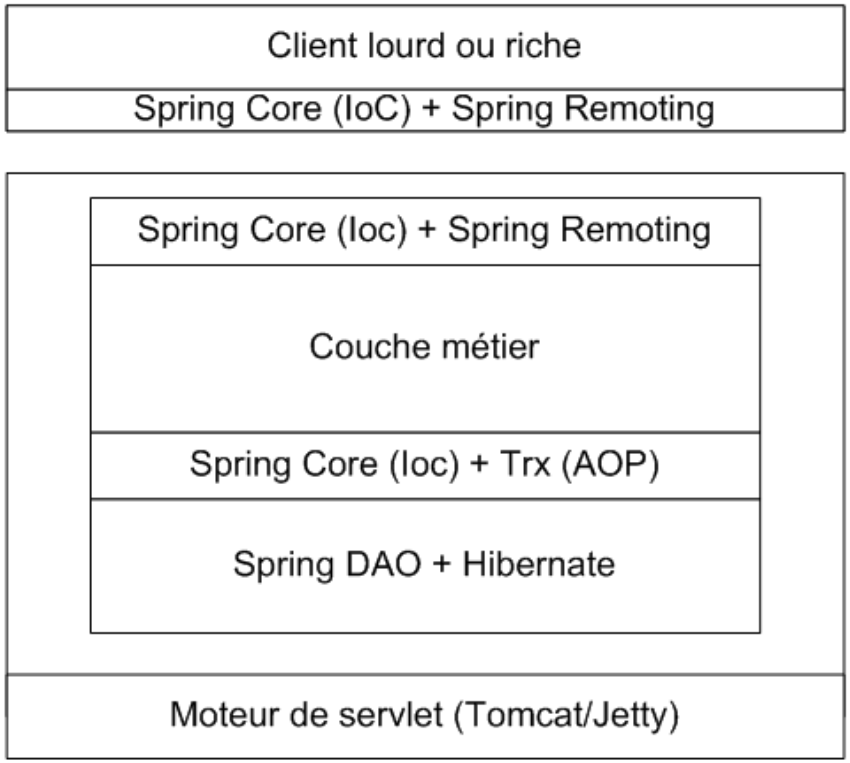
Dans cet exemple on peut recenser les parties suivantes:

1 Une couche de présentation, qui peut être ***Spring MVC***, ***Struts*** ou encore ***WebWock***.
2 Une couche métier qui en l'occurrence est constituée de POJO (Plain old Java object)
3 Une couche d'accès aux données qui peut être JDBC, Hibernate, JDO ou autres.

Entre chaque couche, Spring fait le lien et apporte des fonctionnalités supplémentaires,
par exemple :

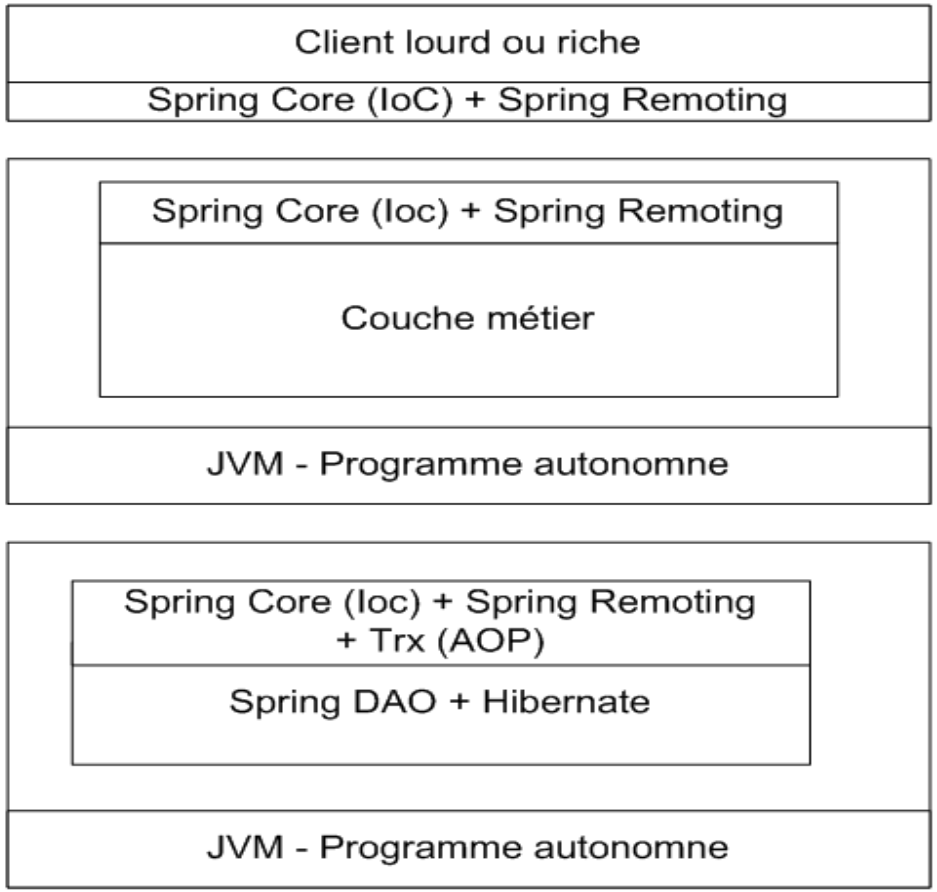
1 sur la couche métier, un mécanisme déclaratif de transactions du type EJB
2 ou encore l'utilisation de l‘ AOP pour activer des *logs* ou enregistrer les temps de traitements.
3 Un mécanisme de *templates* pour réduire la quantité de code.

2 - Client lourd ou riche



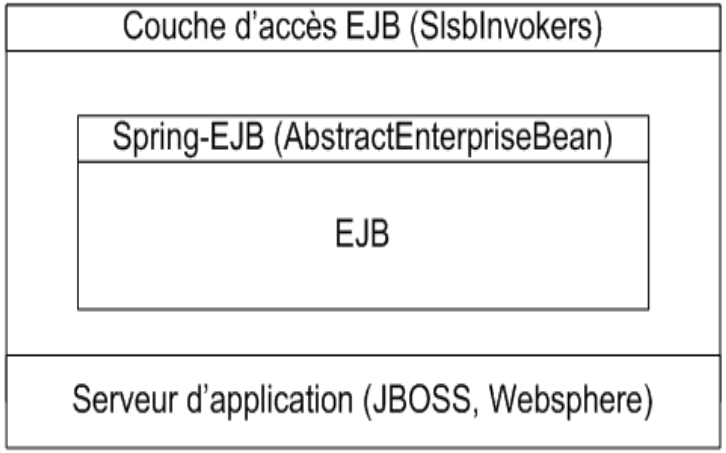
Ce cas de figure reste relativement proche du précédent, c'est là toute la puissance de Spring et de la programmation par contrat, il suffit de rajouter un ou deux paramètres et voila nos classes métier précédentes exposées par Web services ou RMI.
Ceci fait, Spring fournit également des outils pour faciliter leurs utilisations sur le client.

3 - Architecture répartie



Ce modèle permet de réaliser une répartition des modules sur plusieurs JVM et donc plusieurs serveurs.

4 - EJB gérés par Spring



La programmation par contrat

Spring oblige à programmer par contrat (***interface***) et rend la chose praticable.
Ce qui présente pour l'architecte un atout non négligeable.
En utilisant des interfaces, Spring permet de changer très facilement ***l'implémentation*** d'une interface.
Spring propose simplement une stratégie de *développement horizontal* (en couches pour séparer les différentes parties de l'application en fonction des compétences de chaque développeur) au lieu d'adopter une position de *développement vertical* (où on cherche plusieurs compétences pour le même développeur, ce qui n'est pas toujours évident).

Développement vertical

Dev. 1	Dev. 2	Dev. 3
JSP	JSP	JSP
Actions	Actions	Actions
Métier	Métier	Métier
Accès aux données	Accès aux données	Accès aux données

Développement horizontal

Dev. 1	JSP	JSP	JSP
Dev. 2	Actions	Actions	Actions
Dev. 3	Métier	Métier	Métier
Dev. 4	Accès aux données	Accès aux données	Accès aux 237 données

Le modèle de conception " fabrique " (*factory*)

C'est grâce à ce patron de conception que Spring peut produire des objets respectant un contrat mais indépendants de leur implémentation.

En réalité ce modèle est basé sur la ***notion d'interface et donc de contrat***, l'idée est simplement d'avoir un point d'entrée unique qui permet de produire des instances d'objets.

La *fabrique de bean* (*Bean Factory*) de Spring est l'interface de base permettant aux applications reposant sur le conteneur léger de Spring d'accéder à ce dernier. Elle définit les fonctionnalités minimales dont dispose l'application pour dialoguer avec le conteneur.

La notion de fabrique de bean, qui offre un moyen d'accès au conteneur léger, est manipulable par l'application au travers deux interfaces :
- BeanFactory, qui définit l'accès aux Beans gérés par le conteneur
- BeanDefinitionRegistry, qui formalise la définition des Beans que le conteneur léger doit gérer.

Ces interfaces possèdent plusieurs implémentations fournies en standard par Spring, mais elles peuvent être implémentées dans un cadre propre à un projet.
Elles se trouvent dans le package: **org.springframework.beans.factory** .

Le modèle de conception singleton

Il s'agit certainement du patron de conception le plus connu.
Il assure qu'il n'y aura toujours qu'une *seule instance* d'une classe donnée qui sera créée dans la machine virtuelle.
Les objectifs sont simples:
1 Éviter le temps d'instanciation de la classe.
2 Éviter la consommation de mémoire inutile.

Ce modèle impose cependant une contrainte d'importance, la classe qui fournit le service ne doit pas avoir de notion de session. C'est à dire qu'importe le module appelant, le service réagira toujours de la même façon à paramètre équivalent, il n'a pas de notion d'historique (ou de session).
Le POJO qui implémentera le service ne doit pas stocker des informations au niveau de l'objet lui-même. Plus simplement, il ne faut pas modifier les variables membres au sein d'une opération (une méthode du service).

Pour la mettre en œuvre :

L'objectif est de garantir *l'unicité de l'instance*, pour cela il faut interdire la création de toute instance en dehors du contrôle de la classe. Pour ce faire, nous rendons le ***constructeur privé*** et mettons à disposition une ***méthode qui retourne une instance de la classe***.

Le modèle de conception singleton

Singleton
+ **instance** :: Singleton + **getInstance ( )**:: Singleton

```
public class Singleton {
public static Singleton getInstance ( ) {
                                        if (instance == null) {
                                            instance = new Singleton ( );
                                        }
                    return instance;
        }
private Singleton ( ) {    } // constructeur privé
private static Singleton instance;
}
```

Remarque importante:

En fait il manque quelque chose pour que cette classe soit une implémentation correcte du singleton.
Il s'agit du problème du ***multithreading***, en effet si plusieurs threads accèdent de façon simultanée à la méthode **getInstance ( )** il se peut que les deux threads détectent en même temps que le membre est **null** ce qui conduira à un problème.
Pour cela il faudrait déclarer la méthode **synchronized** ou tout du moins la portion de code qui manipule l'instance.

C'est justement un des avantages de Spring, vous n'avez plus à vous soucier de cela, Spring le fait pour vous et il le fait très bien.

Framework Hibernate

Hibernate est un framework populaire open source présentant un ensemble d'outils permettant de gérer le mappage objet/relationnel pour la platform Java.
Il permet de gérer donc la persistance des objets Java en permettant au développeur de lier les objets dans des bases de données relationnelles. Il permet par ailleurs de gérer les requêtes vers le système de stockage des données.
Désormais hibernate fournit un outil pour les requêtes SQL, il s'agit de HQL, acronyme de ***Hybernate Query Language***.

Dans ce cours, nous utilisons la version 3.0 de cet outil.

