



Global Knowledge®

GKJSPWEB

© GK 2018

Présentation

Qui suis-je ?



Présentation

Qui êtes-vous ?



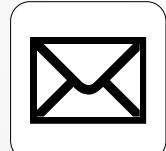
Logistique



Pause en milieu de session



**Vos questions sont les bienvenues.
N'hésitez pas !**



Feuille d'évaluation à remettre remplie en fin de session



Merci d'éteindre vos téléphones

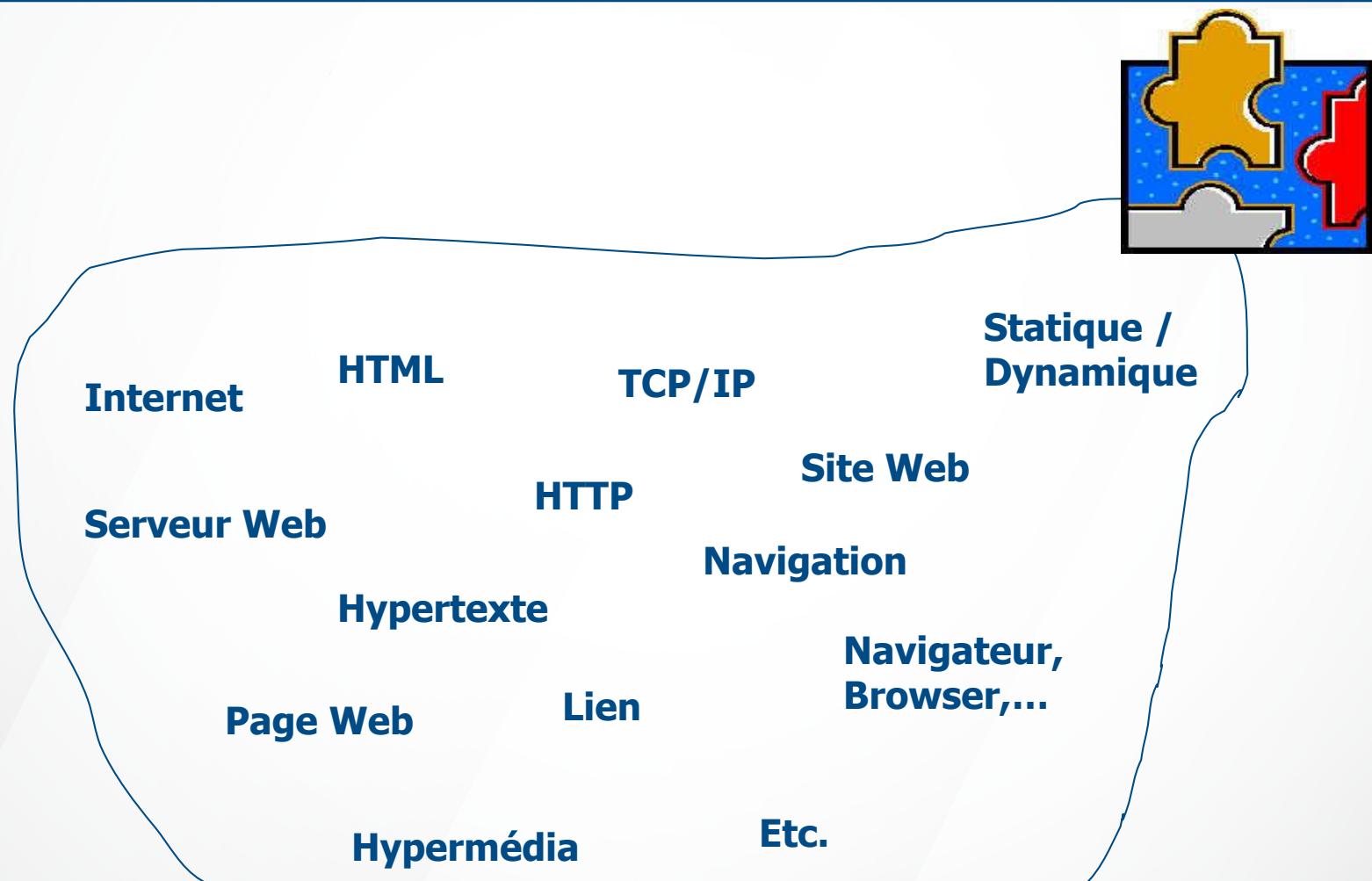
Sommaire

- **Chapitre 1 : INTRODUCTION**
 - Chapitre 2 : PLATE-FORME JAVA EE
 - Chapitre 3 : SERVLET / JSP
 - Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE
 - Chapitre 5 : JSF
 - Chapitre 6 : PRIMEFACES
-
-

INTRODUCTION



Assembler le puzzle



Historique

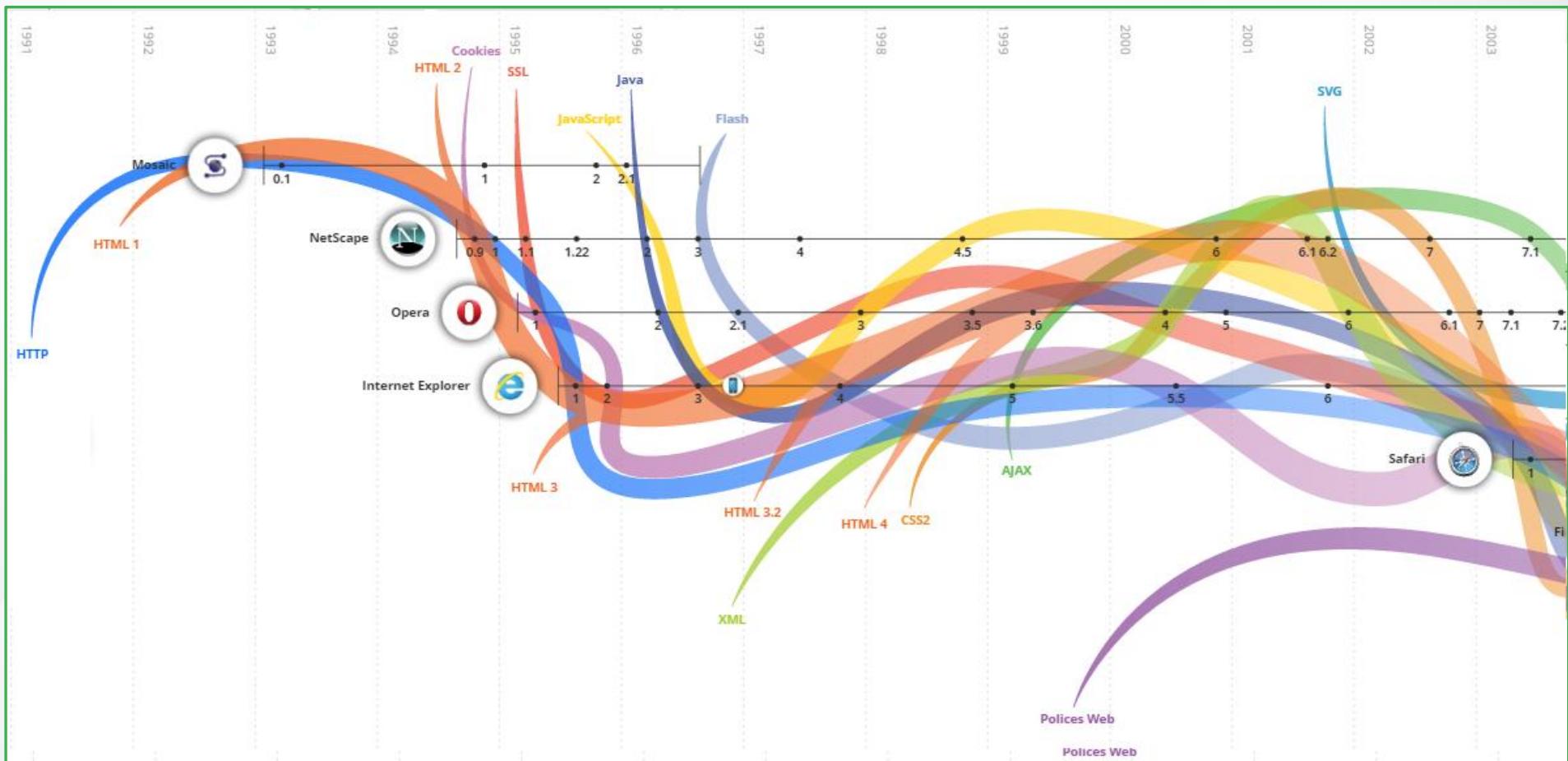
- 1945 - Vannevar Bush. « As we may think »
 - conception de Memex (système de gestion et d'accès aux connaissances)
 - Idée de liens entre les documents
- 1965 - Ted Nelson
 - Invention du terme « hypertext »
 - Projet Xanadu : bibliothèque universelle
 - circulation en utilisant des liens
- 1989-1990 Conception du World Wide Web et HTML
 - CERN (Tim Berners-Lee et Robert Cailliau)
 - Rendre accessible des documents scientifiques aux ordinateurs situés à différents endroits sur la planète
- Novembre 1990 – premier prototype développé au CERN

Historique

- Janvier 1992 – premier navigateur disponible
 - que du texte
 - code HTTP rendu publique
- Mars 1993 - Mosaic
 - Marc Andreessen et Eric Bina (NCSA National Center for Supercomputing Application)
 - premier navigateur Web graphique (texte et image) gratuit
- 1993 - Médiatisation (NY Times)
 - augmentations du nombre des intéressés
- 1994 – fondation de Netscape
 - Marc Andreessen et Jim Clark
 - Netscape Navigator
- ...

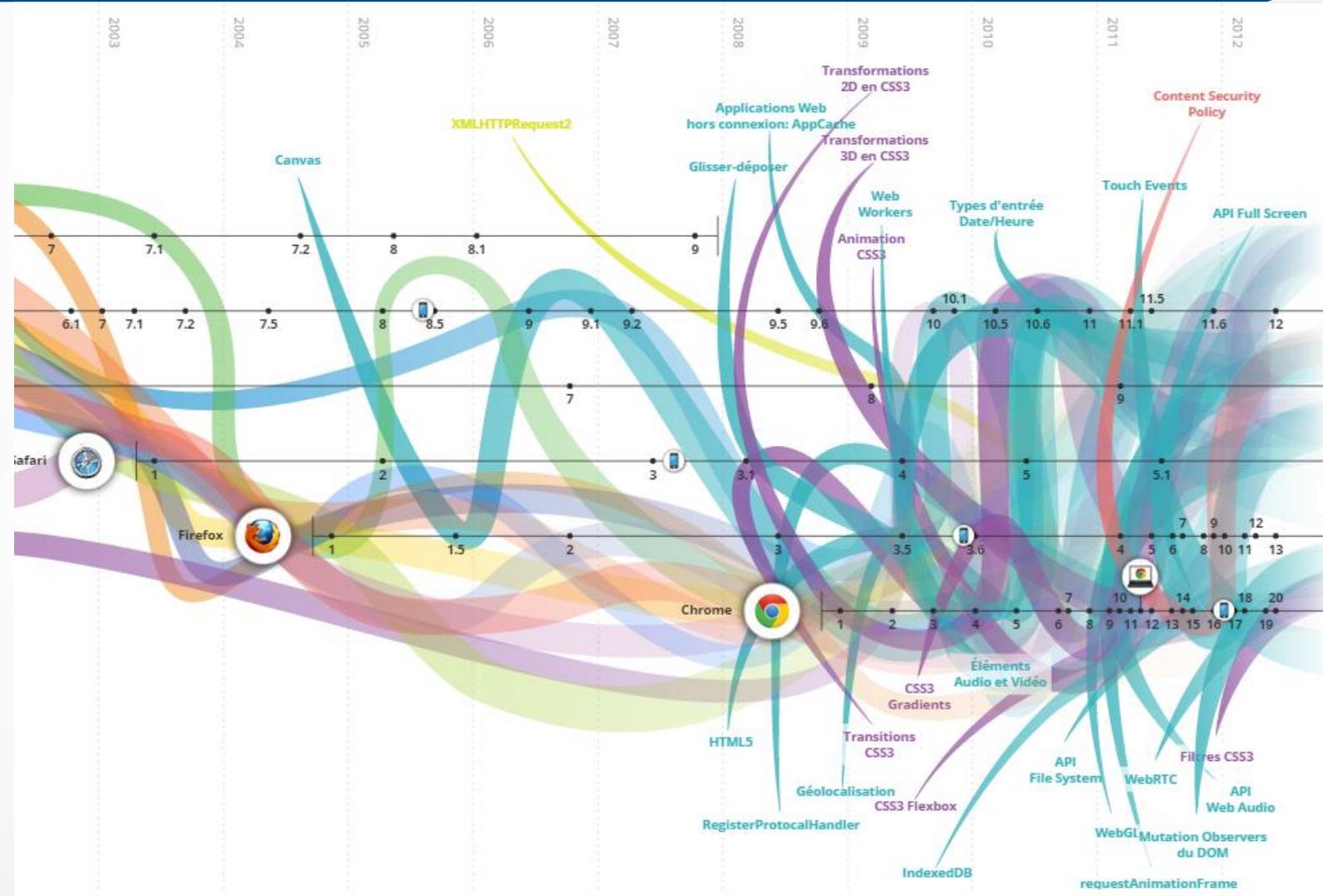
Evolution du web

➤ 1991 → 2003



Evolution du web (2)

➤ 2003 → ...

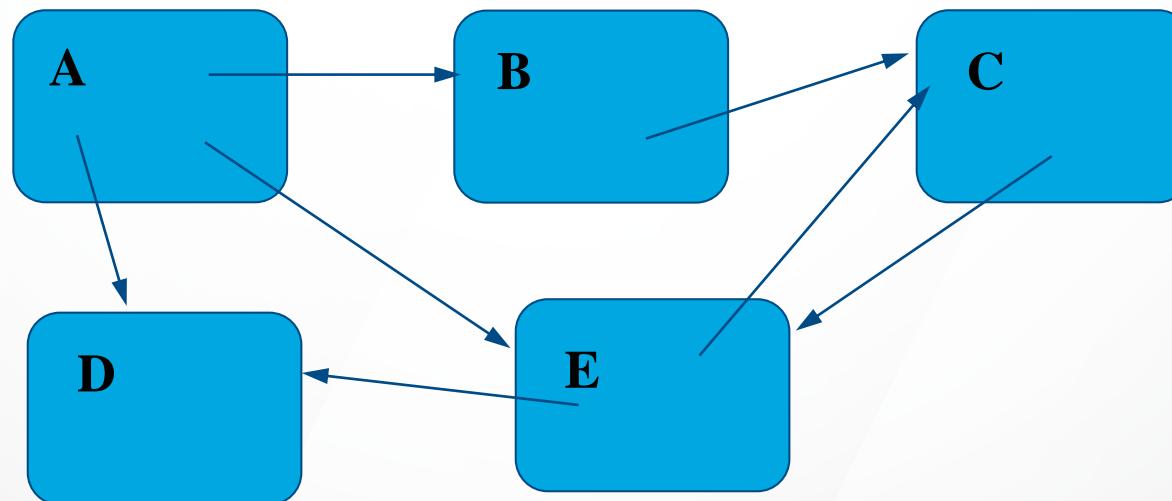


Définir le Web!

- Plusieurs niveaux :
 - Un système Hypertexte/hypermédia
 - Service d'Internet (se positionner % à Internet)
 - Architecture / Fonctionnement

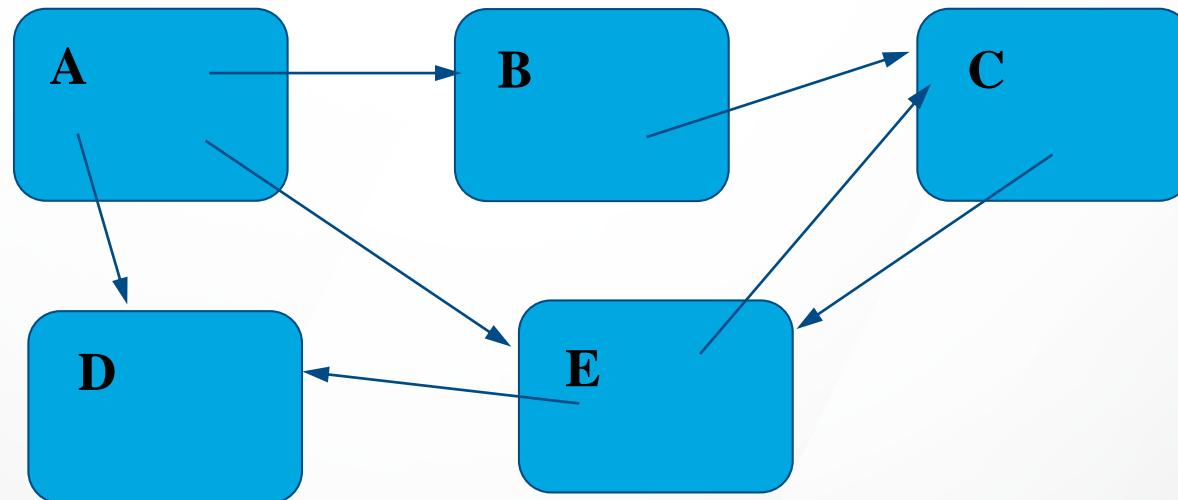
Le Web est un système Hypertexte/Hypermédia

- Document textuel "plat" ~ ordre de lecture séquentiel: page 1, puis page 2, puis ...
- Un hypertexte ~plusieurs ordres de lecture
 - ex. pour aller de A à D
 - A B C E D
 - A D



L'hypertexte est un graphe

- **Nœuds → pages individuelles**
- **Arcs → liens entre les pages**
- Accès par navigation (ou "browsing")
- Un nœud peut pointer plusieurs liens
- Un nœud peut être pointé par plusieurs liens



Hypertexte/Hypermédia

- Hypermédia : Hypertexte multimédia
 - Texte
 - Image
 - Audio
 - Vidéo
- On emploie en général le terme "hypertexte" dans le sens "hypermédia"

Liens

- Un lien : une source et une cible
- La cible : document entier ou une section précise dans un document
- L'ancre : objet cliquable pour activer un lien
- Ancre dans les hypermédias
 - texte (caractère, mot, phrase, etc.)
 - image entière
 - zones dans une image
- Web : ancrès textuelles ~ généralement écrites en bleu et soulignées

Le Web est un service d'Internet

- D'abord c'est quoi Internet ?
- Qu'est ce qu'on entend par service ?
- Comment créer un service ?
- Un service peut-il disparaître ?
- Un service peut-il évoluer ?