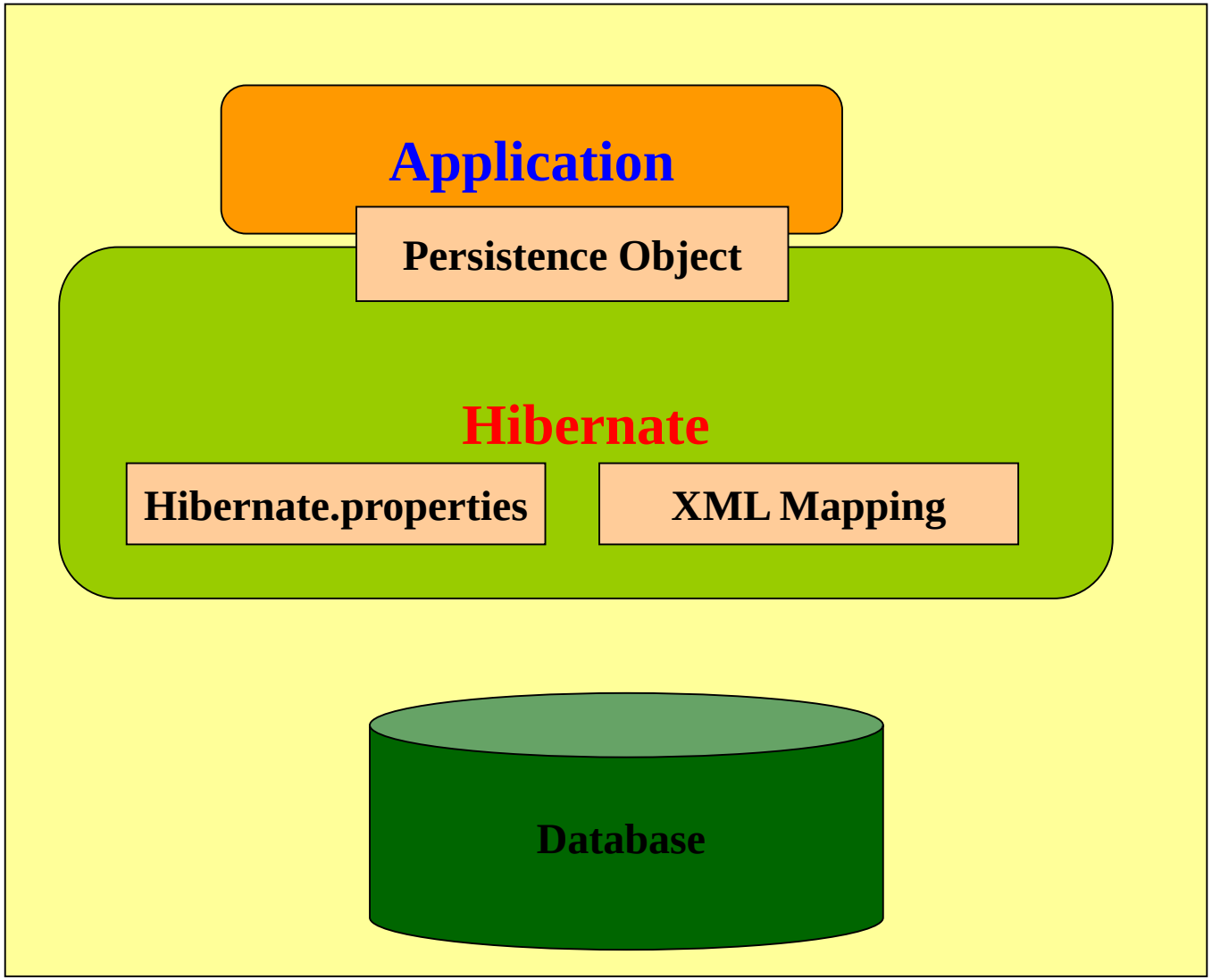
Cet outil est sous licence LGPL et vous pouvez télécharger tous les packages à l'adresse: http://www.hibernate.org/Download/DownloadOverview .

Tutoriel sur Hibernate

Hibernate mappe les classes Java (donc les objets Java) dans les tables de la base de données. Il permet de créer les requêtes sur les données de la base avec un gain de temps considérable.

Hibernate 3.0 fournit 3 principales options:
- Hibernate Query Language
- Hibernate Criteria Query API
- native SQL

Architecture d' Hibernate



Le diagramme ci-dessus montre que Hibernate utilise la base de données et la configuration des données pour fournir les services de persistance (persistance des objets) à votre application.

Pour utilise Hibernate, il est nécessaire de créer des classes Java qui vont représenter les différentes tables de la base de données et de mapper ainsi les instances de la classe avec les colonnes de tables dans la base.

Ainsi Hibernate peut être utilisé pour générer des requêtes de type ***select, insert, update*** et ***delete*** sur les enregistrements d'une table donnée.
Hibernate crée automatiquement les requêtes pour produire ces opérations.

L'architecture d' hibernate comprend trois composantes principales:

1) **Connection management** (gestionnaire de connexions)
   cet outil fournit un gestionnaire efficient des connexions sur la base de données.
2) **Transaction management** (gestionnaire de transactions):
   cet outil permet au développeur d'exécuter plusieurs requêtes sur la base à un instant donné.
3) **Object relationnel mapping** (mappage Objet/relationnel):
   les données du modèle objet (classe Java) sont représentées en un modèle relationnel (base de données).
   Cet outil permet les opérations ***select, update insert delete*** sur les tables de la base.
   Lorsqu'un objet Java est passé à une méthode **Session.save ( ),** Hibernate lit l'état des variables (champs) de cet objet et effectue les requêtes nécessaires.

Notes importantes.

Hibernate est un très bon outil du point de vue mappage objet/relationnel mais en terme de ***gestionnaire*** de ***connexions*** et de ***transactions***, il souffre de quelques insuffisances. De ce fait souvent pour ces deux dernières fonctions, il est couplé avec d'autres outils qui servent de gestionnaire de connexions et de transactions. Un exemple est **apache DBCP** qui gère les *pools de connexions*, couplé avec hibernate.

Exemple de mise en œuvre

Nous allons à présent donner un exemple de mise en œuvre d'une application avec l'outil Hibernate.
Voici les différents éléments dont nous aurons besoin:

-Une base de données MYSQL **hiberbase** contenant la table
CONTACT (ID, FIRSTNAME, LASTNAME, EMAIL) que vous créerez.
- une classe persistante **Contact** à mapper dans la base
- une classe **HiberTest** qui permet d'enregistrer des enregistrements dans la table **CONTACT**.
- un fichier **contact.hbm.xml**
- un fichier **hibernate.cfg.xml**

Notes:

Je suppose que vous avez téléchargé Hybernate et que MySql aussi est bien installé.

Fichier de mapping : contact.hbm.xml .

Hibernate a besoin de savoir comment charger et stocker des objets d'une classe persistante. C'est là qu'intervient le fichier de **mapping Hibernate**. Le fichier de mapping indique à Hibernate à quelle table dans la base de données il doit accéder, et quelles colonnes de cette table il devra utiliser. Ce fichier ressemble à ceci:

```
<?xml version="1.0"?> <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
 Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

    [...]

</hibernate-mapping>
```

Remarque :
Le fichier DTD est inclus dans **hibernate3.jar** ainsi que dans le répertoire **src** de la distribution Hibernate (**hibernate-core.jar)** .

1) Créer un projet de type **Java Project** (ex. testhiber)
2) Lors de la création, définir **\testhiber\src** comme répertoire des sources java et **\testhiber\bin** pour les fichiers compilés.
3) Ajoutez dans ce projet la classe **Contact** suivante qui est un *JavaBean*.

```
package essai;
public class Contact implements java.io.Serializable{
        private String firstName;
          private String lastName;
          private String email;
          private long id;
          public String getEmail ( ) {     return email;       }
          public String getFirstName ( ) {   return firstName;  }
          public String getLastName ( ) {  return lastName;            }
          public void setEmail (Sring string) {  email = string;          }
          public void setFirstName (String string) {    firstName = string;         }
          public void setLastName (String string) {  lastName = string;           }
          public long getId ( ) {    return id;       }
          public void setId (long l) {    id = l;      }
        }
```

4) Ajouter dans le répertoire **src** les fichiers **contact.hbm.xml** et **hibernate.cfg.xml**
voici le code de ces deux fichiers:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="essai.Contact" table="contact">
            <id name="id" type="long" column="ID" >
            <generator class="assigned"/>
           </id>
           <property name="firstName">
                    <column name="FIRSTNAME" />
           </property>
           <property name="lastName">
                    <column name="LASTNAME"/>
           </property>
           <property name="email">
                    <column name="EMAIL"/>
           </property>
  </class>
</hibernate-mapping>
```

Si le type est absent, Hybernate peut le trouver par réflexion.

contact.hbm.xml

Chaque instance de la classe Contact sera une ligne (un tuple) dans la table contact.

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost/hiberbase
    </property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password"></property>
    <property name="hibernate.connection.pool_size">10</property>
    <property name="show_sql">true</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.hbm2ddl.auto">update</property>
    <!-- Mapping files -->
    <mapping resource="contact.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

Mise en œuvre : déroulement

La première chose à savoir c'est que l'unité de travail dans Hibernate, ce qui nous sert à accéder à la base de données, c'est la **« Session »**. Une ***Session*** dans Hibernate et le ***« Persistence Manager »,*** nous l'utilisons pour mettre et récupérer des données depuis la base. On dit aussi qu'une *session factory* est une fabrique d'instances de Session très importante pour Hibernate. Elle implémente un Design Pattern, qui assure que, pendant l'exécution, chaque Thread utilise la même instance de la session

Pour récupérer une Session, nous le faisons grâce à la SessionFactory comme cela :

```
SessionFactory sessionFactory =
                    new Configuration ( ).configure ( ).buildSessionFactory ( ) ;
```

Une SessionFactory ne devrait être généralement construite qu'une *seule fois*, par exemple avec une servlet chargée au démarrage, grâce à un *« load-on-startup »* dans le Web.xml , et elle doit aussi être un genre de « singleton ».

Le fichier hibernate.cfg.xml (et contact.hbm.xml) peut être créé dans le DWP:
Clic droit sur le dossier "src", et faîtes:
New → Other et choissez Hibernate → Hibernate Configuration File (cfg.xml)
→ Hibernate XML Mapping file (hbm.xml)

5) Ajouter maintenant la classe de test suivante:

```
package essai;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HiberTest {
                public static void main(String[] args) {
                  Session session = null;
                  try{

//lecture de  hibernate.cfg.xml et  preparation de  hibernate pour utilisation
SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
try{
      session =sessionFactory.openSession();


   }
catch(Exception r){
                              System.out.println("Erreur session : "+r.getMessage());
                              r.printStackTrace();
                }
```

```
//Creer une instance de  Contact et insertion dans la base
                System.out.println("Etape d'insertion");
                Contact contact = new Contact ( );
                contact.setId(3); //necesaire avant save ( ) à cause de assigned
                contact.setFirstName ("Joseph");
                contact.setLastName ("Ndong");
                contact.setEmail ("joesis00@gmail.com");
                session.save (contact);
                System.out.println ("c'est fait !!!");

              }catch(Exception e){
               System.out.println("Erreur :"+e.getMessage ( ));e.printStackTrace ( );
              }finally{
//l'insertion effective a lieu ici
               session.flush ( );
               session.close ( );

               }
            }
          }
```


HiberTest.java

Exécution de l’application

Pour l’exécution tous les jars de l’archive d’Hibernate doivent être ajoutés à l’application.

Si tout se passe bien la sortie ressemble à ceci:

```
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Inserting Record
Done
Hibernate: insert into contact (FIRSTNAME, LASTNAME, EMAIL, ID) values (?, ?, ?, ?)
```

Vous pouvez faire maintenant des exemples JEE véritables.

Commentaires.

La propriété **<mapping resource ="contact.hbm.xml" />** permet d'effectuer le mappage avec la table contact de la base de données.
Hibernate utilises les POJOs (Plain Old Java Object) ou simplement des **JavaBeans** pour le mappage des données dans la base de données. Vous remarquerez l'usage des **setters** pour effectuer le mappage proprement dit.
Retenez donc que le fichier **contact.hbm.xml** doit contenir les informations de mappage des objets de la classe **Contact** avec les *colonnes* de la table **contact** de la base de données.

Pour dialoguer avec une base de données, il faut utiliser un **dialect**; pour MySQL, le dialect est donné par la propriété (property): **org.hibernate.dialect.MySQLDialect .**

Voici une liste d'autres dialects :

DB2 :	org.hibernate.dialect.DB2Dialect
HypersonicSQL :	org.hibernate.dialect.HSQLDialect
Informix:	org.hibernate.dialect.InformixDialect
Ingres:	org.hibernate.dialect.IngresDialect
Interbase:	org.hibernate.dialect.InterbaseDialect
Pointbase:	org.hibernate.dialect.PointbaseDialect
PostgreSQL:	org.hibernate.dialect.PostgreSQLDialect
Microsoft SQL Server:	org.hibernate.dialect.SQLServerDialect
Oracle (any version):	org.hibernate.dialect.OracleDialect
Oracle 9:	org.hibernate.dialect.Oracle9Dialect
Progress:	org.hibernate.dialect.ProgressDialect
FrontBase:	org.hibernate.dialect.FrontbaseDialect
Sybase:	org.hibernate.dialect.SybaseDialect

La classe HiberTest sert à enregistrer quelques données dans la table contact.

Vue du fichier de mapping contact.hbm.xml

1) <hibernate-mapping> element

The first or root element of hibernate mapping document is <hibernate-mapping> element. Between the <**hibernate-mapping**> tag class element(s) are present.

2) <class> element

The **<class>** element maps the class object with corresponding entity in the database. It also tells what table in the database has to access and what column in that table it should use. Within one **<hibernate-mapping>** element, several <class> mappings are possible.

3) <id> element

The <id> element in unique identifier to identify and object. In fact <id> element map with the primary key of the table. In our code :

<id name="id" type="long" column="ID" >

primary key maps to the **ID** field of the table **CONTACT**. The attributes of the id element are:

name: The property name used by the persistent class.

column: The column used to store the primary key value.

type: The Java data type used.

unsaved-value: This is the value used to determine if a class has been made persistent. If the value of the id attribute is null, then it means that this object has not been persisted.

Vue du fichier de mapping contact.hbm.xml

4) <generator> element

The **<generator>** method is used to generate the primary key for the new record.
Here is some of the commonly used generators :

* **Increment** - This is used to generate primary keys of type long, short or int that are unique only. It should not be used in the clustered deployment environment.

* **Sequence** - Hibernate can also use the sequences to generate the primary key. It can be used with DB2, PostgreSQL, Oracle, SAP DB databases.

* **Assigned** - Assigned method is used when application code generates the primary key. In this case <generator> element do not generate the primary key and it is required to set the primary key value before calling **save ( )** method.

5) <property> element

The property elements define standard Java attributes and their mapping into database schema. The property element supports the column child element to specify additional properties, such as the index name on a column or a specific column type.

Mise de jour (Update)

Comment utiliser hibernate pour faire une mise d'un enregistrement d'une table.
Je suppose que j'ai une table **compte** **(NUM, SOLDE, NOM)** pour gérer un compte bancaire.
Voici comment faire pour mettre à jour le premier enregistrement, par exemple.

Je suppose qu'on a une classe **Compte** qui est un JavaBean (un compte étant identifié par
Les attributs **num** (long) **solde** (double) et **nom** (String).

Je crée la classe **UpdateExample** suivante pour faire la mise à jour.

```
package joe.hibertest;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
public class UpdateExample {
  public static void main(String[] args) {
    Session sess = null;
    try {
      SessionFactory fact = new Configuration().configure().buildSessionFactory();
      sess = fact.openSession ( );
      Transaction tr = sess.beginTransaction ( );
      Compte ins = (Compte)sess.get(Compte.class, new Long (1));
      ins.setNom ("NDONG");
      ins.setSolde(20000);
      sess.update (ins);
      tr.commit ( );
      sess.close ( );
      System.out.println ("Update successfully!");
    }
    catch (Exception e){      System.out.println(e.getMessage());    }  }
}
```

Numéro de l'enregistrement

Je rajoute ces informations dans le fichier de mapping **contact.hbm.xml**

```
<class name="joe.hibertest.Compte" table="compte">
            <id name="num" type="long" column="NUM" >
            <generator class="assigned"/>
           </id>

           <property name="solde">
                    <column name="SOLDE" />
           </property>
           <property name="nom">
                   <column name="NOM"/>
           </property>

  </class>
```

Sélection (clause select)

```
package joe.hibertest;
import org.hibernate.Session; import org.hibernate.*; import org.hibernate.cfg.*;
import java.util.*;
public class SelectHQLExample {
  public static void main (String [ ] args) {
  Session session = null;
  try{
            SessionFactory sessionFactory = new
            Configuration ( ).configure ( ).buildSessionFactory ( );
            session = sessionFactory.openSession ( );
      String SQL_QUERY ="from Contact contact";
     Query query = session.createQuery (SQL_QUERY);
  for (Iterator it = query.iterate ( );it.hasNext();)
  {
             Contact c = (Contact)it.next ( );
             System.out.println ("ID: " + c.getId ( ));
             System.out.println ("First Name: " + c.getFirstName ( ));
   }
    session.close ( );
}catch (Exception e){    System.out.println(e.getMessage ( ));      }
  }  }
```

Hibernate Query Language (HQL)

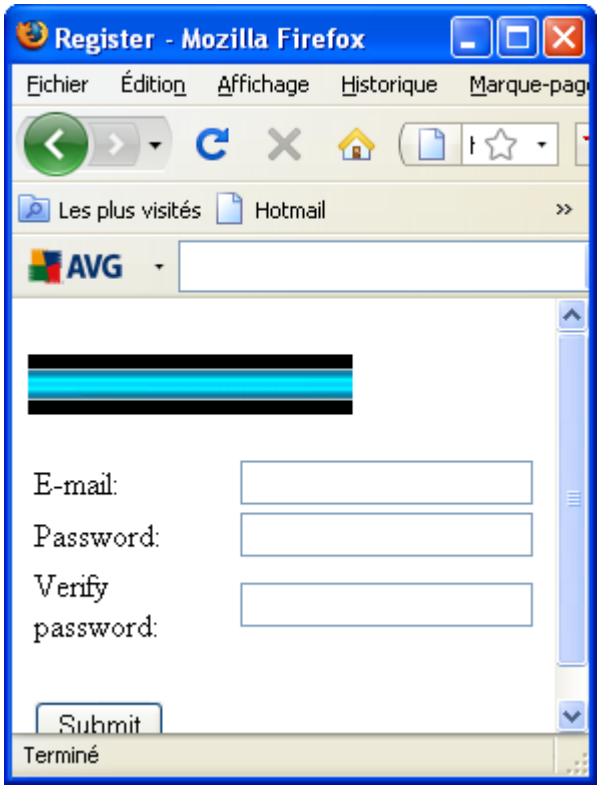
HQL est un outil puissant utilisé pour envoyer des requêtes sur une base de données.
Il est insensible à la casse, excepté pour les noms de classes Java et les propriétés des objets de ces classes.
Si vous utiliser HQL dans votre application, Hibernate génère automatiquement les requêtes SQL
HQL est basé sur les modèles objet/relationnel et produit un langage SQL orienté objet.

Les requêtes HQL utilise des **classes Java et leurs propriétés** au lieu des *colonnes* et *tables* de la base de données comme on le fait classiquement avec du SQL standard.

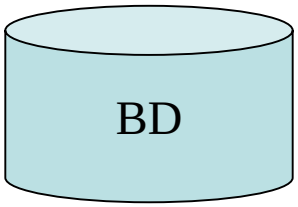
Struts + Spring + Hibernate dans un exemple

Création + *validation* + *récupération* des paramètres du formulaire (***DynaBean***) par Struts

Chargement depuis **Struts** dans **Spring** des données du ***DynaBean*** Spring crée un pool d'objets (***factory***) encapsulant les données.



Hibernate récupère un objet depuis Spring et mappe cet objet dans une table de la BD.



Voici les différentes étapes:

1) Création des formulaires (pages JSPs: register.jsp [page d'accueil] et *login.jsp* [bienvenue].
2) Configuration de *struts-config.xml* pour définir le DynaBean (JavaBean dynamique Struts) qui doit récupérer par réflexion les données fournies dans la page *register.jsp* .
3) Bien configurer Struts dans le fichier *web.xml* (*ActionServlet* et tag *struts-html.tld*)
4) Création d'un fichier PROPERTIES pour validation du formulaires (saisies erronées) Mise en place de *validator-rules.xml* et *validation.xml*)
5) Création de l'Action Struts (qui dit ce que l'on fait après validation du formulaire).
6) Configuration de Spring (mise en place d'un factory de bean)
7) Chargement de ce bean grâce à Struts (au bean *DynaValidatorActionForm*)
8) Création d'un *plug-in* Struts pour Hibernate (charger le fichier *hibnernate.cfg.xml*)
9) Configuration de Hibernate pour mappage Objet/relationnel (mapping du bean Spring dans la base de données)

```
<html:html>
<head><title>Register</title></head>
<font color="red"><html:errors/></font>
<body>
<table width="162", border="0",
cellspacing="0" cellpadding="0">
<tr>    <td> </td>  </tr>
<tr bgcolor="#000000">
  <td height="30" width="162">
  <div align="left">
     <img  src="barreseule.gif"
 width="162"   height="30">
  </div>
  </td>
  </tr>
  <tr>
  <td> </td>
  </tr>
</table>
<%-- Form display --%>
```

register.jsp

```
<html:form action="registerAction.do">
<table>
<tr><td>E-mail:</td> <td>
<html:text property="email"/></td></tr>
<tr><td>Password:</td> <td>
<html:password property="password"/></td>
</tr>
<tr><td>Verify password:</td> <td>
<html:password property="verify_password"/>
</td></tr>
<tr><td><br/></td></tr>
<tr><td><html:submit value="Submit"
property="Submit"/></td></tr>
</table>
</html:form>
<font color="red">
<html:javascript formName="/registerAction"
dynamicJavascript="true"
staticJavascript="true " />
</font>
</body>
</html:html>
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<jsp:useBean id="mybean" class="joe.User" scope="request"/>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
Bienvenue dans notre site! <%= mybean.getEmail ( ) %>
</body>
</html>
```

login.jsp

```
(…) <struts-config>
<form-beans>
<form-bean name="registerActionForm“
 type="org.apache.struts.validator.DynaValidatorActionForm">
<form-property name="email" type="java.lang.String" />
<form-property name="password" type="java.lang.String" />
<form-property name="verify_password" type="java.lang.String" />
</form-bean>
</form-beans>
<action-mappings>
<action input="/register.jsp" name="registerActionForm" path="/registerAction"
scope="session" validate="true" type="eyrolles.RegisterAction" >
<forward name="register" path="/register.jsp" />
<forward name="login" path="/login.jsp" />
</action>
</action-mappings>
<message-resources parameter="ApplicationResources" />
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
<set-property property="pathnames" value="/WEB-INF/validator-rules.xml,
/WEB-INF/validation.xml" />
</plug-in>
<plug-in className =“joe.HibernatePlugin">
<set-property property="path" value="/hibernate.cfg.xml" />
</plug-in></struts-config>
```

struts-config.xml

```
# Struts Validator Error Messages
errors.required={0} est requis.
errors.minlength={0} doit avoir plus de {1} caracteres
errors.maxlength={0} ne doit pas avoir plus de {2} caracteres

#Other error messages
errors.passwords.different=Les mots de passe de concordent pas.
errors.email.found=Cet email est déjà enregistré.

# Parameters of the form
label.email=Le champ email
label.password=Le champ mot de passe
label.verify_password=le champ verification mot de passe
```

ApplicationResources.properties

```
(…)
<form-validation>
<formset>
<form name="/registerAction">
        <field property="email" depends="required">
                <arg0 key="label.email"/>
        </field>
        <field property="password" depends="required,minlength,maxlength">
                <arg0 key="label.password"/>
                <arg1 name="minlength" key="${var:minlength}" resource="false"/>
                                <var>
                                        <var-name>minlength</var-name>
                                        <var-value>8</var-value>
                                </var>
                <arg2 name="maxlength" key="${var:maxlength}" resource="false"/>
                                <var>
                                        <var-name>maxlength</var-name>
                                        <var-value>36</var-value>
                                </var>
        </field>
</form>
</formset>
</form-validation>
```

validation.xml

Action Struts: class RegisterAction

RegisterAction.java

```
import javax.servlet.http.*;
import org.apache.struts.*;
import org.apache.struts.action.*;
import org.apache.struts.validator.*;
import org.hibernate.*;
import org.hibernate.criterion.*;
import org.springframework.beans.factory.*;
import org.springframework.beans.factory.xml.*;
import org.springframework.core.io.*;
public class RegisterAction extends Action
{   private static Class<RegisterAction> thisClass = RegisterAction.class;
     /*
   Méthode de vérification du mot de passe
   */
   private boolean isValidPassword(HttpServletRequest request, String password,
String verify_password)
   {   boolean valid = false;
       if (password.matches(verify_password)) { valid = true; }
       return valid;
   }
```

```
/*
Recherche dans la base de données si l'email n'est pas déjà enregistré .
  */
  private boolean isUniqueEmail (Session session, HttpServletRequest request, String email)
  {   boolean result = true;
     Criteria criteria = session.createCriteria (User.class);
      criteria.add (Expression.eq("email", email));
      criteria.setMaxResults (1);
      User duplicate = (User) criteria.uniqueResult ( );
      if (duplicate != null)
      {    if (duplicate.getEmail().matches(email))
          { result = false; // retourne false si l'email est déjà présent dans la base
          }
      }
          return result;
  }
```

```
/*
  Enregistrement effectif d'un nouvel utilisateur dans la base de données
  */
  public void registerUser (Session session, HttpServletRequest request,
                                DynaValidatorActionForm dform)
  {  Resource res = new ClassPathResource ("user.xml");
     BeanFactory factory = new XmlBeanFactory(res);
     User user = (User) factory.getBean("user");
     user.setEmail((String)dform.get("email"));
     user.setPassword ((String) dform.get ("password"));
     request.setAttribute ("mybean", user); // pour etre utilisé dans login.jsp
     org.hibernate.Transaction tx = session.beginTransaction();
     session.save (user);
     tx.commit();
  }
```

```
/**
   * sauvegarde les messages d'erreur dans la requete appropriée
   * attribute qui sera utilisé par le tag  <html:errors>  de la page JSP, if any messages
   * S4il n'y a pas d'erreur l'attribut de requete n'est pas créé.
   *
   * @param request The servlet request we are processing
   * @param errors Error messages object
   */
   protected void saveErrors (HttpServletRequest request, ActionMessages errors)
   {   // s'il n'y a pas d'erreurs, on supprime tous les attributs d'erreur de la requete
if ((errors == null) || errors.isEmpty())
       {   request.removeAttribute(Globals.ERROR_KEY);
           return;
       }
       // sauve l'attribut dans la requete
request.setAttribute(Globals.ERROR_KEY, errors);
   }
```