Internet

- Câbles ;
- Routeurs ;
- ...
- Infrastructure matérielle (réseau) => permettre aux machines de s'échanger des données (communiquer)

- Quel langage ?

Le réseau Internet

Le protocole TCP/IP

- TCP/IP
 - La “langue“ du réseau Internet
 - Permet le dialogue entre les machines
- Faire fonctionner le TCP/IP --> besoin d'une infrastructure matérielle : le réseau Internet

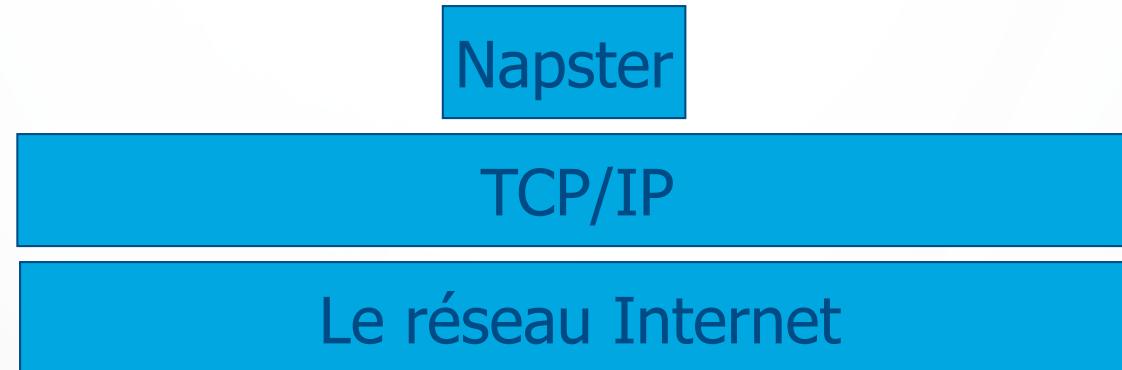


À quoi ça peut servir ?

- Satisfaire nos besoins d'information, de communication, etc.
- Exemples de besoins
 - Échange de messages (e-mail)
 - Publication de données
 - Échange de fichier
 - Partage de ressources musicales
 - ...

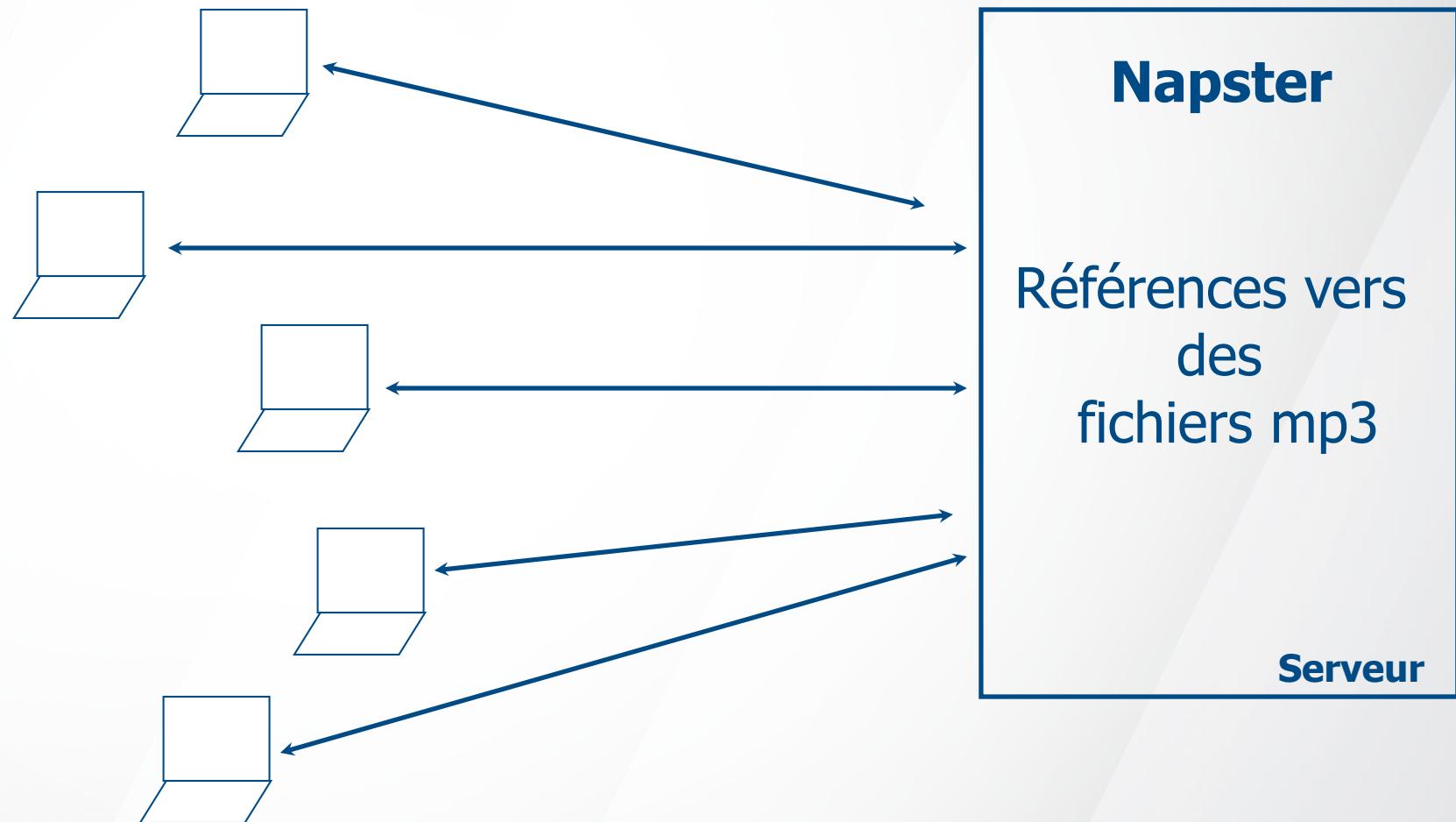
Création de service

- Pour chaque besoin, on peut créer un service
- Exemple
 - Partage de la musique
 - Service créé : Napster



Napster

- Architecture de type "clien-serveur"

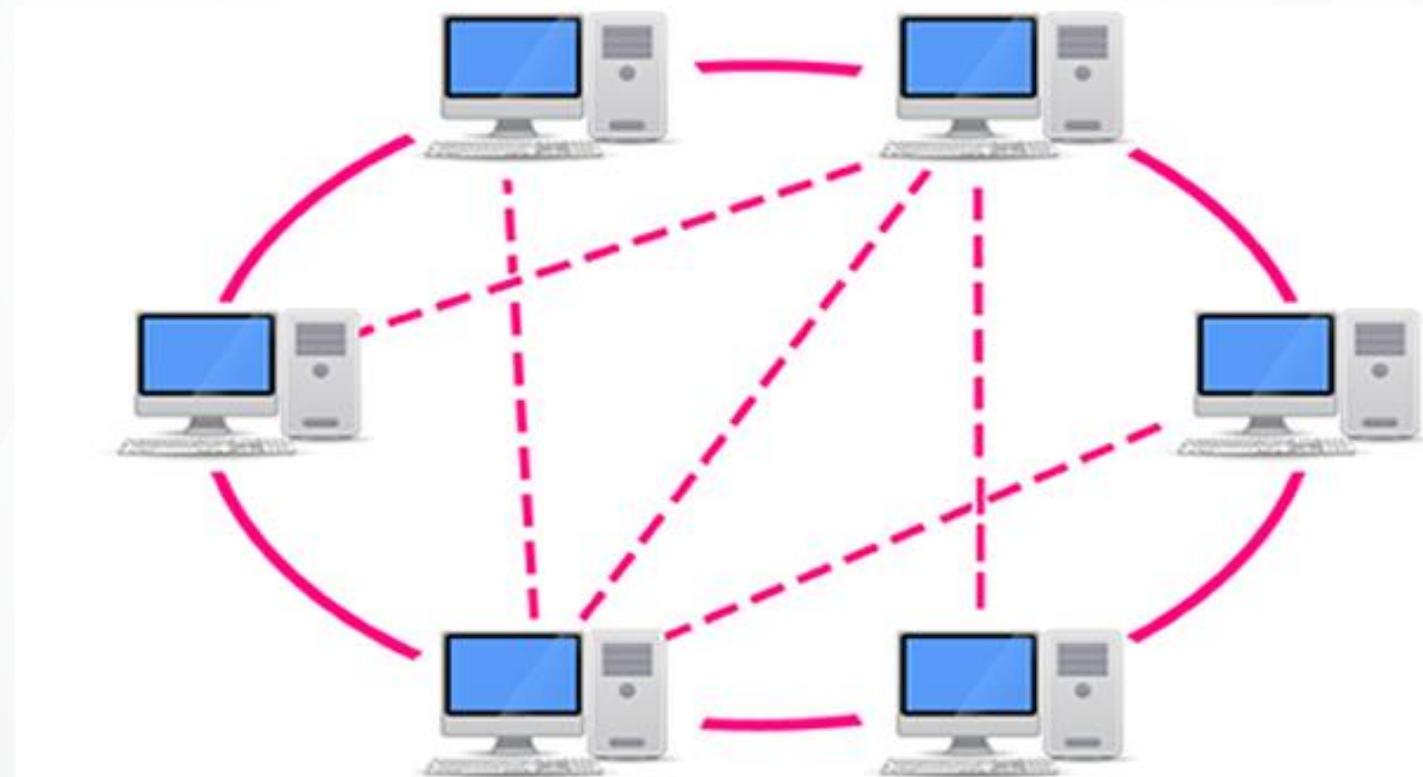


Napster

- Problèmes juridiques
- Fermeture de Napster
- On a toujours un besoin!
- Créer un nouveau service

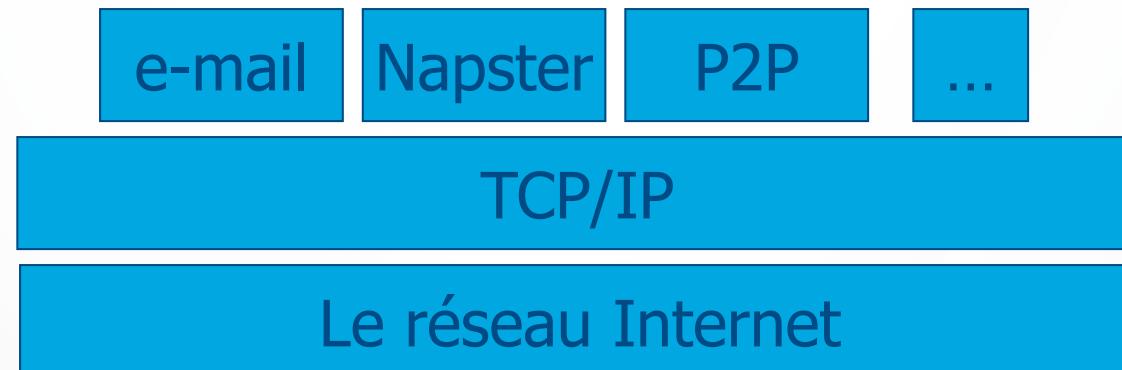
Peer-to-Peer (P2P)

- Architecture de type "client-client"



Pourquoi tout ce tour ?

- Pour chaque besoin, on peut créer un service
- Internet est flexible et "modulable"



Quel protocole ?

- Le protocole HTTP (HyperText Transfert Protocol)
- Protocole de communication de type “client-serveur”:
 - Client : navigateur (Chrome, Internet Explorer, Firefox)
 - Serveur Web : logiciel tels que IIS ou Apache
- Permet de transférer n’importe quel type de fichier (HTML, images, etc.)

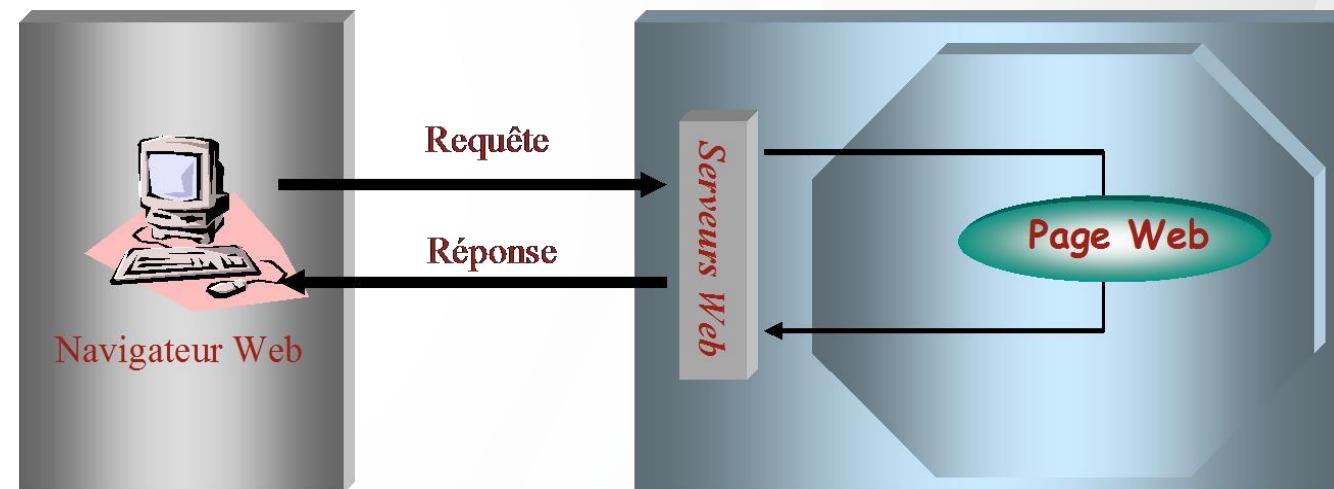
Fonctionnement du protocole HTTP

➤ Requête : demande d'une ressource (un document)

- comment identifier cette ressource ?
- comment récupérer cette ressource ?
- à quelle adresse l'envoyer ?
- ...

➤ Réponse : envoyer une ressource

- analyser la requête ?
- comment "fabriquer" la réponse ?
- comment présenter une ressource ? (format des documents)
- ...



Requête HTTP

- Navigateur (Internet Explorer, Chrome, Mozilla, etc.)
 - logiciel → interface entre le client et la complexité technique du protocole HTTP
- Requête HTTP
 - Nom du fichier : <http://unige.ch/.../exemple.html>
 - Adresse IP du client
 - Modèle du navigateur
 - ...

Identification des ressources

- Identification unique
- Une **URL** (*Uniform Resource Locator*) est un format universel pour identifier une ressource (sur Internet ou en local)
- L'URL est un moyen pour communiquer sur Internet

URL - Uniform Resource Locator

protocole://id:mp@serveur:port/fichier[#ancre]?paramètres]

protocole : quel “langage“ on va utiliser ?
 transfert de fichiers entre machines
 échanger des pages HTML

paramètres d'accès à un serveur sécurisé

nom du serveur : nom du domaine de la machine
contenant la ressource demandée
on peut utiliser l'adresse IP (URL moins lisible)

liste des paramètres
envoyés à une
application sur le
serveur

nom du pointeur dans le document HTML

Chemin d'accès à la ressource (emplacement
de la ressource)

numéro associé à un service
facultatif dans le cas du Web (80)

Exemples :

<ftp://radhouan@cui.unige.ch>

<http://cui.unige.ch/isi/reports/rfia2004.pdf>

Réponse d'un serveur Web

- Serveur Web :
 - Logiciel qui tourne en permanence et fournit des documents en réponse aux requêtes des clients
- Réponse-1 :
 - contrôle des droits d'accès
 - récupération des ressources
 - "fabriquer" le fichier demandé
 - mettre à jour le journal d'activité (log)

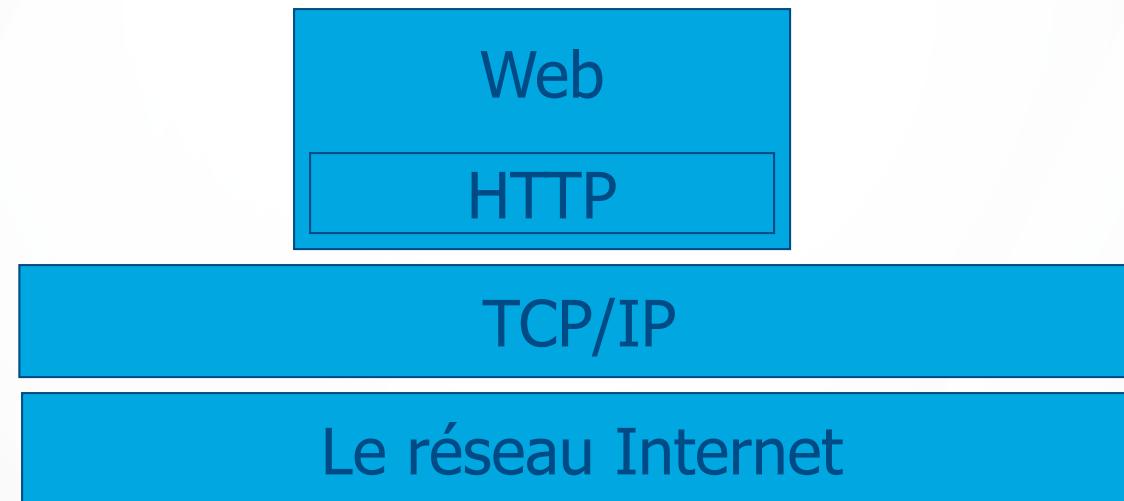
Réponse d'un serveur Web

➤ Réponse-2 :