```
public ActionForward execute (ActionMapping mapping, ActionForm form,
HttpServletRequest request, HttpServletResponse response)
   {   String forward = "register";
      SessionFactory sessionFactory = (SessionFactory) servlet.getServletContext().
                                                getAttribute(HibernatePlugin.KEY_NAME);
      Session session = sessionFactory.openSession();
      DynaValidatorActionForm dform = (DynaValidatorActionForm) form;
      ActionMessages errors = dform.validate (mapping, request);
      if (errors.isEmpty ( ))
      { try
         { if (!isValidPassword(request, (String) dform.get ("password"),
                  (String) dform.get("verify_password")))
     { errors.add(Globals.ERROR_KEY, new ActionMessage("errors.passwords.different"));
            }
            else if (!isUniqueEmail(session, request, (String) dform.get("email")))
            { errors.add(Globals.ERROR_KEY, new ActionMessage("errors.email.found"));
            }
            else              {registerUser(session, request, dform);  forward = "login";
                              }          }
         catch (Exception e) { e.printStackTrace(); }
         finally          {  session.flush();    session.close();          }       }
      saveErrors(request, errors);       return mapping.findForward(forward);
   }}
```

HibernatePlugin.java

```
package joe;
import java.net.URL;
import javax.servlet.ServletException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.struts.action.ActionServlet;
import org.apache.struts.action.PlugIn;
import org.apache.struts.config.ModuleConfig;
import org.hibernate.cfg.*;
import org.hibernate.*;
public class HibernatePlugin implements PlugIn
{
private Configuration config;
private SessionFactory factory;
private String path = "/hibernate.cfg.xml";
private static Class<HibernatePlugin> hpclass = HibernatePlugin.class;
public static final String KEY_NAME = hpclass.getName();
private static Log log = LogFactory.getLog(hpclass);
public void setPath(String path) { this.path = path; }
public void init (ActionServlet servlet, ModuleConfig modConfig) throws ServletException
{
```

```
try  {  URL url = HibernatePlugin.class.getResource (path);
        config = new Configuration ( ).configure (url);
        factory = config.buildSessionFactory ( );
        servlet.getServletContext ( ).setAttribute (KEY_NAME, factory);
     }
catch (MappingException e)
{        log.error ("Mapping errors:", e);
                 String [] messages = e.getMessages();
                 for (int i = 0; i < messages.length; i++)
                 log.error(messages[i], e);
                throw new ServletException ( );
}
catch (HibernateException e)
{         log.error ("Hibernate errors:", e);
                 String [] errors = e.getMessages();
                 for (int i = 0; i < errors.length; i++)
                 log.error (errors[i], e);
                throw new ServletException ( );
}}
public void destroy ( )
{ try { factory.close ( ); }
catch (HibernateException e) { log.error (“Ne pas fermer la  factory", e); }
}}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<!-- This (Spring) xml file defines the bean used with the Spring bean factory object -->
<beans>
        <bean id="user" class="joe.User">
        </bean>
</beans>
```

user.xml

```java
package joe;
public class User
{
private long id;
private String email;
private String password;
public User() {}
public void setId(long id) { this.id = id; }
public long getId() { return id; }
public String getEmail() { return email; }
public String getPassword() { return password; }
public void setEmail(String email) { this.email = email; }
public void setPassword(String password) { this.password = password; }
public String toString() { return ""; }
public boolean equals(Object o) { return false; }
public int hashCode() { return 0; }
}
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- This (Hibernate) xml file maps all of the properties in the User class to the
      conresponding columns in the database -->
<hibernate-mapping>
<class name="joe.User" table="users">
<id name="id" type="long" column="id" >
<generator class="increment"/>
</id>
<property name="email">
<column name="email"/>
</property>
<property name="password">
<column name="password"/>
</property>
</class>
</hibernate-mapping>
```

user.hbm.xml

```
(…)
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost/tutorial</property>
<property name="hibernate.connection.username">root</property>
<property name="hibernate.connection.password"></property>
<property name="hibernate.connection.pool_size">10</property>
<property name="show_sql">true</property>
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
<property name="hibernate.hbm2ddl.auto">update</property>
<property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider
                    </property>
<property name="hibernate.cache.use_query_cache">true</property>
<!-- Set this to make batch updates larger: -->
<property name="hibernate.jdbc.batch_size">20</property>
<property name="hibernate.cglib.use_reflection_optimizer">true</property>
<!-- Mapping files -->
<mapping resource="user.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

hibernate.cfg.xml

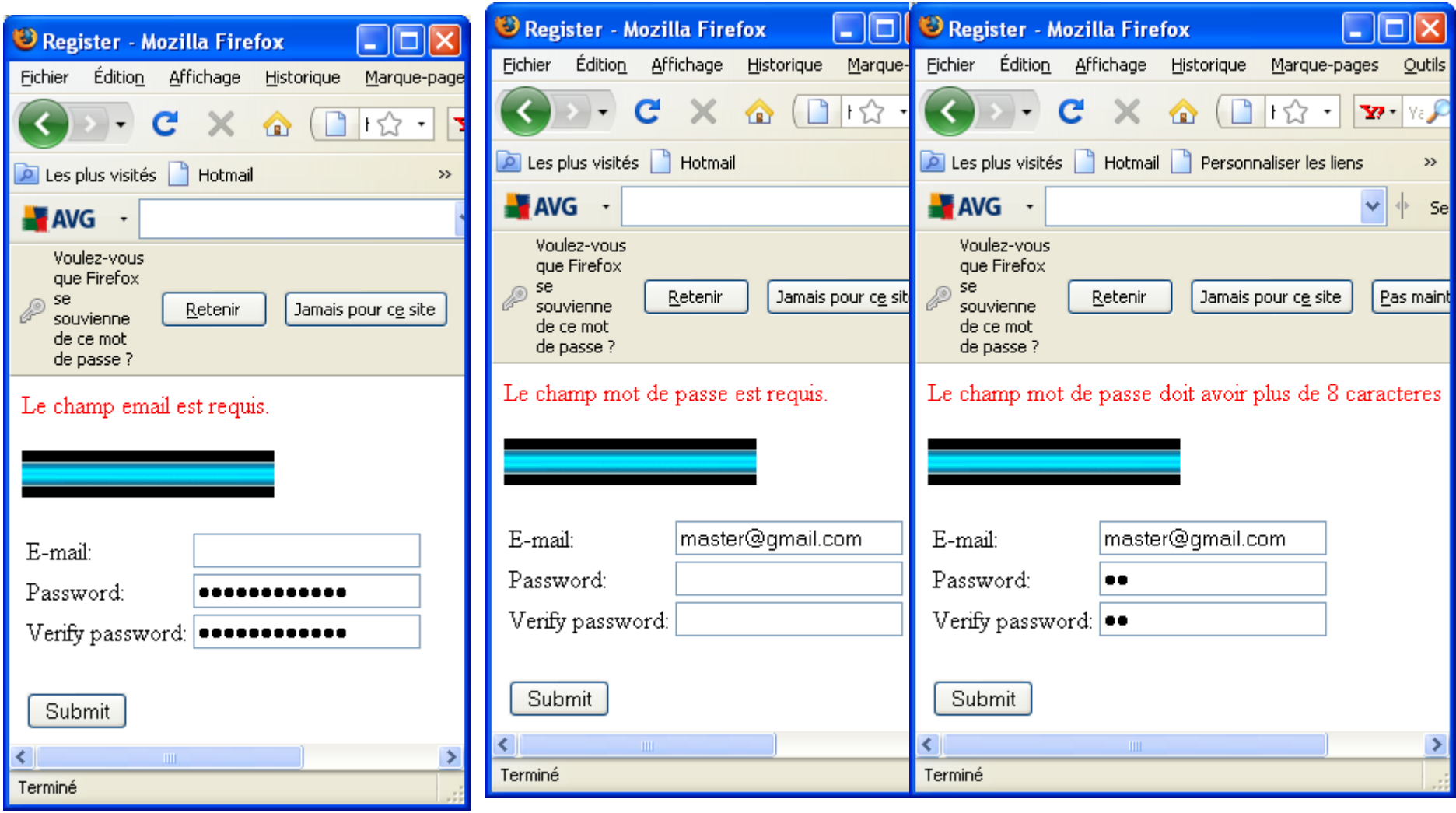
Localisation des différentes ressources de l'application:

Les fichiers **user.xml** **user.hbm.xml** et **hibernate.cfg.xml** et **ApplicationResources.properties** doivent se trouvent dans le répertoire des sources Java.

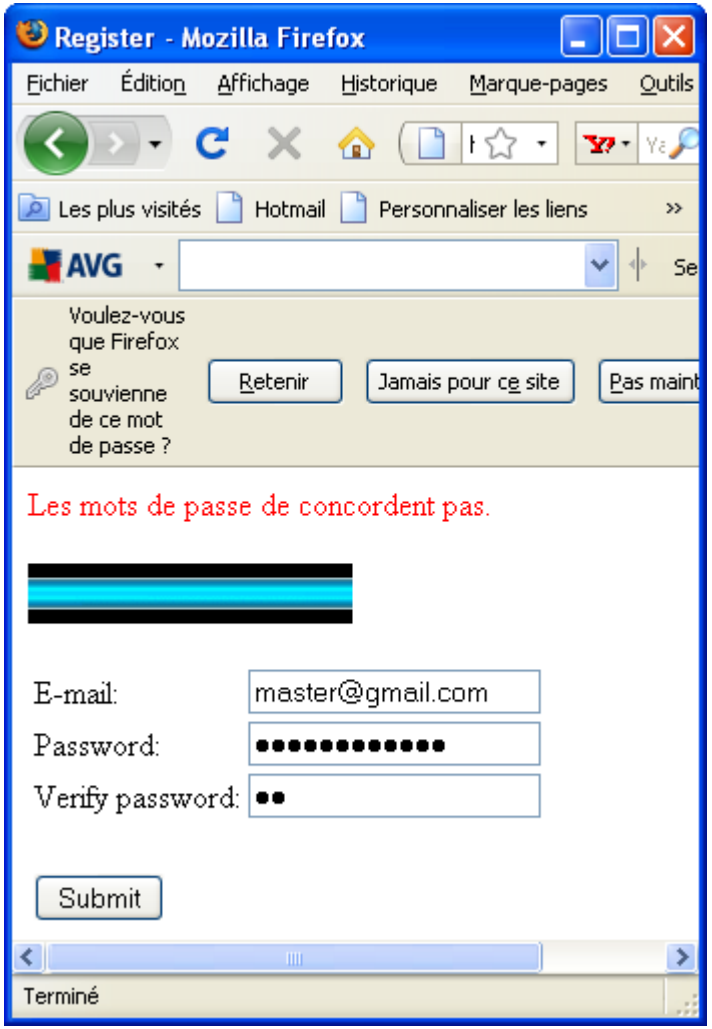
Dans le fichier ***user.xml***, on configure les classes beans que **Spring** doit accéder pour générer une ***factory*** de beans directement utilisables.
Hibernate utilise le fichier **hibernate.cfg.xml** pour effectuer la configuration de la base de données et le fichier **user.hbm.xml** pour effectuer le mappage entre la classe **User.class** et la table **users** de la base de données (tutorial utilisée dans cet exemple).

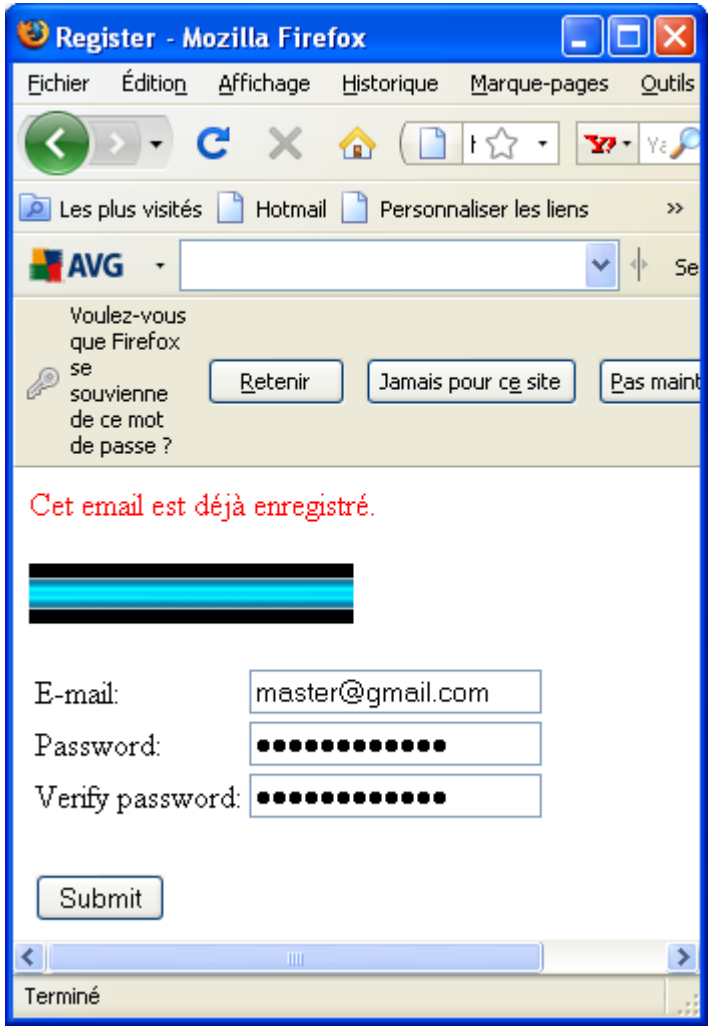
Les fichiers *validation.xml*, *validator-rules.xml*, *struts-config.xml* sont à placer dans le répertoire **WEB-INF** de l'application Web.

Les fichiers TLDs de **Struts** (ici on a seulement utilisé ***struts-html.tld***) sont à placer dans le répertoire **WEB-INF** ou un des sous répertoires de **WEB-INF** (exemple ***tlds***) et il faut les déclarer bien sur dans le fichier **web.xml**.

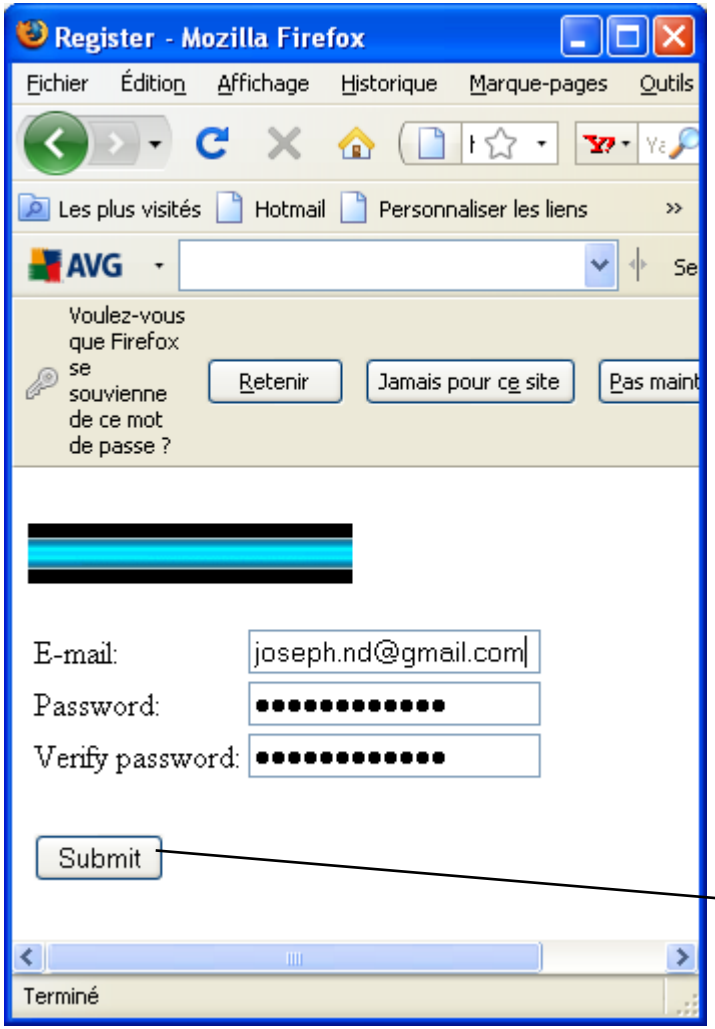


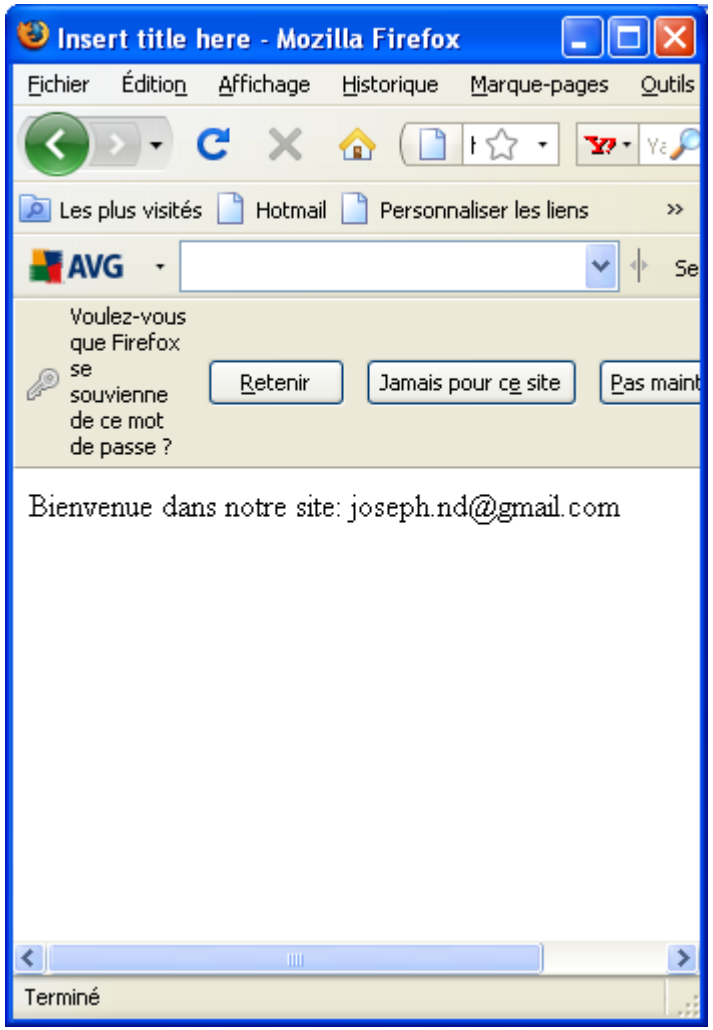
Exemples de validations gérées coté **client**.





Exemples de validations gérées coté **serveur**.





Validation correcte !!!

Hibernate Annotations

Dans cet exemple, nous montrons comment, avec les annotations, hibernate permet de Créer des tables dans la base de données. La table que nous allons créer correspond simplement à un objet (classe java).
En plus de la création dynamique de la table Client, nous montrons aussi l'insertion d'un enregistrement dans la table.
Nous utilisons des classes Java simples.

Les Annotations sont gérées par un méchanisme de réflexion durant la phase de runtime par la JVM.

Matériel nécessaire:
Commencez par créer un Dynamic Web Project (DWP):

1) **Download Hibernate Core (par exemple HibernateTools-3.2.4.GA.zip)**
2) **Décompressez dans un répertoire de votre choix**
3) **Rentrez dans \plugins\org.hibernate.eclipse_3.2.4.GA-R200905070146-H18\lib**
4) **Copier tous les jars des sous dossiers annotations, hibernate, tools dans \WebContent\WEB-INF\lib de votre DWP.**

```
package beans;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity // this tell to Hibernate that this class is a entity in the database
public class Client {
@Id // this tell to hibernate that id is the primary key of our table
private int id;
private String name;
public int getId ( )          {
                                  return id;
                                }
public void setId (int id)  {
                                  this.id = id;
                                }
public String getName ( ) {
                                    return name;
                                }
public void setName (String name) {
                                          this.name = name;
                                        }
}
```

C'est cette classe qui sera la table Client à ajouter dans la base.

```
hibernate.query.substitutions true 1, false 0, yes 'Y', no 'N'
## MySQL
hibernate.dialect org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class com.mysql.jdbc.Driver
## Specify the host and database name
hibernate.connection.url jdbc:mysql://localhost/tutorial
## Specify schema, username and password of the database
hibernate.default_schema
hibernate.connection.username root
hibernate.connection.password
## Some Hibernate settings
hibernate.connection.pool_size 1
hibernate.proxool.pool_alias pool1
hibernate.show_sql true
hibernate.order_updates true
hibernate.max_fetch_depth 1
hibernate.default_batch_fetch_size 8
hibernate.jdbc.batch_versioned_data true
hibernate.jdbc.use_streams_for_binary true
hibernate.cache.region_prefix hibernate.test
hibernate.cache.provider_class org.hibernate.cache.HashtableCacheProvider
```

Ce fichier configure l'accès à la BD.

hibernate.properties

```
log4j.rootCategory=DEBUG, Default
log4j.appender.Default=org.apache.log4j.FileAppender
log4j.appender.Default.File=test.log
log4j.appender.Default.layout=org.apache.log4j.PatternLayout
# A default format: [Category] msg\n
log4j.appender.Default.layout.ConversionPattern=[%c{1},%p] %m%n
# A format useful for debugging concurrency: [Category,thread,elapased_ms]
msg\n
#log4j.appender.Default.layout.ConversionPattern=[%c{1},%t,%r] %m%n
# Do not truncate if it already exists.
log4j.appender.Default.Append=true
```

log4j.properties

Ce fichier configure les messages de Log (sur la console).

Les deux fichiers PROPERTIES sont à placer sous le répertoire \SRC de votre projet.

```
package util;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.cfg.AnnotationConfiguration;
import beans.Client; // we import our class here
public abstract class HibernateUtil {
private static org.hibernate.SessionFactory fabrique;
static {
        AnnotationConfiguration conf = new AnnotationConfiguration ( );
        conf.addAnnotatedClass (Client.class); // we add the Client class here
        // if you create more classes at your project, you must add them here
        fabrique = conf.buildSessionFactory ( );
      }
public static Session getSession ( ) {
                                        return fabrique.openSession ( );
                                    }
public static void closeSession ( ) throws HibernateException {
                                                                }
}
```

Cette classe crée les sessions hibernate pour accéder à la base.

```
package util;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.tool.hbm2ddl.SchemaExport;
import beans.Client;
/**
* Create tables based on beans
*/
public class DataGeneration {
public static void main (String [ ] args) {
AnnotationConfiguration conf = new AnnotationConfiguration ( );
conf.addAnnotatedClass (Client.class);
SchemaExport sE = new SchemaExport (conf);
sE.create (true, true);
}
}
```

Cette classe crée effectivement la table Client dans la base.
Si cela se passe bien, vous aurez le message ci dessous.

```
drop table if exists .Client
create table .Client (id integer not null, name varchar(255), primary key (id))
```

```
package util;
import org.hibernate.Session;
import beans.Client;
public class Test {

public static void main (String [ ] args) {
// creating a client
Client c = new Client ( );
// setting attributes
c.setId (1);
c.setName ("Joseph");
//Geting a Hibernate session
Session session = HibernateUtil.getSession ( );
//Starting the transaction
session.beginTransaction ( );
//Inserting the costumer
session.save (c);
//Commiting the transaction
session.getTransaction ( ).commit ( );
}
}
```

Cette classe insère un enregistrement dans la table Client précédemment créée.

Hibernate: insert into .Client (name, id) values (?, ?)

Deuxième exemple: utilisation de hibernate.cfg.xml

Au lieu d'utiliser le fichier hibernate.properties, on peut utiliser le fichier de configuration d'Hibernate hibernate.cfg.xml .
Vous conservez le fichier log4j.properties pour les messages console.

La seule modification dans ce fichier est l'ajout de cette ligne.

```
<mapping class="beans.Employee"/>
```

Employee est la classe bean utilisée pour créer la table du même nom.

```
package beans;
import java.io.Serializable;        import javax.persistence.Column;
import javax.persistence.Entity; import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "Employee")
public class Employee implements Serializable {
  public Employee ( ) {  }
  @Id
  @Column(name = "id")
  Integer id;
  @Column(name = "name")
  String name;
  // mettre ici les getters and setters pour les champs id et name
}
```

```
package util2;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
public class HibernateUtil {
  private static final SessionFactory sessionFactory;
  static {
    try {
              // Create the SessionFactory from hibernate.cfg.xml
      sessionFactory = new AnnotationConfiguration().configure().buildSessionFactory( );
    }
 catch (Throwable ex) {

    }
  }
  public static SessionFactory getSessionFactory ( ) {
    return sessionFactory;
  }
}
```

```
package util2;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.tool.hbm2ddl.SchemaExport;
import beans.Employee;
/**
* Create tables based Employee on beans Employee
*/
public class TableGeneration {
public static void main(String[] args) {
AnnotationConfiguration conf = new AnnotationConfiguration ( );
conf.addAnnotatedClass (Employee.class);
SchemaExport sE = new SchemaExport (conf);
sE.create (true, true);
}
}
```

L'exécution de cette classe génère la table Employee, le message suivant le confirme.

```
drop table if exists .Employee
create table .Employee (id integer not null, name varchar(255), primary key (id))
```

```
package beans;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import util2.HibernateUtil;
public class TestEmployee{
  public static void main (String [ ] args) throws Exception {
    /*Utilisation de la  Session Factory et d'une session */
    SessionFactory session = HibernateUtil.getSessionFactory ( );
    Session sess = session.getCurrentSession ( );
    /* demarrage de la Transaction */
    Transaction tx = sess.beginTransaction ( );
    Employee pojo = new Employee ( );
    pojo.setId (new Integer (5));
    pojo.setName ("NDONG");
    /*Enregistrement de l'objet  */
    sess.save (pojo);
    /* Commiting the changes */
    tx.commit ( );
    System.out.println ("Record Inserted");
    /*Fermeture de la  Session */
    session.close ( );
  }}
```

```
Hibernate: insert into .Employee (name, id) values (?, ?)
Record Inserted
```

Java Server Faces

JSF (**Java Server Faces**) est une technologie JAVA (framework) permettant la création d'interfaces WEB,
Comme **Struts**, ***JSF est une implémentation du design pattern MVC***.
Il bénéficie des apports de STRUTS et des concepts du monde
JAVA-JEE (Swing, modèle événementiel, JSP, Servlets).
Mais JSF va plus loin encore en apportant un lots de de nouveaux concepts.
JSF est un **standard**.