- envoyer le type MIME (Multipurpose Internet Mail Extensions)
 - standard pour étendre les possibilités du courriel
 - insérer des documents dans un courrier (image, son, etc.)
 - typer les documents transférés par le protocole HTTP
 - pour décider du type à utiliser, le serveur examine l'extension du fichier
 - guide pour le navigateur : savoir de quelle manière afficher le document (ou bien utiliser un plugin).
 - **syntaxe** : Content-type:type_mime_principal/sous_type_mime
 - **exemple** : Content-type:image/gif
- envoyer le fichier à l'utilisateur

Architecture du Web

- Le protocole HTTP est le fondement du Web



Navigateur

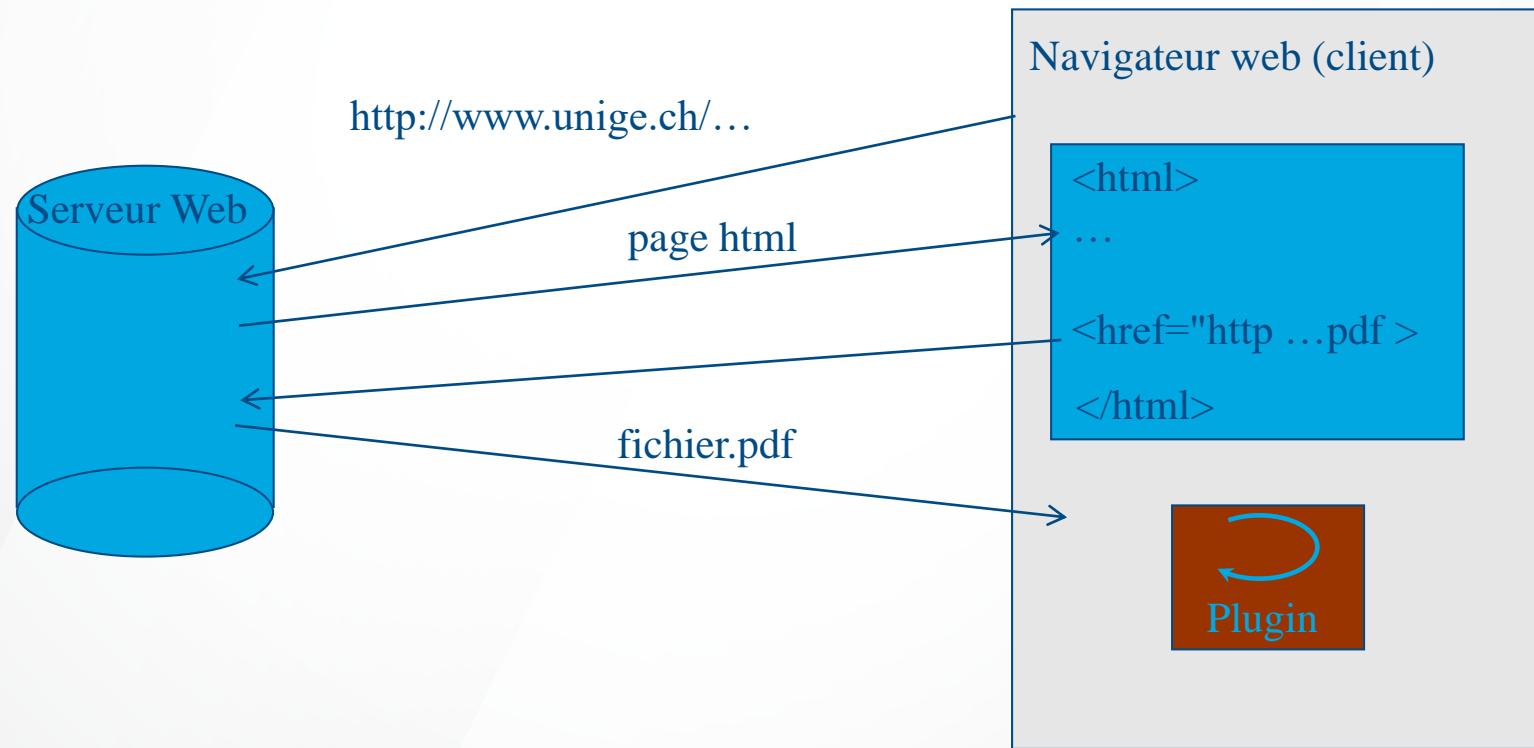
- Le navigateur est un logiciel capable d'afficher un certain nombre de fichiers
- Interprète les données envoyées par le serveur (document texte au format HTML)
- Crée la page à afficher
- Une même page Web peut être affichée de différentes manières (selon l'ordinateur ou le navigateur)
- Démo Responsive Design

Navigateur

- Si le type du fichier n'est pas reconnu du navigateur ?
 - le sauvegarder sur le disque et l'ouvrir avec l'application adéquate
 - utiliser un plugin
- Plugin : programme "accessoire" appelé automatiquement par le navigateur
- Exemples
 - Acrobat Reader documents pdf
 - RealPlayer son ou vidéo

Navigateurs et plugins

- Envoi d'une page html + lien vers un doc (ex. pdf)
- Exécution d'un programme sur le client



HTML

- HTML = HyperText Markup Language
- Langage utilisé pour créer des pages Web
- Fichier HTML = fichier texte dans lequel sont insérées des marques (balises) pour formater le texte
- Le code HTML décrit ce qui doit être affiché par le navigateur
- Il contient des indications sur :
 - la succession des éléments
 - la police du texte (taille et couleur de chaque portion de texte, ce qui doit être mis en italique, en gras ou souligné, etc.)

HTML

- Le langage HTML permet de définir des liens vers d'autres documents en utilisant leurs URLs
- Le navigateur analyse tous les éléments du code HTML et crée une représentation globale de la page
- Pour voir un exemple de code HTML: navigateur -> menu Affichage -> code source de la page

Structure d'une page HTML

- Entête (`<head>`) : information qui n'apparaissent pas à l'écran (sauf le titre, dans la barre de la fenêtre)
 - meta-information
- `<h1>...<h6>` pour structurer le document (titres et sous-titres)

Bienvenue au cours "**Nouvelles technologies du Web**!"

Bienvenue au cours "Nouvelles technologies du Web"!

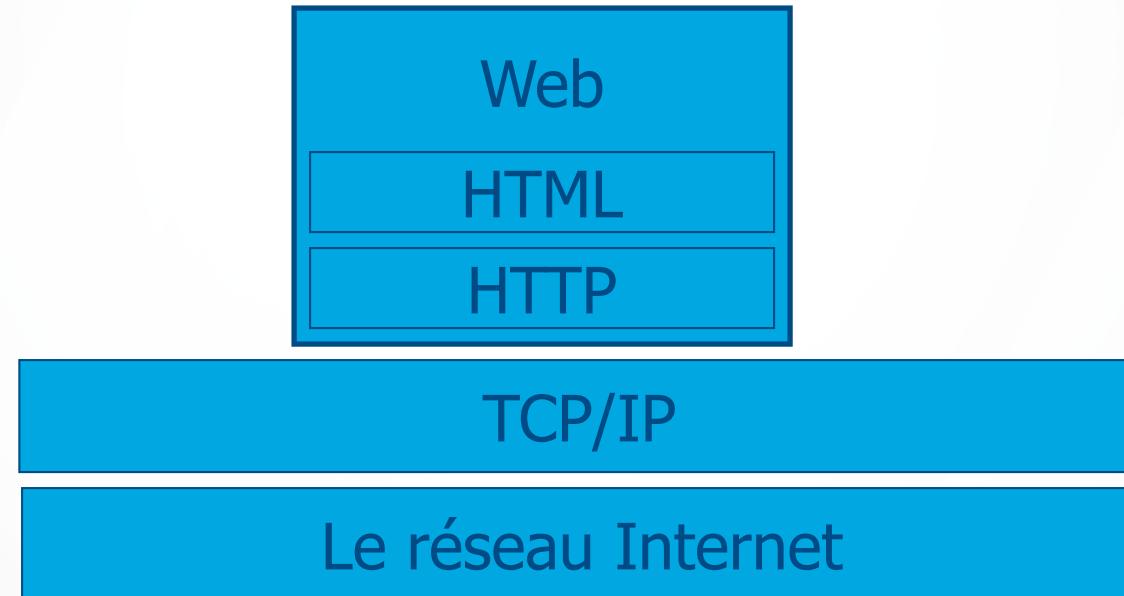
```
<!DOCTYPE html>
<html>
  <head>
    <title>Première page HTML</title>
    <meta charset= "utf-8">
  </head>

  <body>
    
    <h1>Mon titre principal</h1>
    <h2>Je suis un titre important</h2>

    
    <p>Voici
    mon
    premier
    paragraphe.</p>
    <p>Et en voilà un second !</p>
  </body>
</html>
```

Architecture du Web

- Le protocole HTTP et le langage HTML se complètent pour donner naissance au service Web.



Le langage HTML et les formulaires

- Introduit en HTML 2.0, les formulaires, les forms HTML, permettent d'interagir avec l'utilisateur en lui offrant la possibilité de saisir des données
- **Créer un formulaire**

```
<form method="post" action="traitement.php">
  <p><input type="text" name="pseudo" /></p>
</form>
```

On peut spécifier trois attributs dans la balise **<FORM>** :

- La méthode HTTP devant être transmise.
- L'action devant être déclenchée lors du "submit".
- Eventuellement le type d'encodage utilisé pour soumettre la requête

Le langage HTML et les formulaires

➤ La balise <INPUT>

- Les éléments "input" peuvent être des champs de saisie, des checkboxes, des radio- boutons des zones de texte, etc.
- Un élément "input" sera constitué de plusieurs attributs dont principalement :
 - TYPE qui définit la nature de l'élément (textField, checkbox, radioButton,etc.)
 - NAME qui permet d'associer un nom à l'élément à des fins de manipulations dans les servlets par exemple
 - SIZE pour définir la taille de l'élément

```
<form method="post" action="traitement.php">
  <p><input type="text" name="pseudo" /></p>
</form>
```

Le Web : un service qui évolue

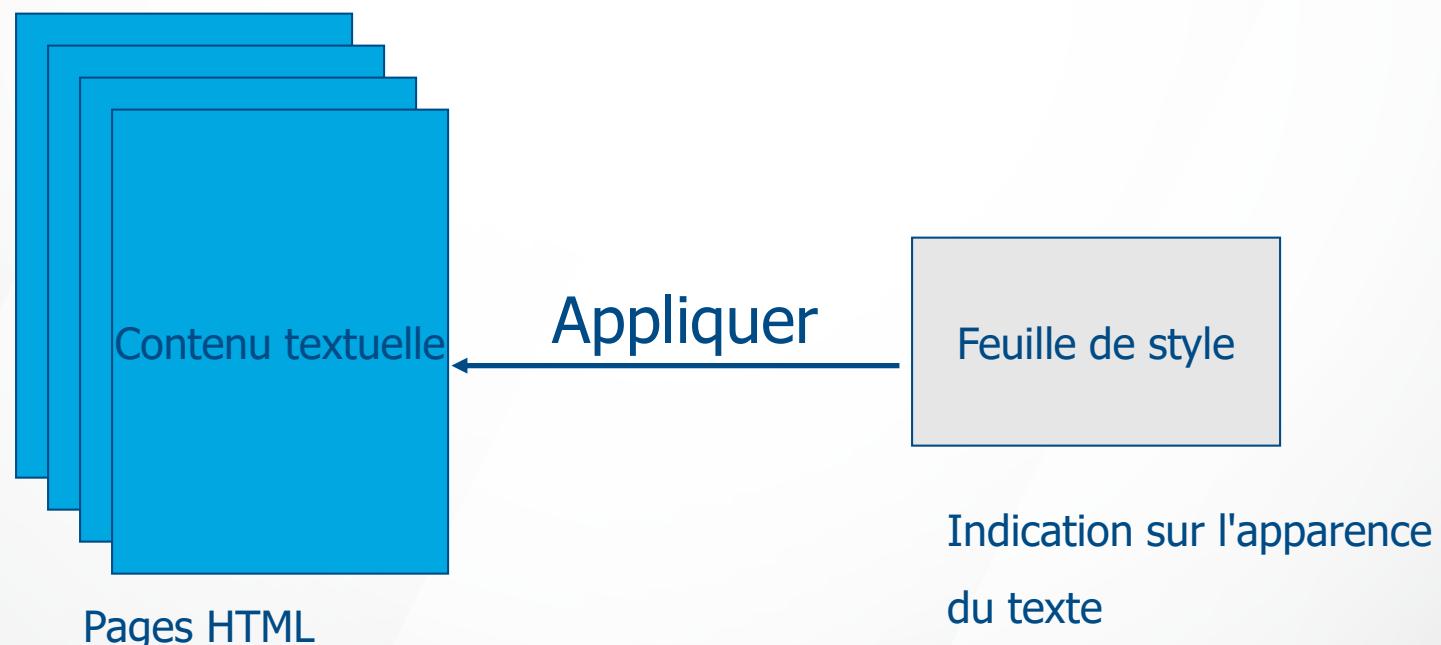
- Le service évolue en fonction des besoins
- Nouveaux Besoins :
 - entreprises commerciales
 - agents de publicité
 - ...
- HTML : prévu surtout pour la structure d'un texte, pas pour sa présentation
- → Ajout de nouvelles balises HTML :
 - cadres,
 - tableaux,
 - clignotement de texte,
 - etc.

Le Web : un service qui évolue

- Pour satisfaire les exigences des utilisateurs → Trop de codage
- Plus de besoins → plus de codage → plus de difficulté
- Solution : séparer le contenu de l'apparence

Les feuilles de style

- Le contenu est indépendant
- L'information concernant l'apparence est indépendante (stockée dans un doc.)



Feuilles de style

- Définir une présentation pour chaque type d'élément
 - exemple : tous les titres h1 doivent être en Arial, 20, rouge, centré
- Peuvent être intégrées :
 - dans une page HTML
 - dans un fichier séparé

Feuilles de styles - Exemple

```
<html>
...
<body>
  <h1>
    <font color=red><b>titre 1</b></font>
  </h1>
  <h2>
    <font color=blue><i>sous-titre</i></font>
  </h2>

  texte

  <h2><font color=blue><i> sous-titre</i></font></h2>
</body>
</html>
```



```
<html>
<head>
  <title>exemple style</title>
  <STYLE type="text/css">
    h1 {color: red; font: bold}
    h2 {color: blue; font: italic }
  </STYLE>
</head>

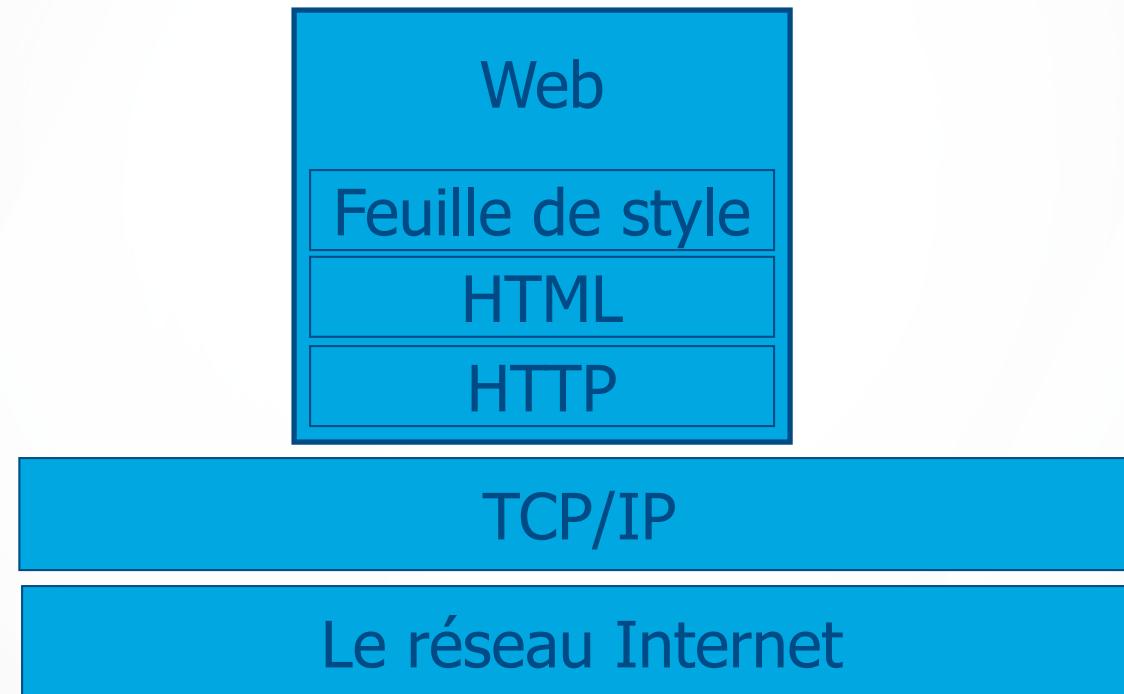
<body>
  <h1>titre 1</h1>
  <h2>sous-titre</h2>
  texte
  <h2>sous-titre</h2>
</body>
</html>
```

Utilité des feuilles de style

- Présentation homogène sur tout le site
- Entretien facile : la modification d'un style s'applique sur toutes les pages correspondantes
- Code HTML plus lisible (diminuer la taille des fichiers)
- Chargement de page plus rapides

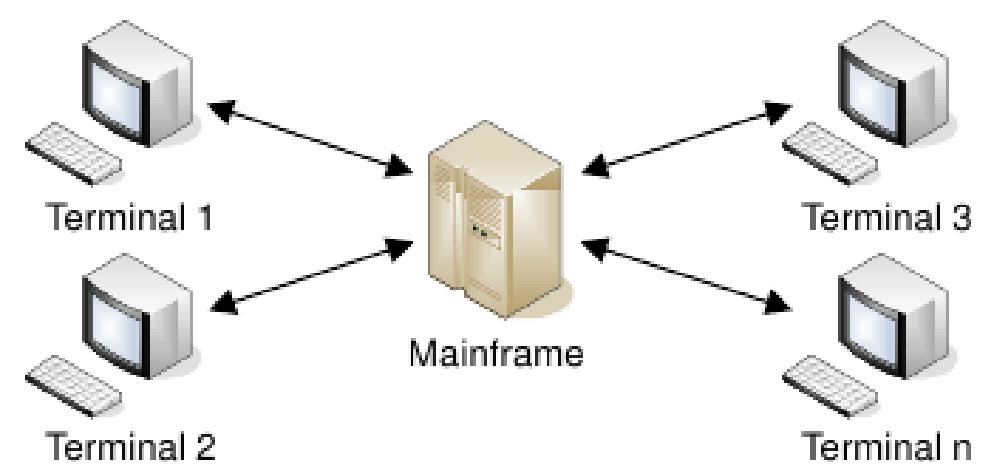
Architecture du Web

- Les feuilles de style : un moyen pour faciliter l'entretien des sites Web



Architecture : Historique

- Applications Monolithiques
 - Caractéristiques
 - Applications mêlant présentation, règles métier et les données
 - Ne communique pas avec l'extérieur
 - Terminaux passifs
 - Avantages
 - Performance
 - Sécurité
 - Inconvénients
 - Maintenance logicielle
 - Ouverture vers d'autres systèmes
 - Technologies réseaux propriétaires



Architecture : Historique (2)

➤ Applications à deux niveaux :

• Caractéristiques

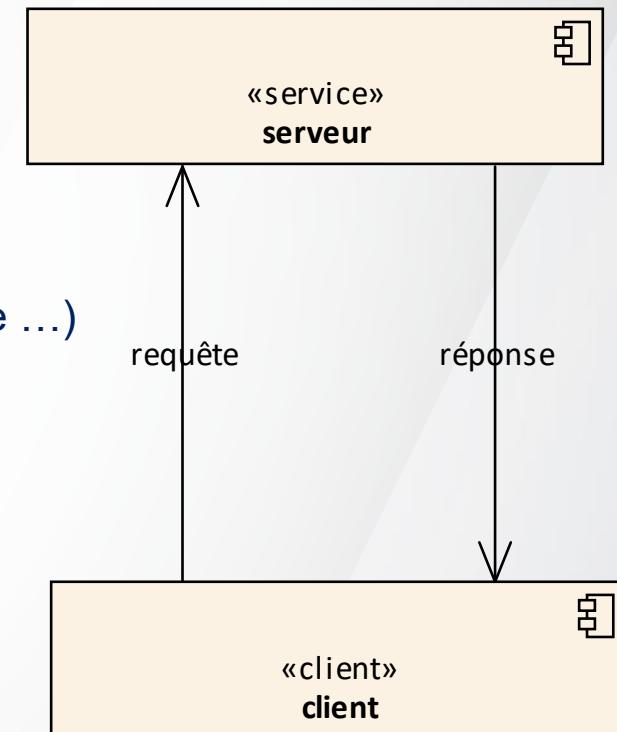
- Généralement appelées architecture client/serveur
- Un SGBD pour les données et une application pour l'IHM et le contrôle
- Seules les données transitent par le réseau
- Exemples de technologie :
 - Java/Swing avec SGBD MySQL
 - Site Web Statique : Serveur web (IIS / Apache), Client web (FireFox / Chrome ...)
 - Serveur FTP (ftpd) / Client FTP (FileZilla)
 - Serveur SMTP (Exchange)/ Client mail (Outlook)

• Avantages

- Réparties la puissance machine sur les clients
- Mise en œuvre du modèle de bases de données relationnelles
- Intégration inter-systèmes au niveau des données possibles

• Inconvénients

- Déploiement
- La logique métier est répartie sur les deux composantes
- Maintenance gestion des versions



Architecture : Historique (3)

➤ Architectures multi-tiers (couches)

- Caractéristiques
 - Au moins trois niveaux : IHM, les règles métiers et la persistance
 - Des normes de communication entre eux
 - Exemples de technologie : php/MySQL (3 niveaux au plus), Java EE (Java Entreprise Edition)



- Avantages
 - Maintenance
 - Nécessite peu de puissance client en cas de clients légers
- Inconvénients
 - Serveur puissant pour la logique métier

Différents types d'application : client ...

- Définition de «client»
 - Logiciel médiateur entre l'utilisateur et le service
 - Exemples : FTP, messagerie (mailer), navigateur (browser), webmail, jeux, ...
- Différentes catégories de client
 - **Lourd:** le service est disponible sur le poste client avec possibilité de connexion à des serveurs (appelée aussi application à architecture client/serveur)
 - Exemples : Yahoo Messenger, Word, Money, Battlefield 2, ...
 - **Léger:** tout le service est disponible sur des serveurs et l'utilisateur y accède par un conteneur spécialisé (appelée aussi application à architecture multi-tiers)
 - Exemples : Google, Yahoo Mail, ...

Différents types d'application : client lourd

- Technologies: Java/Swing, C#/.NET, C++/VC++, Delphy, JavaFx
- Avantages
 - Interfaces utilisateurs riches
- Inconvénients
 - Déploiement(utilisation de CD , téléchargement / installation)
 - Gestion des versions(patch, problème de compatibilité)

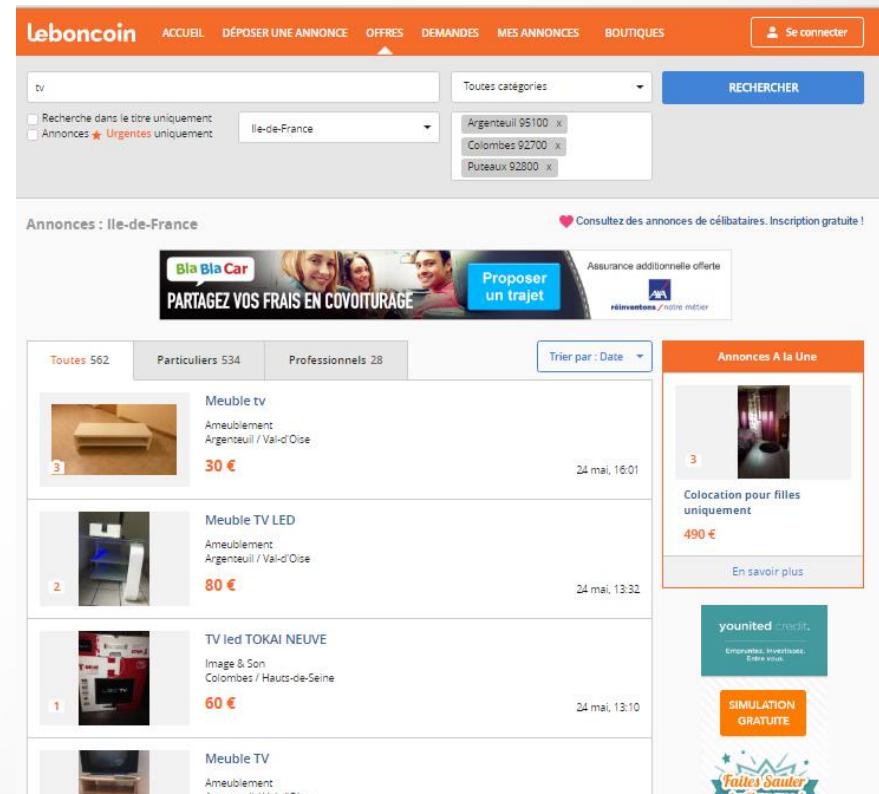


Interfaces riches (possibilité de dessiner dans un canvas, notion hiérarchique de composants, ...)

Interactions évoluées
(Drag&Drop, popup menu, ...)

Différents types d'application : client léger

- Architecture dite «multi-tiers» de trois à n-niveaux
- Les technologies pour la génération et le traitement de l'IHM sont présentes dans le client et dans le serveur
- Technologies côté client
 - HTML, DHTML, JavaScript, ...
- Technologies côté serveur
 - PHP, ASP, JSP, ...
- Avantages
 - Maintenance
 - Accessibilité
- Inconvénients
 - Interfaces utilisateurs pauvres et proches du classique formulaires



Serveur Web

- Sur internet (qui est le plus grand réseau informatique du monde), il y a aussi des serveurs qui jouent un rôle très important.
- Ce sont les **serveurs web**. Leur rôle principal est de stocker (**héberger**) tous les sites web et d'envoyer leur contenu à tous les ordinateurs (clients) qui veulent les visiter.
 - Bien sûr, ces ordinateurs doivent être connectés au réseau internet pour contacter le serveur.

Serveur Web (2)

- À chaque fois que tu visites un site web, ton ordinateur (client) doit contacter un ordinateur distant (serveur) qui lui envoie le contenu du site.

L'ordinateur client envoie au serveur l'URL (adresse) d'une page

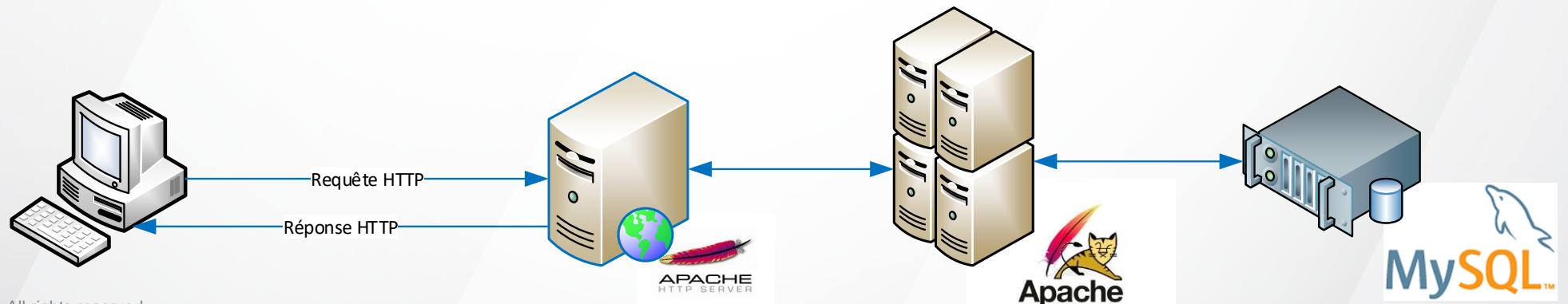
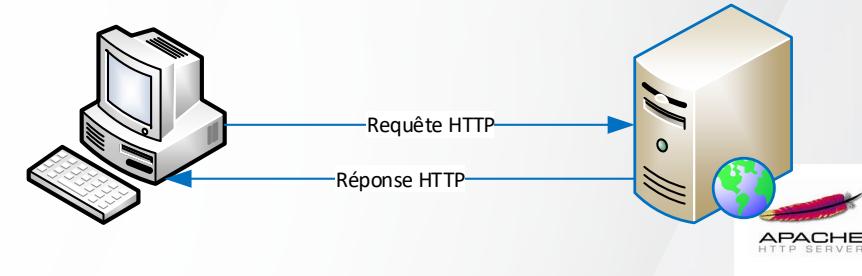


Serveur Web (2)

- Un serveur fonctionne de manière assez simple, mais pour comprendre ce qui va suivre, voici un petit rappel :
 - Le client = ton ordinateur
 - Le serveur = l'ordinateur qui héberge le site que tu visites
 - Site statique = site simple, écrit en HTML
 - Site dynamique = site complexe, programmé p. ex. en PHP, JAVA, .NET ...

Serveur Web (3)

- Un serveur fonctionne de deux manières différentes selon le langage utilisé :
 - Pour le site statique, le client va demander la page "**index.html**". Le serveur va lui envoyer la page directement.
 - Pour le site dynamique, le client va demander la page "**forum.jsp**", le serveur va d'abord devoir la générer (la créer) puis l'envoyer !



Client léger : technologies côté serveur

- Les technologies côté serveur permettent à l'aide de langages spécialisés de générer plus ou moins du code HTML
- Nous distinguons deux types de langages
 - Langages à balises : ceux qui sont utilisés dans le code HTML
 - PHP, ASP, JSP et .NET
- Langages de contrôle : ceux qui ne contiennent que du code propre au langage et qui généralement s'occupe du contrôle de l'application
 - CGI et Servlet
- Les langages de contrôles sont adaptés au traitement de fonctionnalités (sécurité, base de données, ...) et délèguent la partie présentation aux langages à balises
- Il n'est pas rare de trouver des applications uniquement avec des langages à balises mais la compréhension du code en devient alors difficile

Client léger : technologies côté serveur

- Deux types de langage, deux types de sémantique
 - Langages «procéduraux» : la portée des variables est limitée et l'absence de persistance oblige à bidouiller pour maintenir la valeur d'une variable pour chaque requête
 - ASP, PHP et CGI
 - Langages à objets : la persistance des objets permet de maintenir des états (valeurs d'attributs) à chaque requête
 - JSP, Servlet et .NET
- Exemple : un compteur
 - Dans le cas des langages procéduraux pour stocker la valeur d'un compteur on peut soit utiliser un simple fichier ou soit utiliser une base de donnée. A chaque nouvelle requête le compteur est initialisé au travers du support,
 - Dans le cas des langages à objets, un objet contenant un attribut compteur est créé à la première requête et sa durée de vie est fonction de différents paramètres (serveur, scope, ...)

Client léger : technologies côté client

- Les technologies côté client permettent d'effectuer des traitements supplémentaires que ceux fournis uniquement par l'HTML
- Deux types de technologie sont à distinguer
 - Affichage : celles qui s'occupent de la partie IHM
 - HTML et DHTML
 - Dynamique : celles qui effectue des traitements dynamiques
 - JavaScript, AJAX, Jquery
- Quelle que soit la technologie utilisée elles devront être codée et transmises par les technologies côté serveur
- Exemple : un formulaire
 - L'HTML ou le DHTML permettent d'afficher le formulaire
 - Le JavaScript permet de vérifier la cohérence «de surface» des données (champs vides, ...)

Client léger : les solutions envisagées

- Quelle que soit la complexité du site Web les technologies côté clients sont toujours identiques : JS / HTML / CSS
- Au contraire le choix de la technologie côté serveur dépend fortement de la complexité de l'application Web
 - Site marketing et recherche de simples informations : PHP ou ASP
 - Site commercial avec transaction: langage à objets Java EE ou .NET
- Avis personnels
 - L'utilisation de technologies à objets permettent d'imposer une architecture
 - Les technologies à objets offrent un nombre important d'API
 - La persistance liée au paradigme objets permet de gérer plus facilement les sessions utilisateurs et le stockage d'attributs (compteur)
- Exemple : Java EE

LABS

Utilisation de Git



Sommaire

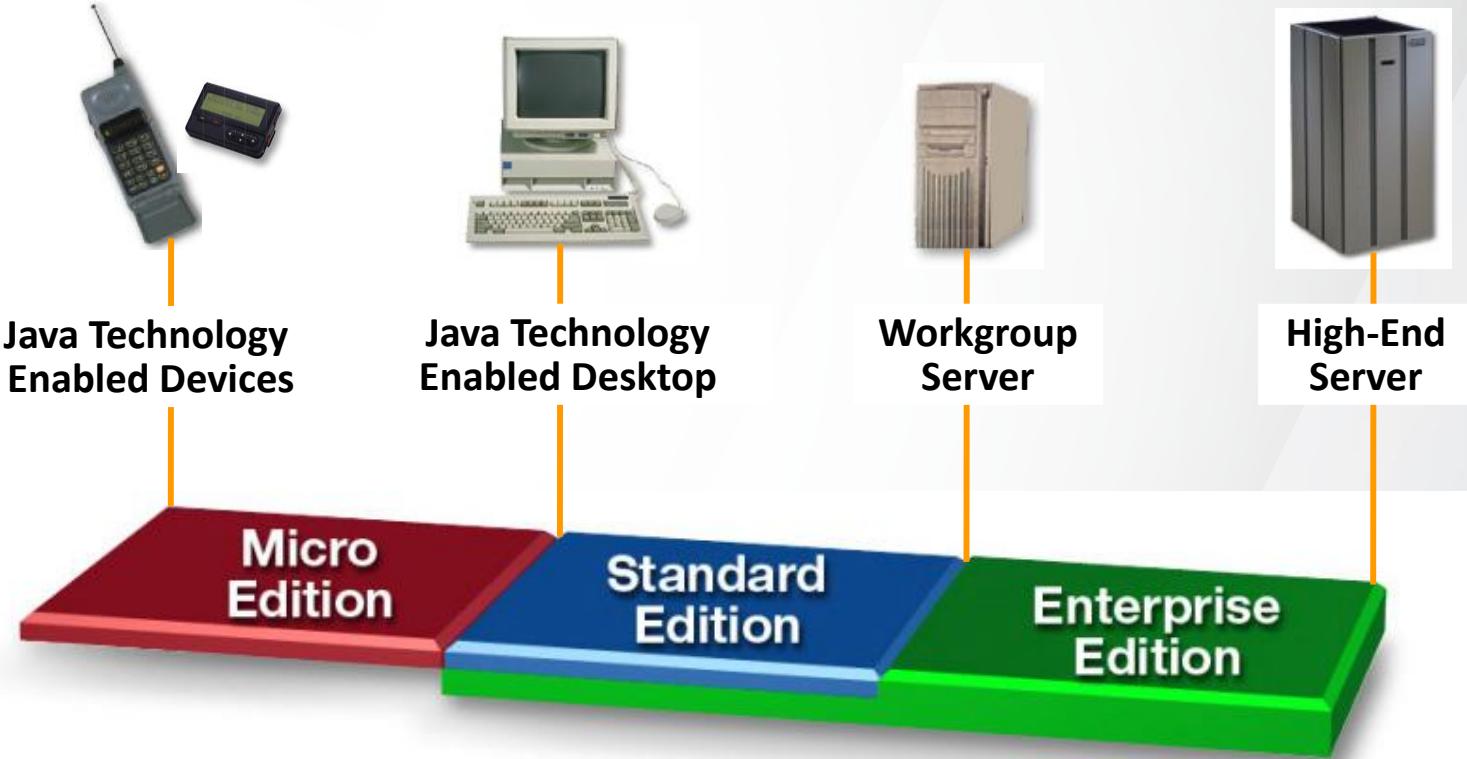
- Chapitre 1 : INTRODUCTION
 - **Chapitre 2 : PLATE-FORME JAVA EE**
 - Chapitre 3 : SERVLET / JSP
 - Chapitre 4 : LES BASES DE DONNEES AVEC JAVA EE
 - Chapitre 5 : JSF
 - Chapitre 6 : PRIMEFACES
-
- 