JSF permet:

- une séparation entre la couche présentation et les autres couches d'une application web
- une mise en place d'un **mapping HTML/OBJET**
- la **ré-utilisation** de composants graphiques
- une gestion de l'état de l'interface entre les différentes requêtes
- une **liaison** entre les actions coté « Client » et les actions des objets Java coté « Serveur » création de **composants customs** grâce à une API
- le **support de différents clients** (HTML, WML, XML, ...) grâce à la séparation des problématiques de construction de l'interface et du rendu de cette interface

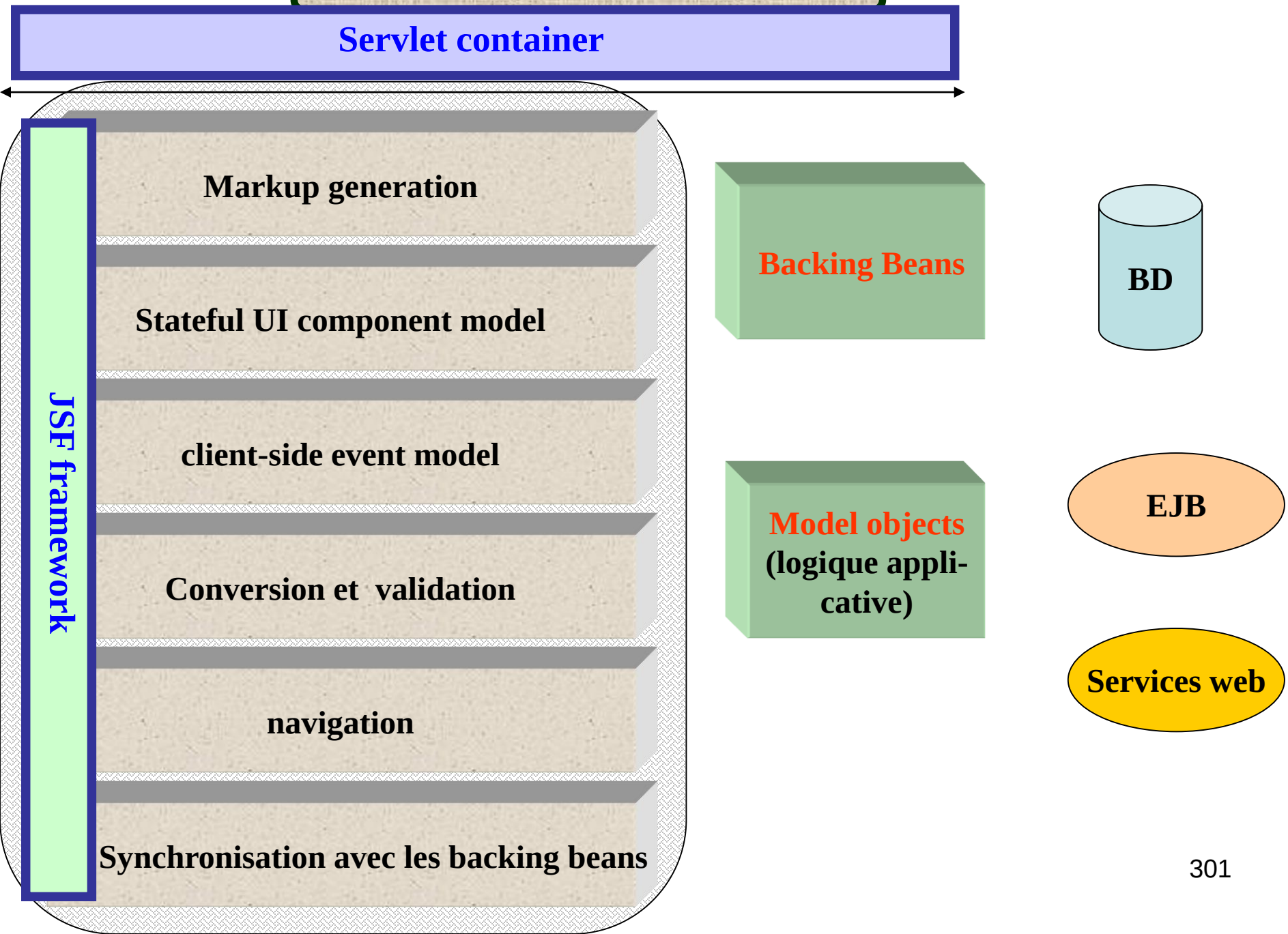
Il existe deux implémentations de ce framework : **SUN** et **MyFaces**.
Nous utilisons les deux.

Notez que JSF est une combinaison fantastique de framework existants. Il est développé en ne tenant compte que des avantages de ces derniers.

JSF fait partie de JEE 1.4

C'est un produit de la communauté **JCP** (***Java Communuty Process***) et ainsi c'est le fruit d'un travail collaboratif entre plusieurs experts du monde du développement d'applications Web, comme ***Jakarta Struts, Oracle, Sun, IBM, ATG,*** etc.
Java Server Faces est une spécification : il est donc nécessaire d'obtenir une implémentation de la part d'un tiers.
Plusieurs implémentations commerciales ou libres sont disponibles, notamment l'implémentation de référence de *Sun* et *MyFaces* qui est devenu un projet du groupe Apache.

Architecture JSF



Composants JSF

JSF offre plusieurs composants :

- Une gigantesque API pour gérer l'état des composants pour *la validation coté serveur*, la *gestion des événements*, la *navigation des pages*, la *conversion de données*, etc….
- des librairies de tags JSP pour créer des composants graphiques afin de bien définir les vues.

Le but de JSF est d'aider à créer des applications web de manière facile et rapide.
Avec JSF, le développeur se concentre sur la création des composants graphiques (UI components), de la gestion des évènements, de la création des beans et leur interaction, au lieu de se focaliser sur le gestion des *requêtes* et ***réponses***.

Versions de JSF:

JSF 1.2 (mai **2006**) - version la plus récente. **JSR** : Java Specification request 252.
JSF 1.1 (mai **2004**) - **JSR** : 127
JSF 1.0 (mars **2004**) -
La version 1.2 s'exécute avec la spécification des **Servlet 2.5** et **JSP 2.1**.
ATTENTION: cette version marche avec **Tomcat 6.0** et non **5.5**.

Composants JSF

Les composants dans JSF (UI components) sont des zones de texte, des boutons, des Tables, etc. ce sont ces objets qui vont interagir avec l'utilisateur.

Vous pourrez créer :
- des ***composants simples*** (simple components): text box, button
-des ***composants complexes*** (compound components) :table, data grid.

Donc sachez que JSF permet de générer deux types de composants:
1) **composants simples** (**standard UI Components):**
zones de texte (text fields) , listes déroulantes (list boxes) , cases à cocher (check boxes), panneau (panel) , label, bouton radio (radio button) etc.

Exemples:

UIForm représente un conteneur pour d'autres composants.
UICommand représente des composants UI comme buton, hyperlinks et menus.
UIInput représente des composants UI comme zone de texte.

2) **composants personnalisables** (**Custom UI Components) :**
comme les objets calendar, qui ne sont pas des composants standards JSFs.

Les Tags JSF

Les applications JSF utilisent typiquement des pages JSPs pour la représentation des vues.
JSF fournit une bibliothèque de tags pour la construction de ces vues.
Chaque tag est associée à un type de composant. Et JSF fournit 43 tags réparties sous forme de deux classes de librairies de tags:

- **JSF Core Tags Library** et
- **JSF Html Tags Library**

Une page JSP utilisera ces tags de la façon suivante:

```
<%@ taglib uri = ”http: //java.sun.com/jsf/core “ prefix=”f” %>
<%@ taglib uri = ”http://java.sun.com/jsf/html “ prefix=”h” %>
```

```
<f:view>
   <h:form>
      ……————————
      ……————————
   </h:form>
</f:view>
```

JSF Html Tags:

Ces tags représente des composants html (zone de texte, bouton , formulaires,…)
Ces tags Html sont divisées en catégories:

Inputs	(inputText, inputTextarea)
Outputs	(outputText, outputLabel)
Commands	(commandButton)
Selections	(selectOneRadio, selectOneListbox, selectOneMenu pour radio buttons, list boxes, menu etc)
Layouts	(panelGrid)
Data table	(dataTable)
Errors and messages	(message, messages)

Voici quelques exemple d'utilisation de ces tags:

<h:inputText id="ID1" value="value" /> **:**

crée une zone de texte, l'attribut id est l'identifiant unique du composant et la valeur par défaut du composant est donnée par l'attribut valeur.

```
<h:commandButton
                id = "submit"
                value = "go"
                action = "nextPage">
</h:commandButton>
```

Bean à invoquer après validation du bouton.

JSF Core Tags:

Ces tags JSF permettent de gérer les validations, évènements, conversions, etc.…
La ***librairie Core*** est compatible (supporte) la ***librairie html***.
La librairie Core contient des tags pour les vues et les sous vue, elle contient aussi des tags pour le chargement de ressources (*resource bundle*), pour l'ajout de texte dans les pages.

Exemple:

f:view	utilisé pour construire la vue parente
f: subview	utilisée pour créer une sous vue d'une vue.
f: validator	pour ajouter un validateur (*validator*) pour un composant.
f: converter	pour ajouter un *converter* à un component.
f: actionListener	pour ajouter un *listener* à un component.
f:valueChangeListener	pour ajouter un *valuechange listener* à un component

Exemple 1:

```
<f:view locale="en">
                    <h:outputText value="label" />
</f:view>
```

Exemple 2:

```
<f:view>
        <h1>head</h1>
         <p>view</p>
      <f:subview id="sub_id">
            <c:import url="second.jsp" />
     </f:subview>
</f:view>
```

Concepts clés de JSF

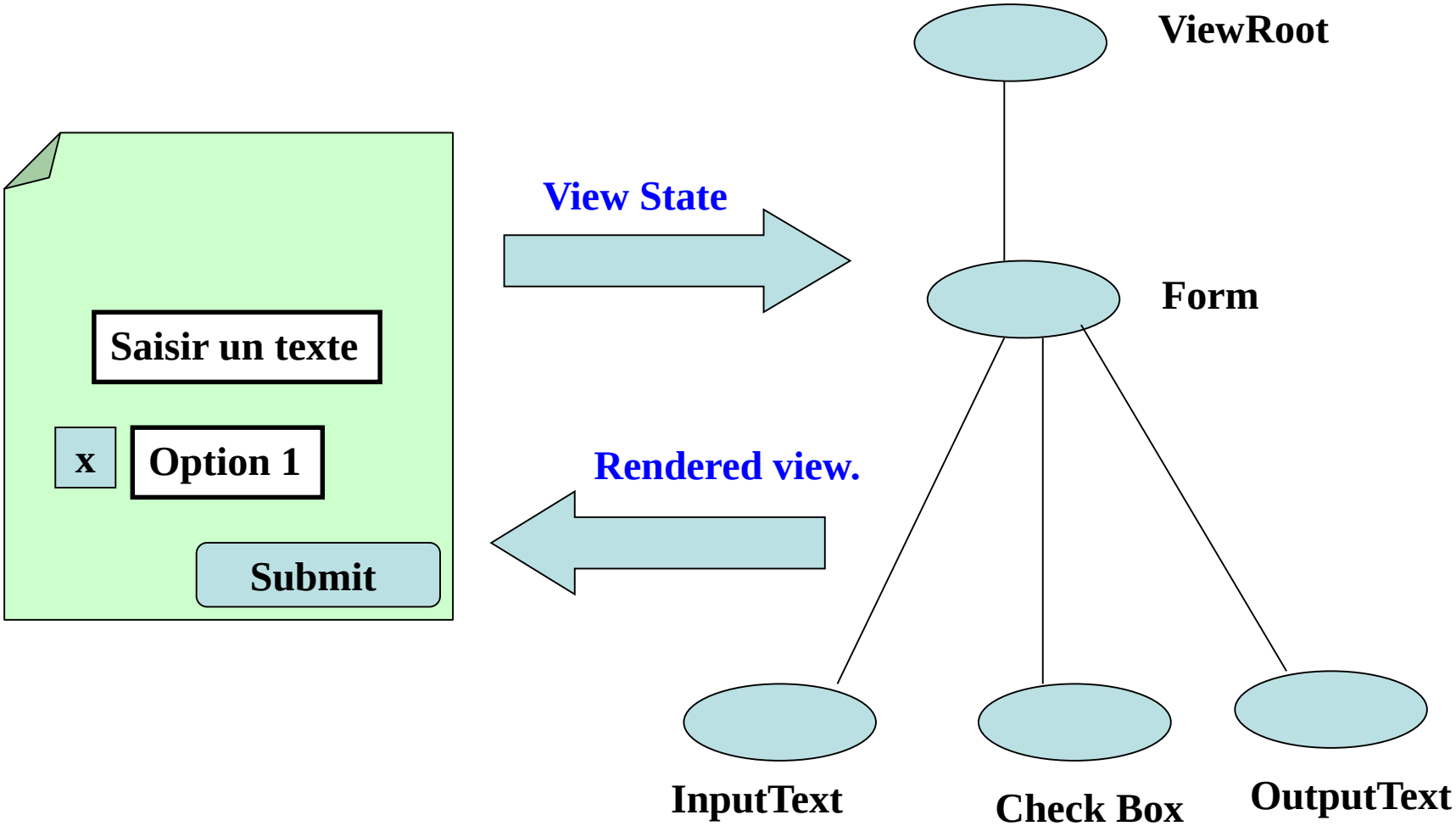
Le framework JSF fournit principalement huit (8) concepts majeurs à maîtriser absolument.