Présentation générale

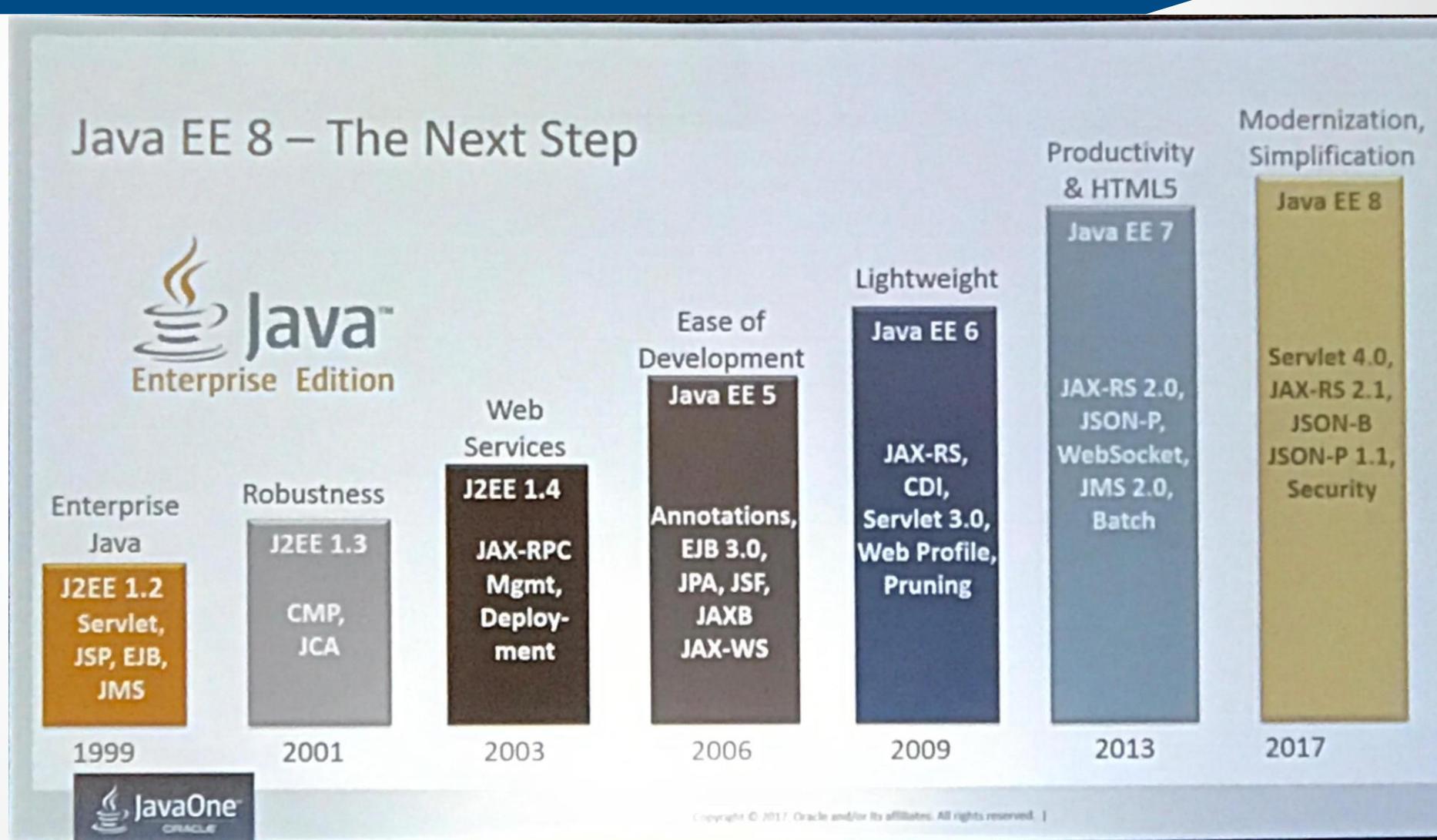
➤ Qu'est ce que JAVA

- Un langage de programmation interprété et compilé.
 - Langage portable : pseudo-code

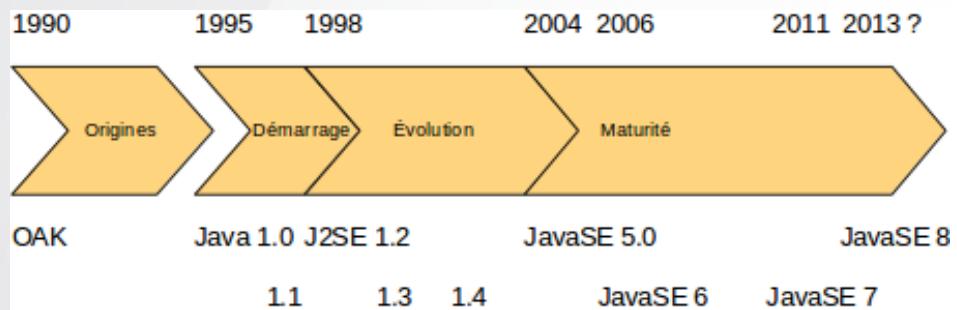


Présentation générale

➤ Historique :



Présentation générale



Année	Événements
1995	mai : premier lancement commercial du JDK 1.0
1996	janvier : JDK 1.0.1
1996	septembre : lancement du JDC
1997	février : JDK 1.1
1998	décembre : lancement de J2SE 1.2 et du JCP
1999	décembre : lancement J2EE
2000	mai : J2SE 1.3
2002	février : J2SE 1.4
2004	septembre : J2SE 5.0
2006	Mai : Java EE 5 décembre : Java SE 6.0
2008	décembre : Java FX 1.0
2009	février : JavaFX 1.1 juin : JavaFX 1.2 décembre : Java EE 6
2010	janvier : rachat de Sun par Oracle avril : JavaFX 1.3
2011	juillet : JavaSE 7 octobre : JavaFX 2.0
2012	août : JavaFX 2.2
2013	juin : Java EE 7
2014	mars : Java SE 8, JavaFX 8

Java

JDK 8

- Lambda
- JSR 310
- Nashorn
- Interopera
- JavaFX E

201

Présentation générale

➤ Comment se fait l'évolution de JAVA



Présentation générale

- Le JCP fait la norme JEE.
- L'industrie et le monde du libre ne passent pas forcément par le JCP.
- Le JCP intègre souvent les bonnes idées mais cela peut prendre du temps.
- Exemple :
 - la JSR 168 est la spécification des portlets définissant le contrat entre les conteneurs de portlets et les portlets.

Présentation générale

- Qu'est ce que la JEE :
 - JEE est la version « entreprise » de Java, elle a pour but de faciliter le développement d'applications distribuées.
 - Depuis sa version 5, J2EE est renommé Java EE (Enterprise Edition).
 - Généralement, les applications JEE fonctionnent à l'intérieur d'un serveur d'applications (conteneur)

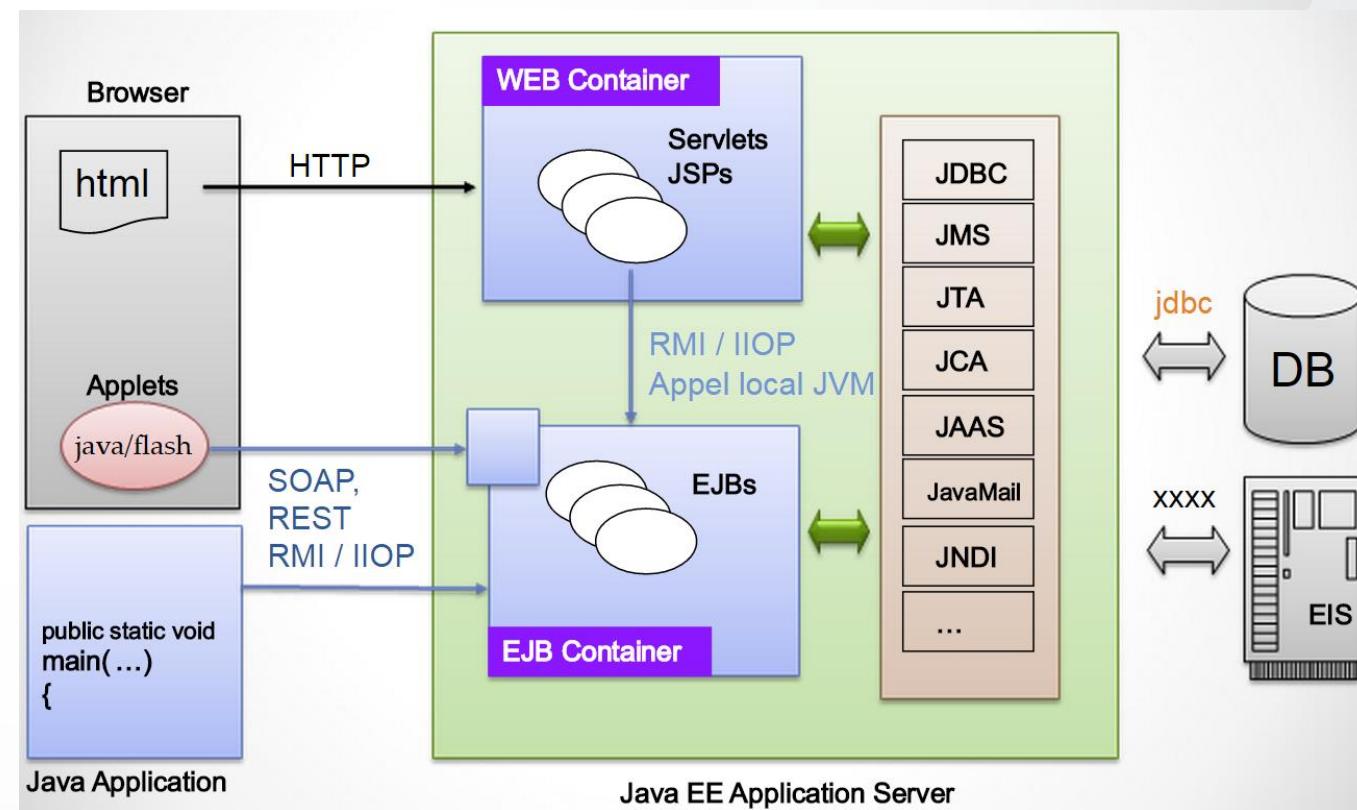


Présentation générale

- JEE est composée de deux parties essentielles :
 - Un ensemble de spécifications pour une infrastructure dans laquelle s'exécutent les composants écrits en Java
 - Un ensemble d'API qui peut être obtenu et utilisé séparément.

Présentation générale : JEE

- Ces API peuvent être regroupées en trois grandes catégories :
 - les composants : Servlet, JSP, EJB
 - les services : JDBC, JTA/JTS, JNDI, JCA, JAAS
 - la communication : RMI-IIOP, JMS, Java Mail



Présentation générale : JEE

➤ Ce qu'il faut retenir :

- JEE décrit des services techniques pour bâtir des applications d'entreprise.
- Une application JEE s'exécute dans un serveur d'applications qu'est un environnement d'exécution.

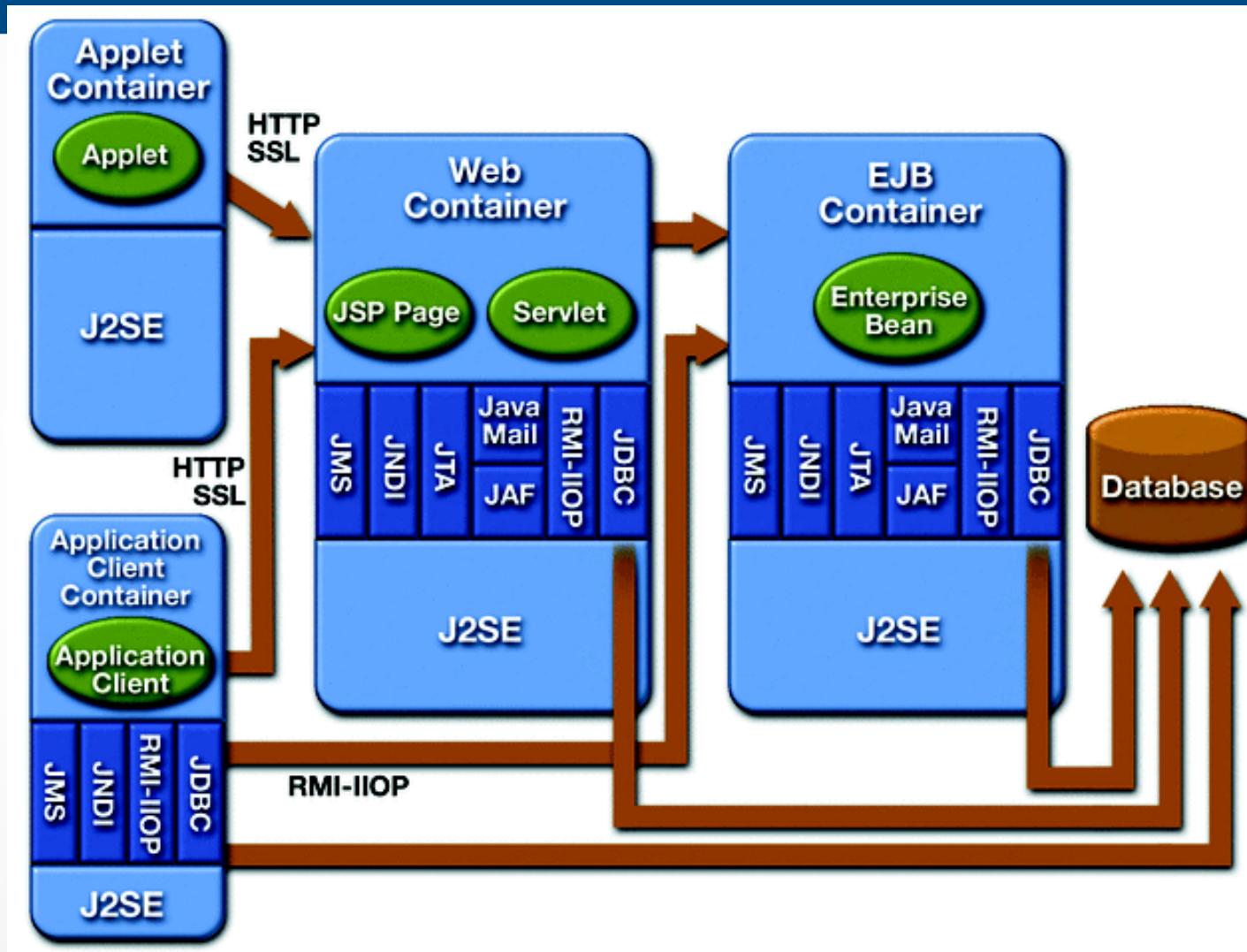
Les conteneurs JEE

- Par définition, Un conteneur JEE est un environnement d'exécution chargé de gérer des composants applicatifs et de donner accès aux API JEE.
- C'est un endroit où seront déployées des applications JEE.
- Le conteneur offre des services tels que:
 - Gestion de la durée de vie des composants applicatifs
 - Pooling de ressources
 - Peuplement de l'espace de noms JNDI avec les objets nécessaires à l'utilisation des API de services des conteneurs
 - Clustering sur plusieurs machines (répartition de charge ou Load Balancing)
 - Sécurité

Les conteneurs JEE

- Les conteneurs sont les éléments fondamentaux de l'architecture J2EE.
- Les conteneurs fournis par J2EE sont de même type.
- Ils fournissent une interface parfaitement définie ainsi qu'un ensemble de services permettant aux développeurs d'applications de se **concentrer** sur la logique métier à mettre en œuvre pour résoudre le problème qu'ils ont à traiter, sans qu'ils aient à se **préoccuper** de toute l'infrastructure interne.
- Les conteneurs s'occupent de toutes les tâches fastidieuses liées
 - au démarrage des services sur le serveur,
 - à l'activation de la logique applicative,
 - la gestion des protocoles de communication intrinsèque
 - à la libération des ressources utilisées.

Les conteneurs JEE



Les conteneurs JEE

- On distingue 4 types de conteneur JEE:
 - Conteneur des applications client
 - Conteneur Applet (applet java)
 - Conteneur Web (Servlet, JSP, etc...)
 - Conteneur EJB (Enterprise JavaBeans)

Les conteneurs JEE

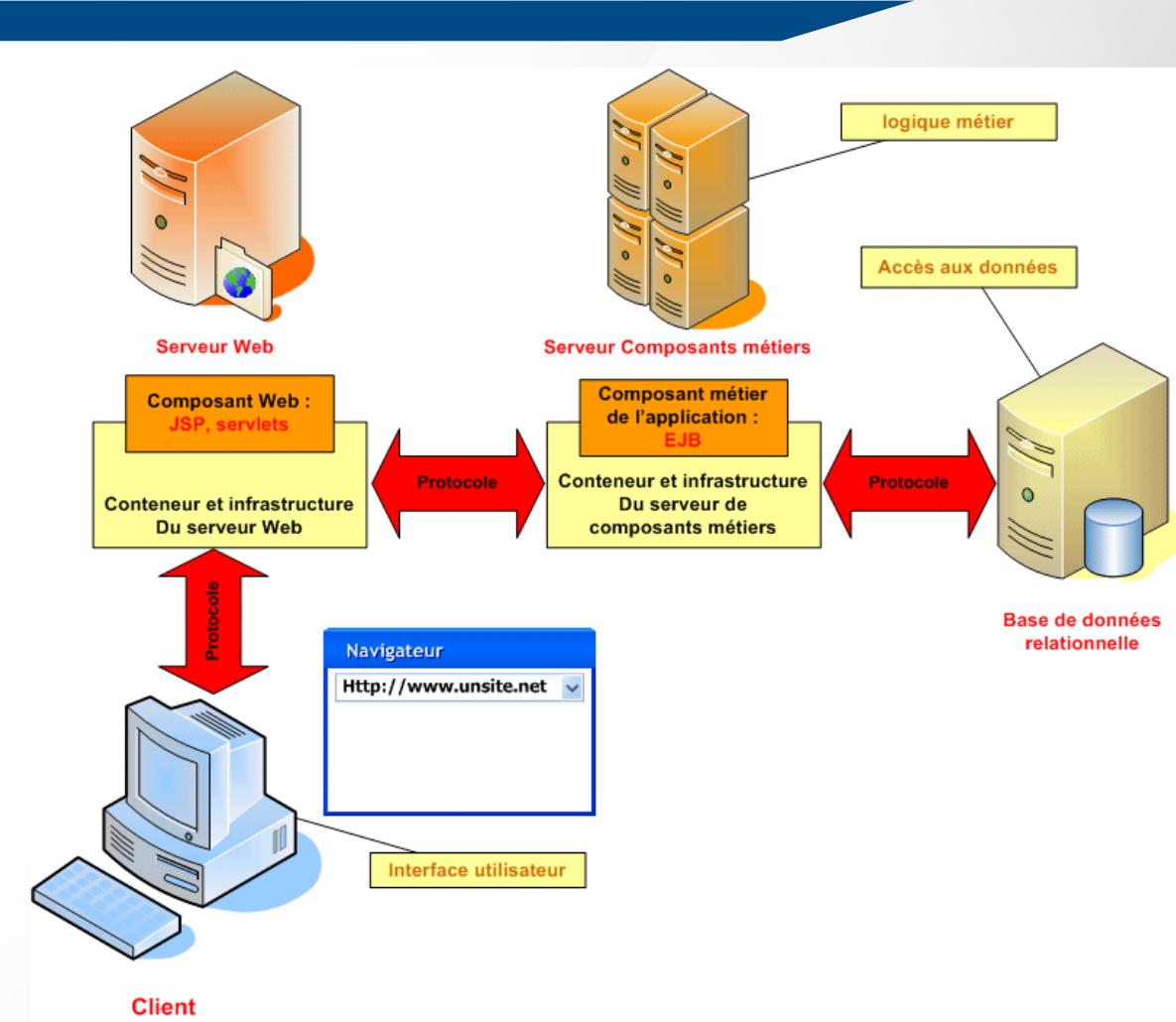
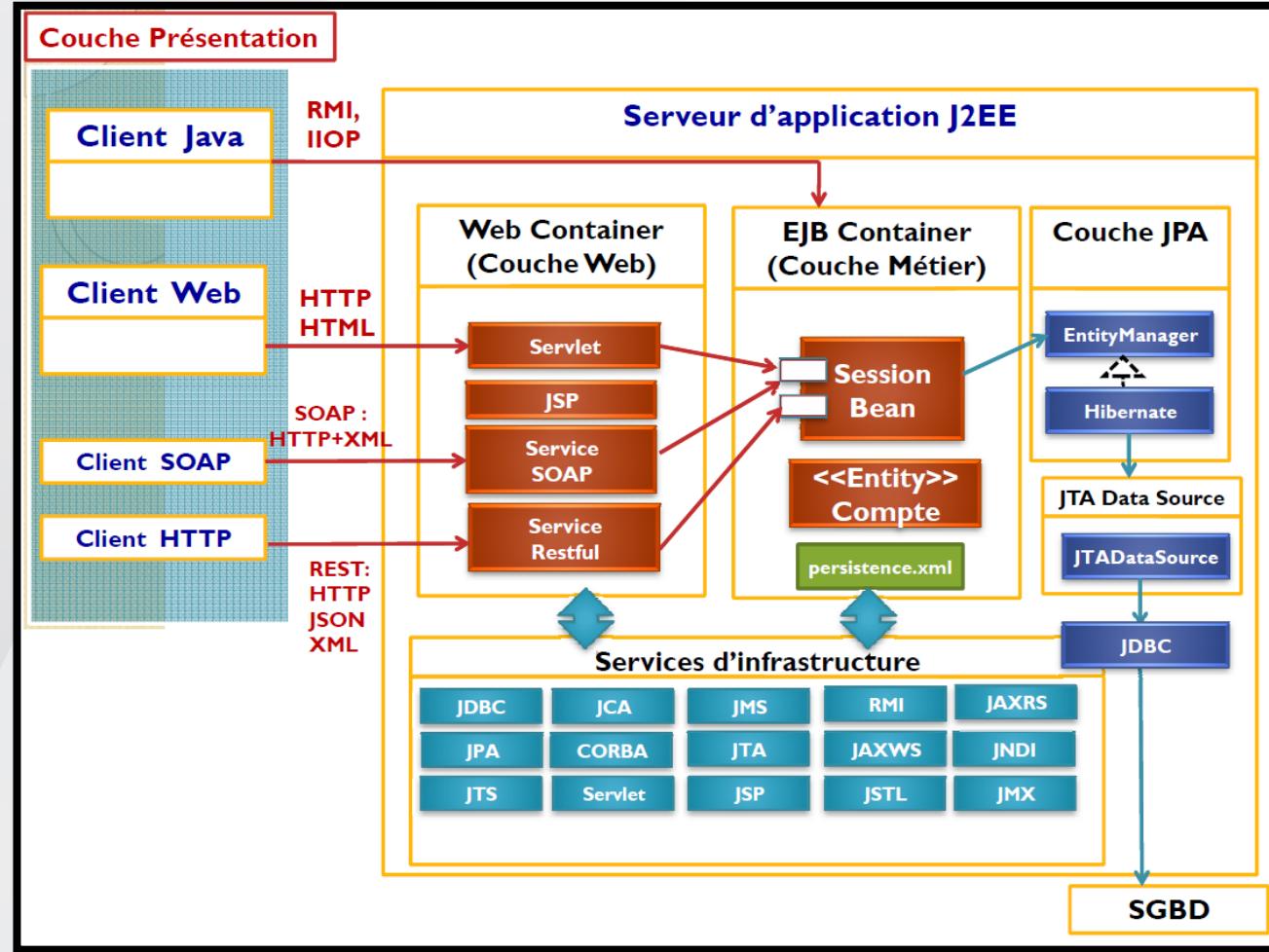
- Les serveurs d'applications peuvent fournir un conteneur web uniquement (exemple : Tomcat) ou un conteneur d'EJB uniquement (exemple : JBoss, Jonas, ...) ou les deux (exemple : Websphere, Weblogic, ...).

- Pour déployer une application dans un conteneur, il faut lui fournir deux éléments :
 - l'application avec tous les composants (classes compilées, ressources ...) regroupés dans une archive ou module. Chaque conteneur possède son propre format d'archive.
 - un fichier descripteur de déploiement contenu dans le module qui précise au conteneur des options pour exécuter l'application

Les conteneurs JEE

- Une plate-forme d'exécution J2EE complète implémentée dans un serveur d'applications propose les services suivants :
 - service de nommage (naming service)
 - service de déploiement (deployment service)
 - service de gestion des transactions (transaction service)
 - service de sécurité (security service)
- Ces services sont utilisés directement ou indirectement par les conteneurs mais aussi par les composants qui s'exécutent dans les conteneurs grâce à leurs API respectives.

Architecture classique JEE



Couche Présentation

- Elle implémente la logique présentation de l'application
- La couche présentation est liée au type de client utilisé :
 - Client Lourd java Desktop:
 - Interfaces graphiques java SWING, AWT, SWT.
 - Ce genre de client peut communiquer directement avec les composants métiers déployés dans le conteneur EJB en utilisant le middleware RMI (Remote Method Invocation)
 - Client Leger Web
 - HTML, Java Script, CSS.
 - Un client web communique avec les composants web Servlet déployés dans le conteneur web du serveur d'application en utilisant le protocole HTTP.
 - Un client .Net, PHP, C++, ...
 - Ce genre de clients développés avec un autre langage de programmation autre que java,
 - communiquent généralement avec les composants Web Services déployés dans le conteneur Web du serveur d'application en utilisant le protocole SOAP (HTTP+XML)
 - Client Mobile
 - Android, iPhone, Tablette etc..
 - Généralement ce genre de clients communique avec les composants Web Services en utilisant le protocole HTTP ou SOAP

Couche Application ou Web

- Appelée également couche web.
- La couche application sert de médiateur entre la couche présentation et la couche métier.
- Elle contrôle l'enchaînement des tâches offertes par l'application
 - Elle reçoit les requêtes http clientes
 - Assure le suivi des sessions
 - Vérifier les autorisations d'accès de chaque session
 - Assure la validation des données envoyées par le client
 - Fait appel au composants métier pour assurer les traitements nécessaires
 - Génère une vue qui sera envoyée à la couche présentation.
- Elle utilise les composants web Servlet et JSP
- Elle respecte le modèle MVC (Modèle Vue Contrôleur)
- Des framework comme JSF, SpringMVC ou Struts sont généralement utilisés dans cette couche

Couche Métier ou service

- La couche métier est la couche principale de toute application
 - Elle implémente la logique métier d'une entreprise
 - Elle se charge de récupérer, à partir des différentes sources de données, les données nécessaires pour assurer les traitements métiers déclenchés par la couche application.
 - Elle assure la gestion du WorkFlow (Processus de traitement métier en plusieurs étapes)
- Il est cependant important de séparer la partie accès aux données (Couche DAO) de la partie traitement de la logique métier (Couche Métier) pour les raisons suivantes :
 - Ne pas se perdre entre le code métier, qui est parfois complexe, et le code d'accès aux données qui est élémentaire mais conséquent.
 - Ajouter un niveau d'abstraction sur l'accès aux données pour être plus modulable et par conséquent indépendant de la nature des unités de stockage de données.
 - La couche métier est souvent stable. Il est rare qu'on change les processus métier. Alors que la couche DAO n'est pas stable. Il arrive souvent qu'on est contraint de changer de SGBD ou de répartir et distribuer les bases de données.
 - Faciliter la répartition des tâches entre les équipes de développement.
 - Déléguer la couche DAO à frameworks spécialisés dans l'accès aux données (Hibernate, Toplink, etc...)

Une plate-forme plus légère

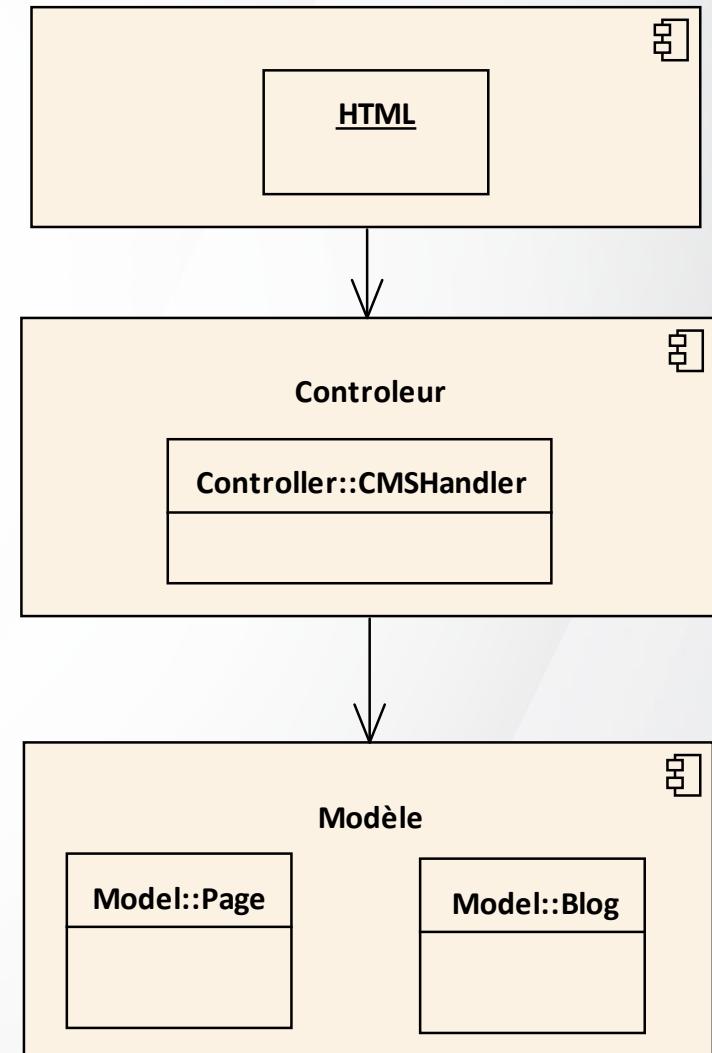
- Une critique récurrente de la plate-forme Java EE est qu'elle est très riche et donc complexe.
- La plupart des applications de petite taille ou de taille moyenne n'ont pas besoin de toutes les technologies proposées par la plate-forme.
- Cette richesse est aussi liée au fait que certaines technologies sont obsolètes car remplacées par une nouvelle technologie : ces anciennes versions sont cependant conservées pour des raisons de compatibilité.

MVC - Définition

- MVC = ***Model View Controller***
- Le modèle représente les ***entités du système***
- Le contrôleur implémente la ***logique métier*** et la ***logique des interactions***
- La vue représente ***l'interface utilisateur***

MVC - Exemple

- .NET : ASP.NET MVC, MonoRail
- Java : JavaServer Faces (JSF), Struts
- Ruby On Rails
- Python : Zope
- Flex : Cairngorm



MVC - Avantages

- Modèle de conception largement apprécié de la communauté de développeurs
- Séparation de la logique de l'interface
- Testabilité accrue (les tests unitaires supportent le modèle et le contrôleur)
- la réutilisation des composants

MVC - Inconvénients

- Assez complexe
- Plus d'efforts de développement car chaque tâche concerne les trois couches
- Structure du système non-intuitive :
 - contrôle des calculs donné aux Système
 - Quelles réactions à un événement ?
 - Dans quel ordre, les traitements?

Approche MVC

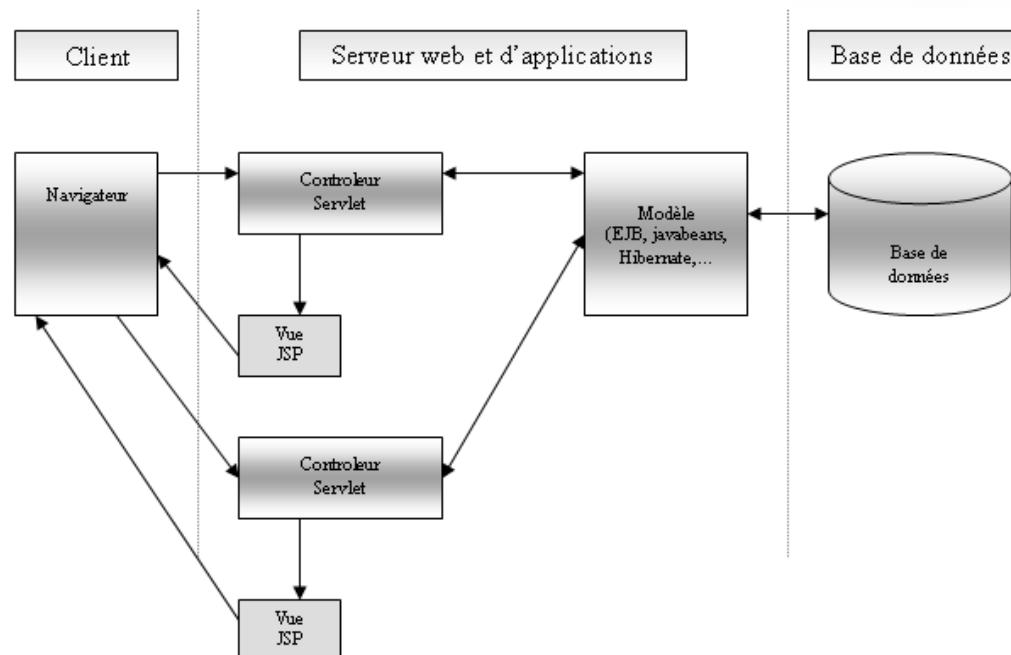
- Le modèle MVC est un modèle de conception logicielle largement répandu, fort et utile. Néanmoins il faut retenir que c'est un modèle de conception, et il est donc indépendant du langage de programmation.
- Le MVC est un modèle d'architecture qui repose sur la volonté de séparer les données, les traitements et la présentation. Ainsi l'application se retrouve segmentée en trois composants essentiels :

Approche MVC

- ***La vue*** correspond à l'IHM. Elle présente les données et interagit avec l'utilisateur. Dans le cadre des applications Web, il peut s'agir d'une interface HTML/JSP, mais n'importe quel composant graphique peut jouer ce rôle.
- ***Le contrôleur***, quant à lui, se charge d'intercepter les requêtes de l'utilisateur, d'appeler le modèle puis de rediriger vers la vue adéquate. Il ne doit faire aucun traitement. Il ne fait que de l'interception et de la redirection.
- ***Le modèle*** représente les données et les règles métiers. C'est dans ce composant que s'effectuent les traitements liés au cœur du métier. Les données peuvent être liées à une base de données, des EJBs, des services Web, etc.