1. **UICOMPONENT**
2. **RENDERER**
3. **VALIDATOR**
4. **BACKING BEANS**
5. **CONVERTER**
6. **EVENTS & LISTENER**
7. **MESSAGES**
8. **NAVIGATION.**

Nous allons les détailler à présent**.**

1. UICOMPONENT

Il s'agit de gérer ***l'abstraction*** et ***l'encapsulation*** de fonctionnalités pour un composant ayant un *état* et une *représentation graphique*.



2. RENDERER

- Renderer par défaut : HTML 4.0
- Implémentation par model direct VS implémentation par modèle de délégation
- Organisé en render kits

3 CONVERTER

- se charge de l'encoding et du decoding.
- les converter de base fournis par JSF

```
<f:convertNumber />
<f:convertDateTime />
```

4. VALIDATOR

-implémenté directement dans le composant
-Via une méthode du backing bean
-- via une classe validateur générique

```
<h:inputText value=" #{user.name}" >
   <f:validatelength minimum="13" />
</h:inputText>
```

```
<h:inputText value=" #{user.Banque}" >
   <f:validator validatorID="joe.jsf.BanqueValidator" />
</h:inputText>
```

5. BACKING BEANS

- lier la valeur d'un composant à un attribut d'un backing bean
- attacher un composant à un attribut d'un backing bean
- déclaration dans le ficher de configuration de JSF
- lien avec des composantes d'affaires (EJB3, Spring)
- niveau de portée: application, session ou requête.

```
<h:commandButton action= =" #{monBean.methodXXX}"
```

6. EVENTS & LISTENERS

Types:
- value change events
- action events
- data model events
- phase events

Les listeners sont appelés à la soumission de la page.
Ils peuvent se présenter sous forme de classe ou de méthode d'un backing bean.

```
<h:inputText value=" #{user.name}"  valueChangeListener=" #{monBean.reaction}" />
```

7. MESSAGES

- erreurs d'applications ou de validation
- composants pouvant ajouter des messages
- fonctionnent comme les tags Struts **<html:errors />**

Au niveau du backing bean

```
FacesContect context = FacesContext.getCurrentInstance;
 context.addMessage (null, new FacesMessage ("mon message");
```

Et au niveau de la vue

```
<h:messages style =" color: blue"
```

8. NAVIGATION

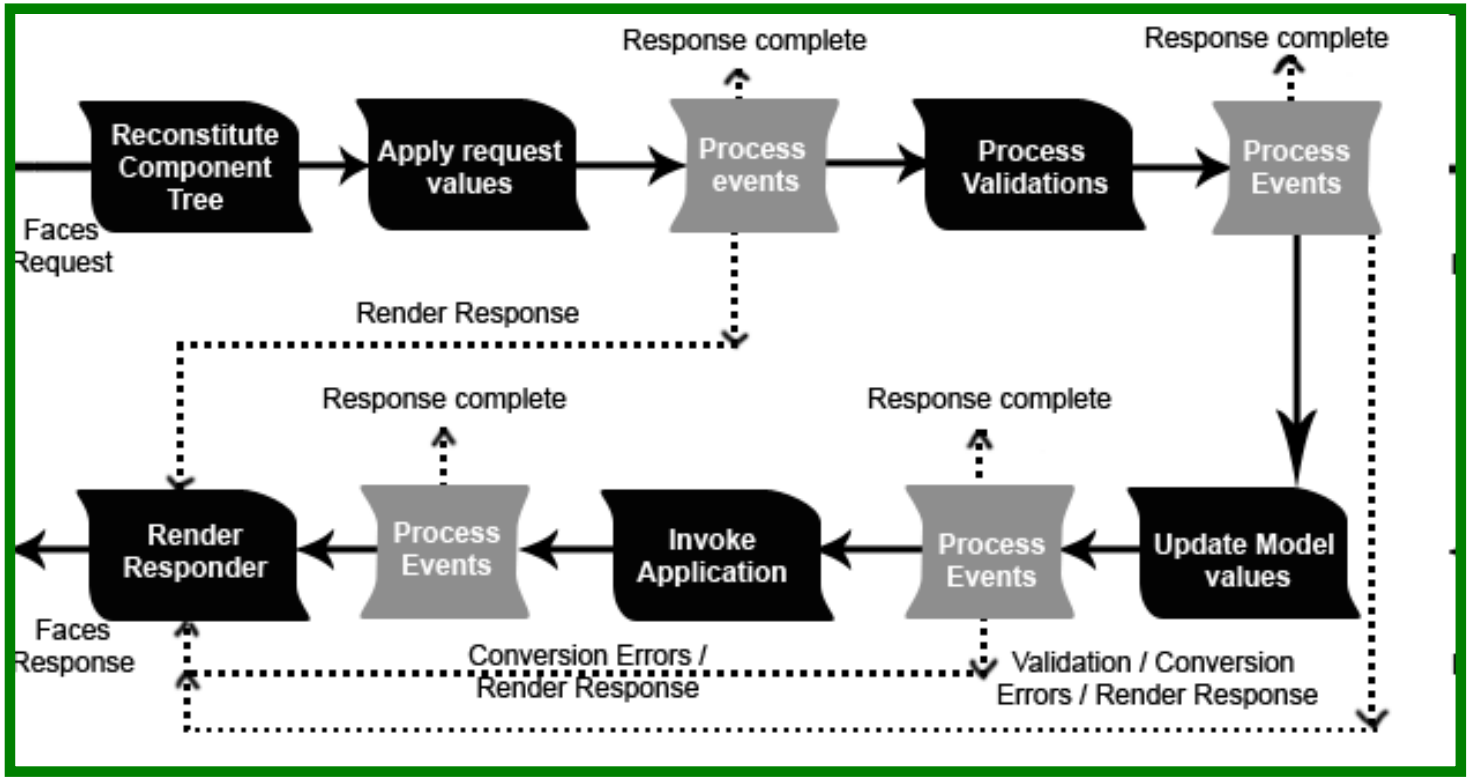
- Déclaration des règles de navigation dans le fichier de configuration de JSF ***faces-config.xml*** .
- basé sur des **outcome** qui sont des chaînes de caractères.

Par exemple dans faces-config.xml:

```
<navigation-rule>
<from-view-id>accueil.jsp</from-view-id>
            <navigation-case>
                                <from-outcome>succes </from-outcome>
                                <to-view-id>initial.jsp</to-view-id>
            <navigation-case>
<navigation-rule>
```

Cycle de vie JSF

Le cycle de vie d'une application web JSF commence lorsque l'utilisateur soumet une requête. Typiquement, il y a 6 étapes dans le cycle de vie d'une application JSF.



1. **Restore View**: construction ou mise à jour de l’arbre logique de composants
2. **Apply request Value**: insertion des valeurs dans chacun des constituants de l’arbre
3. **Process Validations**: JSF demande à chacun des composants de se convertir et de se valider
4. **Update Model Values**: insertion des valeurs dans le modèle (usage des backing bean d’autres objets du modèle)
5. **Invoke Application**: exécute la méthode action (ou un autre listener enregistré) appelée par l ’événement par défaut déclenché par les composants de type command, lien ou bouton
6. **Render response**: affiche la réponse en mettant à contribution les vues associées (JSF, Facelets).

Ces différentes étapes sont transparentes lors d'une utilisation standard de JSF.

Premier exemple JSF

Nous utilisons JSF sous la référence de SUN.

Installation de JSF.

Au moment où ce cours est écrit ,j'utilise la dernière version de **JSF 1.2_09** qu'il faut télécharger à l'adresse: **https://javaserverfaces.dev.java.net/**

Placer les fichiers: jsf-api.jar et jst-impl.jar dans WEB-INF/lib de votre WDP.

Il faut aussi télécharger les librairies JSTL (jakarta taglib) à l'adresse
http://people.apache.org/builds/jakarta-taglibs/nightly/

Placer aussi les fichiers: jstl.jar et standard.jar dans WEB-INF/lib.

Créer un projet WDP et ajoutez-y les deux pages JSPs suivantes:

Nous écrivons deux pages JSP ***accueil.jsp*** et une page de bienvenue ***bienvenue.jsp***.

```
(…)
<html>
    <head>
        <title>Insert title here</title>
    </head>
  <body>
        <jsp:forward page="bienvenue.jsp"/>
  </body>
</html>
```

accueil.jsp

bienvenue.jsp

```
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>

<html>
<head>
<title>Insert title here</title>
</head>
<body>
 <f:view>
    <h:outputText value="Felicitations ! votre premier exemple JSF 1.2"/>
</f:view>
</body>
</html>
```

Un projet avec JSF nécessite la création d'un fichier ***faces-config.xml*** de configuration de votre application. Dans cet exemple le ficher sera créé mais ne contient aucune instruction de configuration. Nous verrons dans les exemples suivantes comment on peut l'utiliser.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.1//EN"
    "http://java.sun.com/dtd/web-facesconfig_1_1.dtd">

<faces-config>

</faces-config>
```

faces-config.xml

Il faut maintenant déclarer ce ficher dans votre descripteur de déploiement ***web.xml*** et configurer JSF pour qu'il fonctionne correctement. Voici comment .

```
(….)
</web-app>
(…)
<!-- Configuration de JSF -->


<!-- Faces Servlet -->
   <servlet>
      <servlet-name>FacesServlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <!-- Faces Servlet Mapping -->
   <servlet-mapping>
         <servlet-name>FacesServlet</servlet-name>
         <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
(….)
</web-app>
```

web.xml

Second exemple JSF avec Managed Bean (backing bean)

Les ***beans managés*** (ou *backing bean*) sont des javabeans gérés par JSF .
Ils permettent de récupérer les données des formulaires utilisateurs.
Ils sont déclarés dans ***faces-config.xml*** avec un tag <**managed-bean**>.
Ce tag possède trois tags fils obligatoires :
<managed-bean-name> **:** le nom attribué au bean utilisé lors de son utilisation
<managed-bean-class> **:** le type pleinement qualifié de la classe du bean
<managed-bean-scope> **:** portée dans laquelle le bean sera stocké et donc utilisable

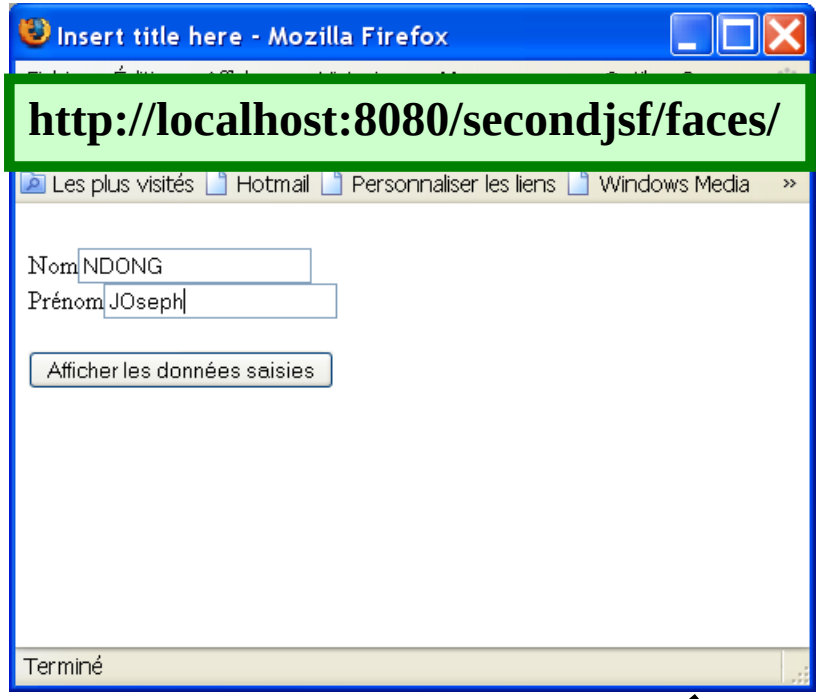
La portée peut prendre les valeurs suivantes :

request : portée limitée entre l'émission de la requête et l'envoi de la réponse.
Les données stockées dans cette portée sont utilisables lors d'un transfert vers une autre page (forward). Elles sont perdues lors d'une redirection (redirect).

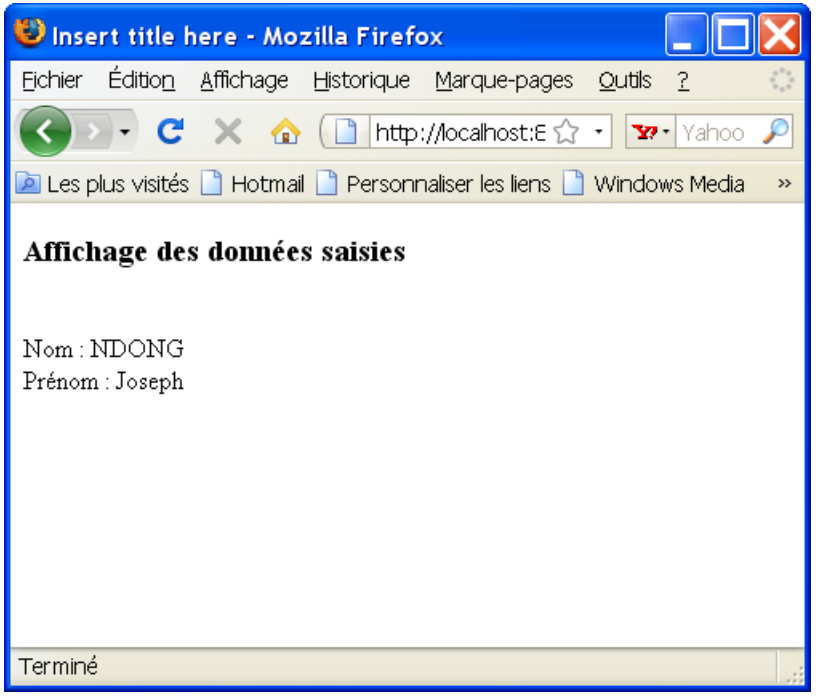
session : cette portée permet l'échange de données entre plusieurs échanges avec un même client.

application : cette portée permet l'accès à des données pour toutes les pages d'une même application quelque soit l'utilisateur.

Il s'agit de saisir un utilisateur dans un formulaire et d'afficher les données concernant cet utilisateur dans une nouvelle page. Les données saisies sont récupérées dans un bean managé.



debut.jsp



fin.jsp

Configuration de *faces-config.xml*