Approche MVC

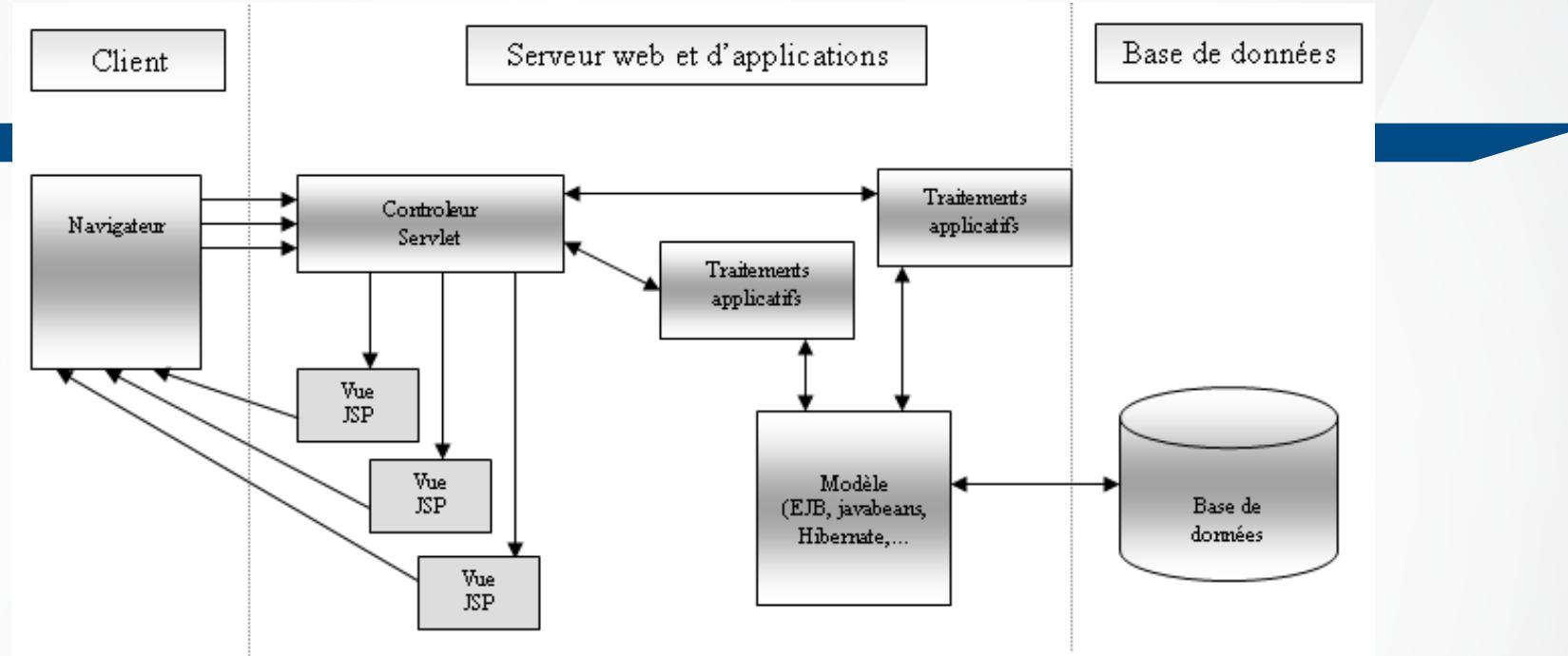
- Il est important de noter que les données sont indépendantes de la présentation.
 - En d'autres termes, le modèle ne réalise aucune mise en forme. Ces données pourront être affichées par plusieurs vues. De ce fait le code du modèle est factorisé : il est écrit une seule et unique fois puis réutilisé par chaque vue.



Approche MVC : MVC2

- Le MVC très pratique, peut se révéler lourd à mettre en place. Ceci à cause de la multitude de contrôleurs à implémenter, car comme nous l'avons expliqué précédemment, chaque vue possède son propre contrôleur.
- Dans l'organisation logicielle MVC2, le Servlet est unique, en classe et en instance.
 - Il garantit l'unicité du point d'entrée de l'application.
 - Il prend en charge une partie du contrôle de l'application.
- Les contrôleurs MVC se retrouvent alors partiellement déportés dans l'entité "dynamique de l'application" qui assure le contrôle de la dynamique de l'application et qui gère les relations entre les objets métier et la présentation.
- Les contrôleurs deviennent essentiellement des contrôleurs du dialogue entre l'utilisateur et les objets métiers.

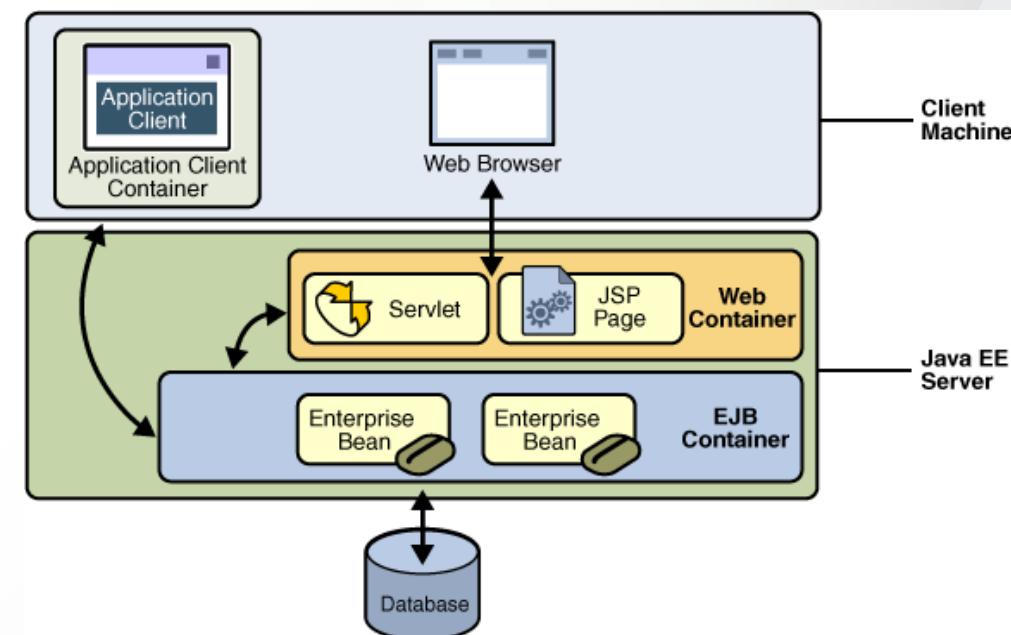
Approche MVC : MVC2



1. Le client envoie une requête à l'application. La requête est prise en charge par le Servlet d'entrée.
2. Le Servlet d'entrée analyse la requête et réoriente celle-ci vers un contrôleur adapté.
3. Le contrôleur sélectionné par le Servlet d'entrée est responsable de l'exécution des traitements nécessaires à la satisfaction de la requête. Il sollicite les objets métiers lorsque nécessaire.
4. Les objets métiers fournissent des données au contrôleur.
5. Le contrôleur encapsule les données métiers dans des JavaBeans, sélectionne la JSP qui sera en charge de la construction de la réponse et lui transmet les JavaBean contenant les données métier.
6. La JSP construit la réponse en faisant appel aux JavaBeans qui lui ont été transmis et l'envoie au navigateur.
7. Lorsque nécessaire, pour le traitement d'erreurs essentiellement, le Servlet d'entrée traite la requête directement, sélectionne la JSP de sortie et lui transmet par JavaBean les informations dont elle a besoin.

Qu'est ce qu'une livraison JEE ?

- Un livrable J2EE représente un fichier possédant une des extensions suivantes :
 - .ear, .war, .jar, .rar
- Les livrables sont déployées :
 - dans un serveur d'application certifié JEE (conteneur Web + conteneur EJB) comme JBOSS , GlassFish ...
 - ou dans un simple conteneur web (moteur de servlet/JSP comme Tomcat).



Qu'est ce qu'une livraison JEE ? (2)

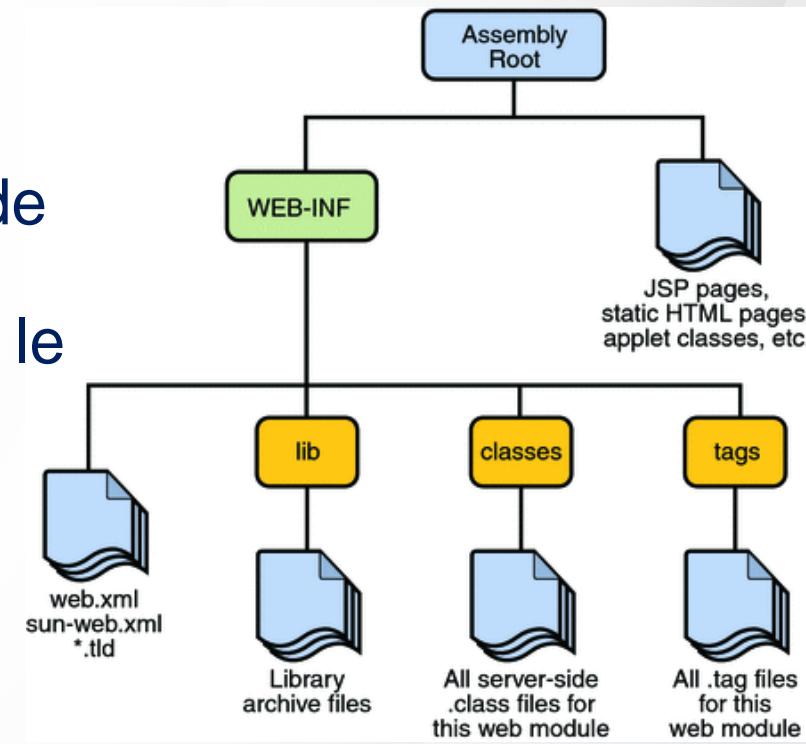
- Développer une application J2EE revient à créer les différents livrables, suivant la complexité des besoins de l'application.
- Le développement de chaque livrable peut être confié à une équipe ou plusieurs équipes de développement réparties dans plusieurs pays (Exemple : France, Tunisie, Inde).
- Les développeurs livrent alors leur travail dans un référentiel commun, comme Subversion.
- La construction (l'assemblage) des livraisons peut être confiée à des outils comme Ant/Ivy ou encore mieux...MAVEN !.

Livraison .War

- La plupart des applications J2EE (70-80%) sont livrées dans un module WAR, contenant les écrans de l'application, les composants d'accès aux données ainsi que des composants métiers.
- Vous trouverez dans ce livrable :
 - Ecrans de l'application (Pages (X)HTML, JSP)
 - Images de l'application
 - Eléments du graphisme (Feuilles de style CSS, XSL, TId)
 - Classes Java métier (JavaBean) ou d'accès aux données (DAO)
 - Fichier de configuration web.xml

Livraison .War (2)

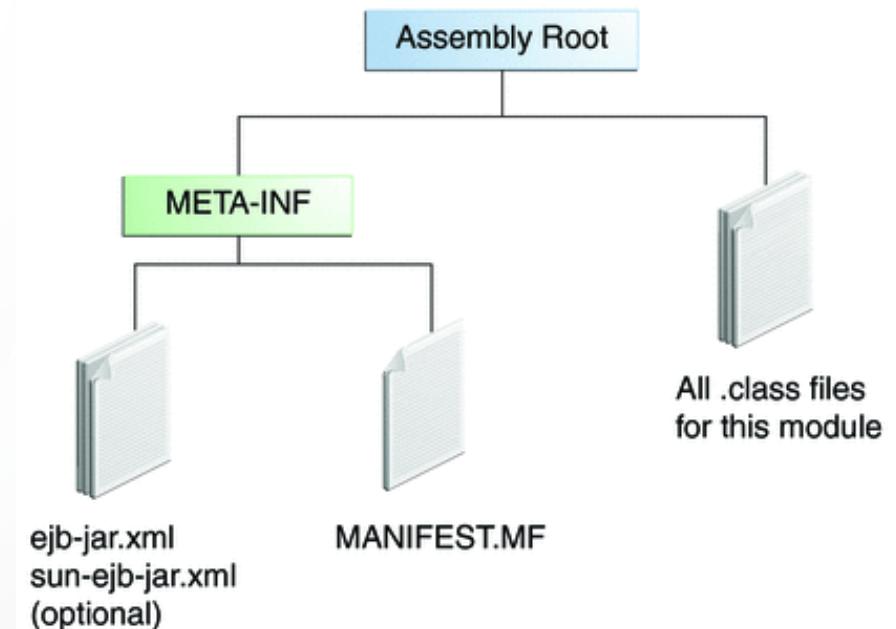
- Selon la spécification J2EE, une application Web doit avoir la structure suivante:
 - un répertoire racine public contenant les pages HTML, les pages JSP, les images...
 - un répertoire **WEB-INF** situé dans le répertoire racine de l'application web.
 - un fichier **web.xml** situé à la racine de WEB-INF : c'est le descripteur de déploiement de l'application web.
 - un répertoire **WEB-INF/classes** contenant les classes compilées de l'application (**servlets, classes auxiliaires...**
Le tout peut être empaqueté dans une archive sous la forme d'un fichier WAR (réalisé avec l'utilitaire jar ou zip).)
 - un répertoire **WEB-INF/lib** contenant les fichiers JAR de l'application (drivers JDBC, frameworks empaquetés...).



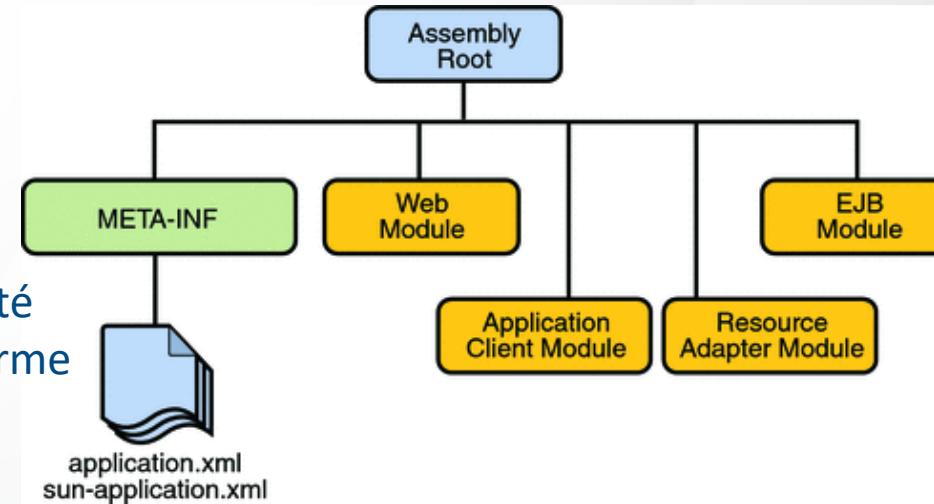
Livraison .jar

- Les EJB (Enterprise JavaBean) sont des composants java métiers évoluant dans le conteneur EJB d'un serveur d'applications JEE.
- Ils bénéficient de plusieurs services offerts par le conteneur web, dont la sécurité et la transaction déclarative.
- Selon la spécification JEE , un fichier JAR doit avoir la structure suivante :
 - un répertoire **META-INF/** contenant un descripteur de déploiement XML du module EJB, nommé **ejb-jar.xml**
 - En racine on trouve les fichiers.class correspondant aux interfaces et aux classes d'implémentation des EJBs

Un EJB est livré sous forme de .jar indépendant ou à l'intérieur d'une livraison globale .ear

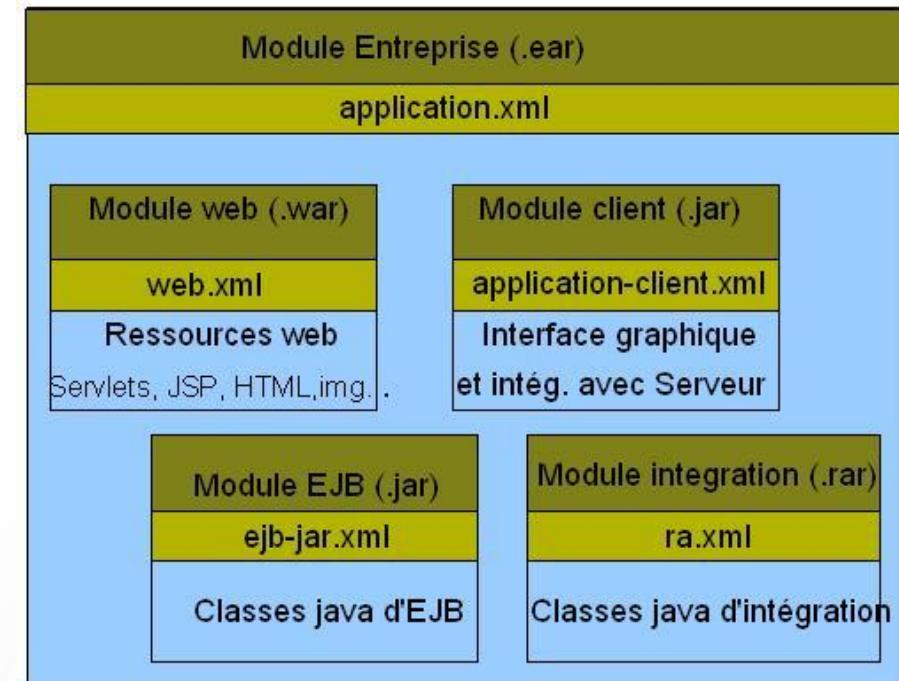


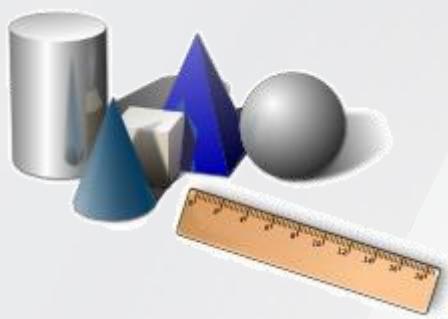
Livraison .ear

- Selon les spécifications JEE, une application d'entreprise doit avoir la structure suivante :
 - un répertoire **META-INF/** contenant le descripteur de déploiement XML de l'application JEE nommé **application.xml**.
 - C'est dans ce descripteur que l'on définit les modules web et EJB qui constituent l'application d'entreprise.
 - On y précise par exemple sur quelle racine du serveur web (placé en frontal devant le serveur d'application) doit résider l'application web.
 - les fichiers archives.JAR et .WAR correspondant aux modules EJB et aux modules Web de l'application d'entreprise.
- Le tout peut être empaqueté dans une archive sous la forme d'un fichier EAR
- 
- ```
graph TD; AR[Assembly Root] --- META[META-INF]; AR --- WM[Web Module]; AR --- EM[EJB Module]; AR --- ACM[Application Client Module]; AR --- RAM[Resource Adapter Module]; META --- XML1[application.xml]; META --- XML2[sun-application.xml]
```
- The diagram illustrates the structure of an Enterprise Archive (EAR) file. At the top is a light blue rounded rectangle labeled "Assembly Root". Below it, several components are shown in colored boxes: a green box for "META-INF", a yellow box for "Web Module", a yellow box for "EJB Module", and two smaller yellow boxes for "Application Client Module" and "Resource Adapter Module". Lines connect the "Assembly Root" to each of these components. Below the "META-INF" component are two smaller blue icons representing XML files, labeled "application.xml" and "sun-application.xml".