```
(…)
<faces-config>
<navigation-rule>
        <!-- Indique pour quelle vue courante la règle s applique -->
    <from-view-id>/debut.jsp</from-view-id>
<navigation-case>
        <!-- Si l outcome renvoyé est verspage alors JSF passe a la page /fin.jsp -->
    <from-outcome>verspage</from-outcome>
    <to-view-id>/fin.jsp</to-view-id>
</navigation-case>
</navigation-rule>
 <!– configuration du backing bean -->

<managed-bean>
                <managed-bean-name>personne</managed-bean-name>
                <managed-bean-class>joe.cours.jsf.Personne</managed-bean-class>
                <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

</faces-config>
```

(….)

```
<body>
<f:view>
<h:form>
    <BR>
        <h:outputText value= "Nom" />
        <h:inputText value="#{personne.nom}" />
    <BR>
        <h:outputText value= "Prénom" />
        <h:inputText value="#{personne.prenom}" />
    <BR>
    <BR>
        <h:commandButton value="Afficher les données saisies" action="verspage"/>
</h:form>
</f:view>
</body>
</html>
```

Renseignements du bean managé.

debut.jsp

Défini dans <***navigation-case***> du fichier ***faces-config.xml.***

```
(….)
<body>
<f:view>
<h3>Affichage des données saisies</h3>
<BR>
<h:outputText value= "Nom : #{personne.nom}" />
<BR>
<h:outputText value= "Prénom : #{personne.prenom}"/>
<BR>
</f:view>
</body>
(….)
```

Accès au backing bean pour lire les données.

fin.jsp

Troisième exemple JSF avec Properties Files

Comme on l'a déjà vu antérieurement, on peut utiliser des fichiers de propriétés (PROPERTIES) pour stocker certains messages qui seront utiliser par l'application.

Rappelez vous que les fichiers de propriétés doivent être stockés dans le répertoire des sources Java.

Voici un exemple de fichier *properties* :

```
monNom      = Entrer votre nom
monPrenom   = Entrer votre prénom
Leprenom    = Prenom
Lenom       = Nom
```

messages.properties

Nous reprenons l'exemple précédent.
Voici les modifications à apporter au niveau des pages JSPs.

Dans les fichiers ***debut.jsp*** et ***fin.jsp*** placer la ligne suivante (elle permet de charger un fichier de ressources (resource bundle) *sous forme d'un objet Map* dans la *portée request*):

```
<f:loadBundle basename="messages" var="msg"/>
```

après les deux lignes ci dessous:

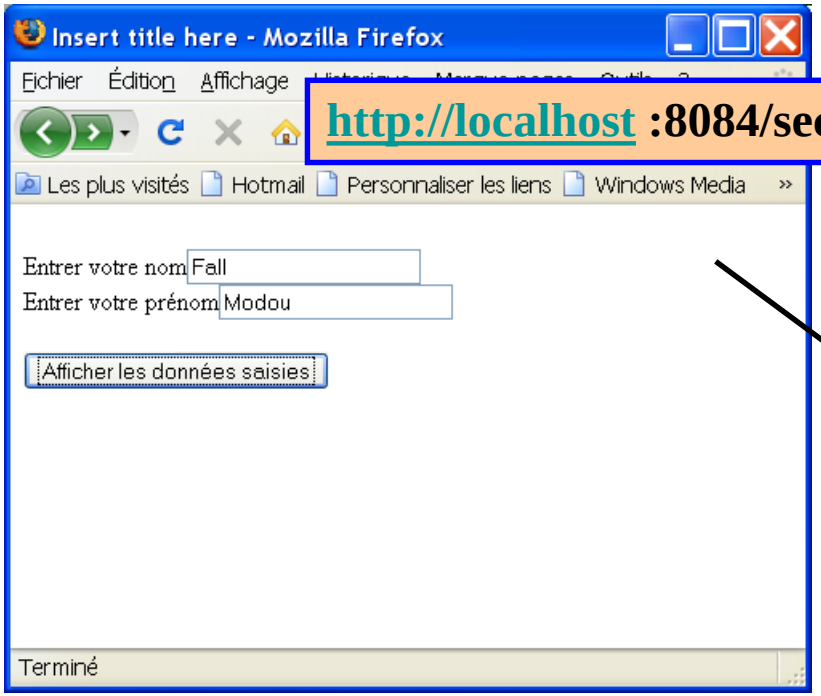
```
<%@ taglib prefix="f"  uri="http://java.sun.com/jsf/core"%>
<%@ taglib prefix="h"  uri="http://java.sun.com/jsf/html"%>
```

```
<f:view>
(…)
        <h:outputText value = "#{msg.monNom}" />
        <h:inputText value   ="#{personne.nom}" />

        <h:outputText value = "#{msg.monPrenom}" />
        <h:inputText value   ="#{personne.prenom}" />
(…)
</f:view>
```

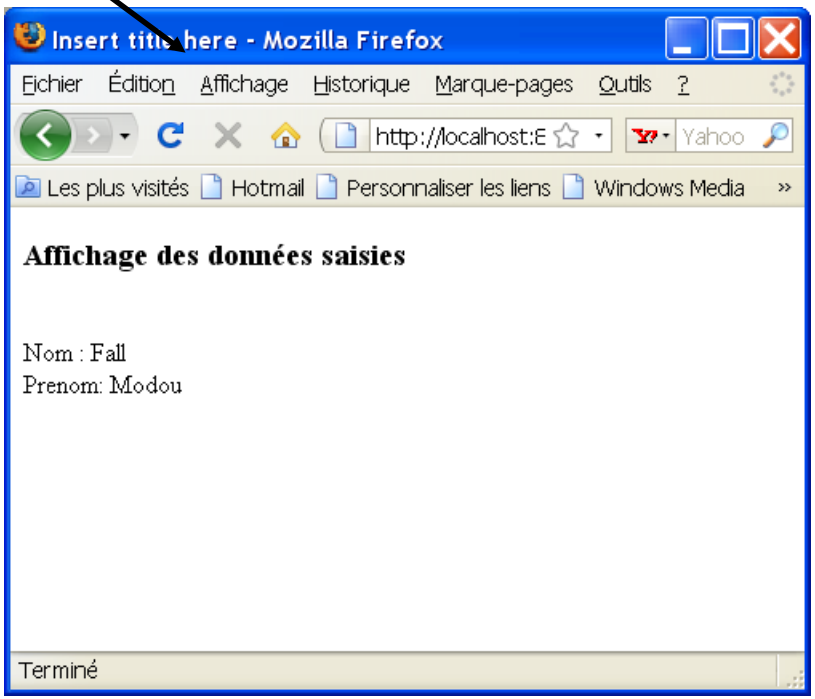
Modifications dans *debut.jsp*

```
<h:outputText value = "#{msg.lenom} : #{personne.nom}" />

<h:outputText value = "#{msg.leprenom}: #{personne.prenom}"/>
```

Modifications dans *fin.jsp*



http://localhost :8084/secondjsf /faces/ debut.jsp

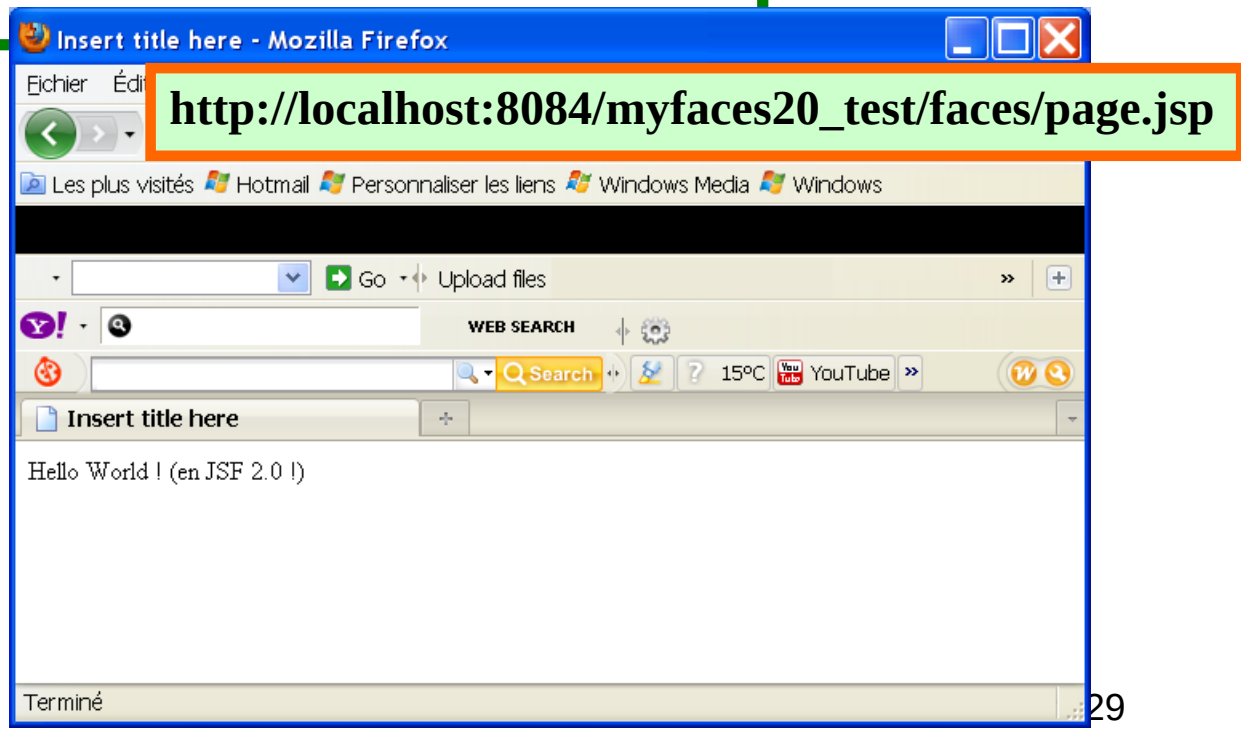


MYFACES

Je montre comment utiliser la distribution de MyFaces 2.0 pour construire des JSF.

1) **Télécharger myfaces-core-2.0.3-bin.jar**
2) **Décompressez le jar dans un répertoire de votre choix**
3) **Créer un DWP et copier les jars du /lib de myfaces dans le /lib de votre projet**
4) **Copier y aussi le fichier jstl.jar (comme pour l'exemple avec SUN).**

Les fichiers web.xml et faces-config.xml sont configurés exactement de la même manière que dans l'exemple utilisant la distribution de SUN.

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>

.............
<body>
<f:view>
        <h:outputText value="Hello World ! (en JSF 2.0 !)" />
</f:view>
</body>
................
```

page.jsp