# Livraison .ear (2)

- La livraison englobant toutes les autres est celle d'extension ear (Enterprise ARchive).
- Utile dès que l'application est complexe, impliquant des services d'entreprise (Transactionel, Sécurité, communication multi-serveurs, Base de données, Mainframe, Reporting, fournisseur de Message...).

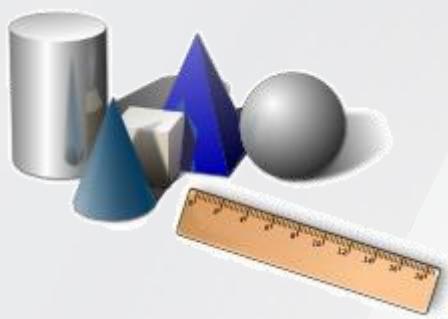




# Atelier 0 : Installation de Tomcat 9

Les prérequis d'installation pour Tomcat8 :

- Vérification du JDK 8
  - Si ce n'est déjà pas fait il faut vérifier si la variable JAVA\_HOME pointe vers la bonne version de java
  - Tapez en ligne de commande :
    - echo %JAVA\_HOME%
  - Sinon il faut télécharger le JDK 8 et créer la variable d'environnement : JAVA\_HOME



# Atelier 0 : Installation de Tomcat 9

- Téléchargement de tomcat sur le site : <http://tomcat.apache.org/>
- Sélectionnez Tomcat9, puis la dernière version stable

The screenshot shows the Apache Tomcat 9 download page. The left sidebar has links for Home, Taglibs, Maven Plugin, Download (with Tomcat 9 selected), Tomcat 8, Tomcat 7, Tomcat 6, and Tomcat Connectors. The main content area has a logo, the title "Apache Tomcat®", and a "Tomcat 8 Software Download" section with links for KEYS, 8.0.35, 8.5.2 BETA, and Browse. Below that is a "Release Integrity" section with a note about verifying integrity. The right side shows the "9.0.13" version information and a "Binary Distributions" section with a list of download links for Core distributions: zip (pgp, sha512), tar.gz (pgp, sha512), 32-bit Windows zip (pgp, sha512), 64-bit Windows zip (pgp, sha512), and 32-bit/64-bit Windows Service Installer (pgp, sha512). The last item in the list is highlighted with a red border.

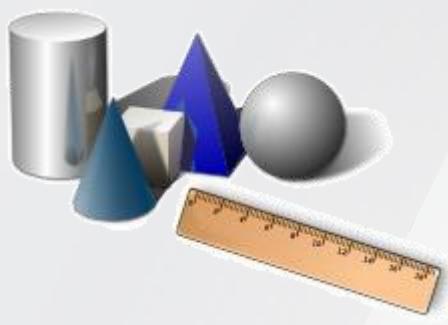
Apache Tomcat®

9.0.13

Please see the [README](#) file for packaging information. It explains what every distribution contains.

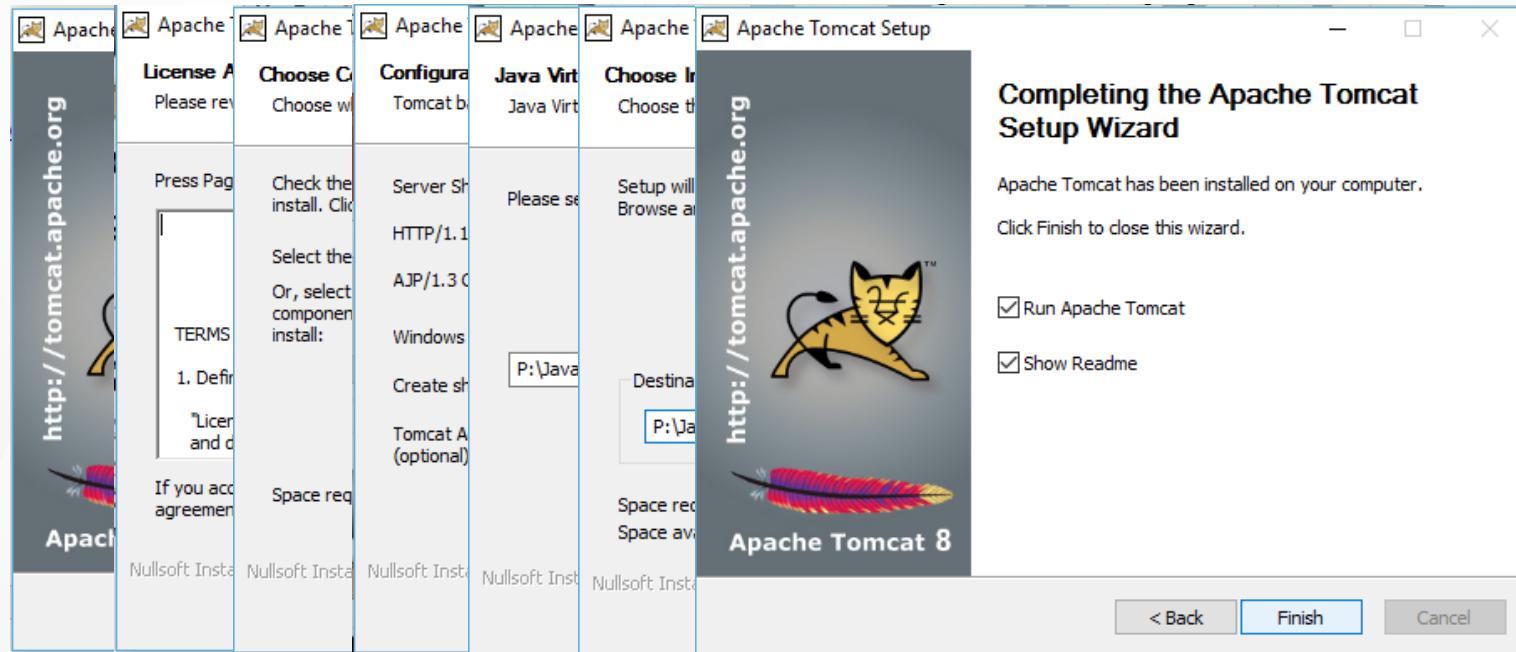
Binary Distributions

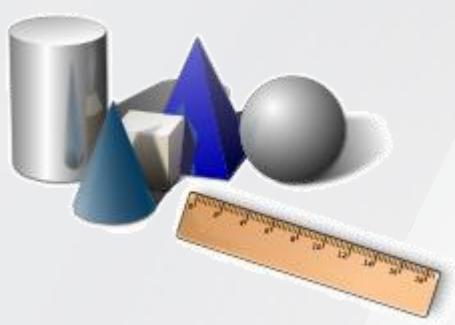
- Core:
  - [zip \(pgp, sha512\)](#)
  - [tar.gz \(pgp, sha512\)](#)
  - [32-bit Windows zip \(pgp, sha512\)](#)
  - [64-bit Windows zip \(pgp, sha512\)](#)
  - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)



# Atelier 0 : Installation de Tomcat 9

- Une fois l'exécutable téléchargé, suivez les étapes :

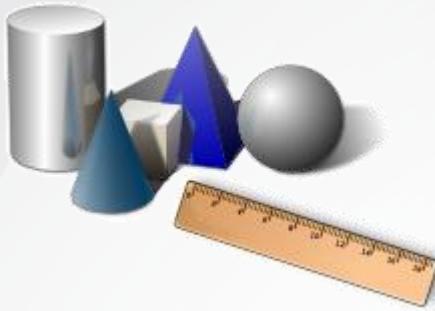




# Atelier 0 : Installation de Tomcat 9

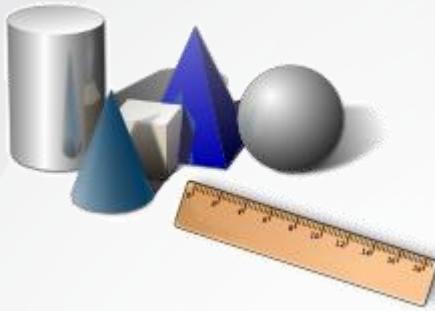
- Test : Lancez le navigateur sur l'url :
  - <http://localhost:8080> . L'écran suivant apparaît.

The screenshot shows a web browser window with the URL `localhost:8080` in the address bar. The page title is "Apache Tomcat/8.0.35". On the left, there's a cartoon cat icon with the text "TM". To its right, under "Recommended Reading", are links to "Security Considerations HOW-TO", "Manager Application HOW-TO", and "Clustering/Session Replication HOW-TO". On the right side of the main content area, there are three buttons: "Server Status", "Manager App", and "Host Manager". Below this, under "Developer Quick Start", are links to "Tomcat Setup", "First Web Application", "Realms & AAA", and "JDBC DataSources". Further down, under "Examples", are links to "Servlet Specifications" and "Tomcat Versions". At the bottom of the page, there are three yellow boxes: "Managing Tomcat" (with a note about restricted access to the manager webapp), "Documentation" (with links to "Tomcat 8.0 Documentation", "Tomcat 8.0 Configuration", and "Tomcat Wiki"), and "Getting Help" (with a link to "FAQ and Mailing Lists" and a note about available mailing lists).



# Atelier 1 : Déploiement TOMCAT

- Récupérer le projet se trouvant sous le répertoire hello
- **1) Déploiement manuelle :**
  - Copier le répertoire hello dans le répertoire webapps de tomcat
  - Vérifier sous l'adresse <http://localhost:8080/manager/html> que l'application est déployé
  - Tester l'application : <http://localhost:8080/hello/>



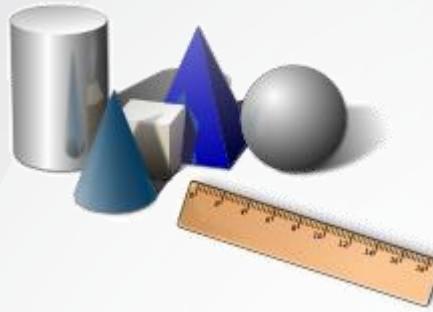
# Atelier 1 : Déploiement TOMCAT

## ➤ 2) Déploiement depuis l'interface d'administration de Tomcat

- Récupérer le fichier hellodynamique.war
- Connectez vous a l'interface d'administration de Tomcat :  
<http://localhost:8080/manager/html>

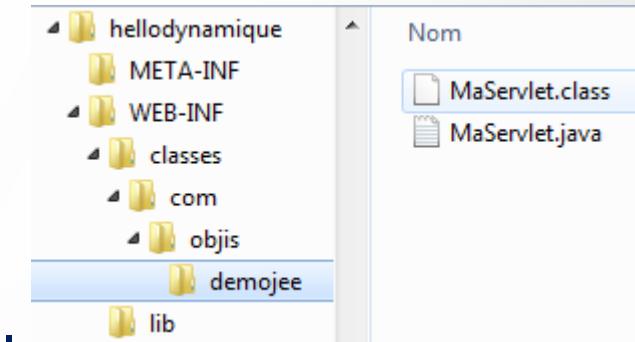
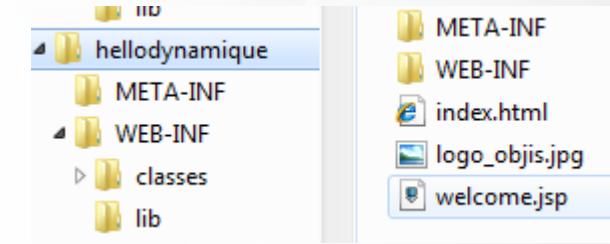
The screenshot shows the Tomcat Manager Web interface. The top section is titled "Deployer" and contains fields for "Chemin de context (requis)", "URL du fichier XML de configuration", and "URL vers WAR ou répertoire", each with an associated input field. A "Deployer" button is located below these fields. The bottom section is titled "Fichier WAR à déployer" and contains a "Choisissez le fichier WAR à téléverser" label, a "Choisissez un fichier" button, and a message "Aucun fichier choisi". Another "Deployer" button is located at the bottom of this section.

- Cliquer que le bouton « Choisissez un fichier » et sélectionner le war récupérer précédemment puis cliquer sur le bouton Déployer

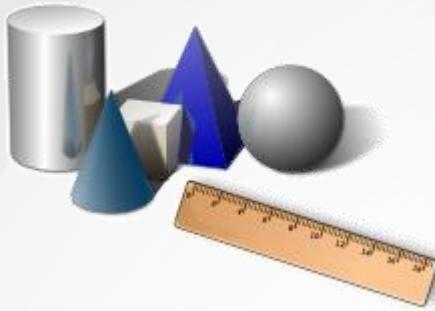


# Atelier 1 : Déploiement TOMCAT

- Vérifier que l'application est démarré
- Tester l'application
- Analysez le contenu du livrable.
  - En particulier, la présence d'une JSP à la racine du livrable....



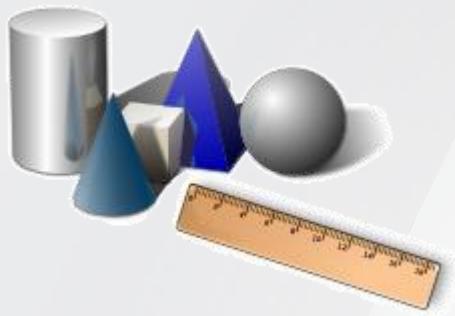
- ...et d'une SERVLET dans le répertoire WEB-INF\classes.



# Atelier 1 :

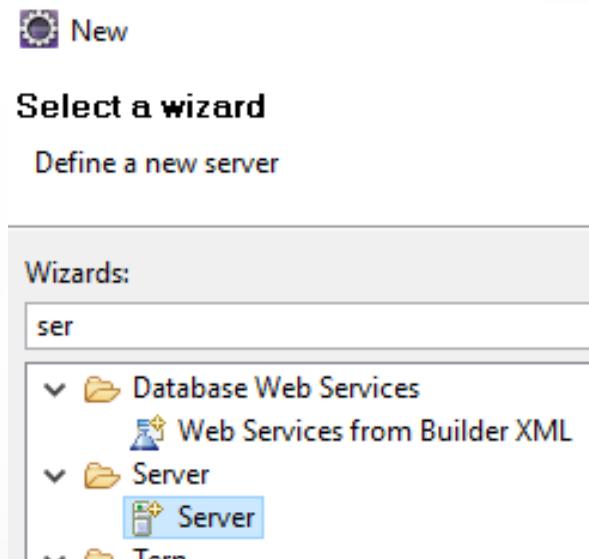
## Déploiement TOMCAT

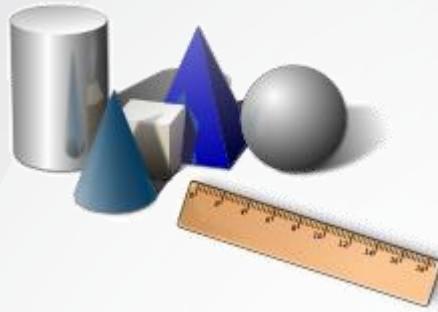
- Quel est l'effet de l'adresse suivante :  
<http://localhost:8080/helldynamique/coucou>
- Indication : Regardez le fichier web.xml
- Analysez le code source de la Servlet
- Analysez le code source de la page jsp
- Analyser le cache de tomcat :
  - P:\Java\tomcat8\work\Catalina\localhost\helldynamique\org\apache\jsp



# Atelier 1 : Déploiement TOMCAT

- **3) Déploiement depuis IDE : cas d'Eclipse**
- Connecté Eclipse a l'installation de Tomcat
  - Création d'un nouveau server / choix de tomcat 9 puis spécifier l'emplacement du serveur Tomcat installé sur votre ordinateur
  - Choisissez le JDK8 comme jre a utilisé pour tomcat

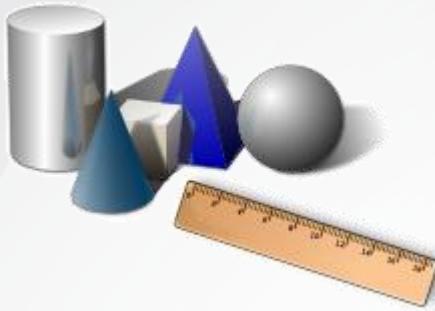




# Atelier 1 : Déploiement TOMCAT

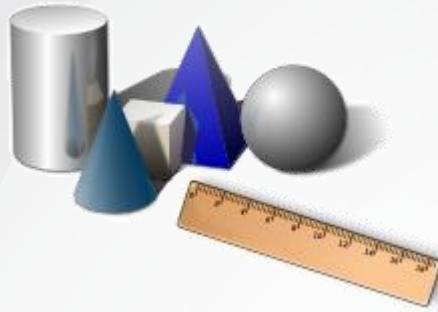
- Arrêt du service Tomcat
- Depuis Eclipse démarrer le serveur
- Tester maintenant en tapant l'url :  
<http://localhost:8080/>
- Une erreur de ce type apparait
  - La bonne nouvelle c'est que c'est Tomcat qui répond
  - Mais il manque les fichier root qu'eclipse n'a pas copié

The screenshot shows the Eclipse IDE interface. A context menu is open over a Tomcat server entry in the Servers view, with the 'Start' option highlighted. Below the interface is a browser window displaying a 404 error page for the URL [localhost:8080](http://localhost:8080). The error message is: "Etat HTTP 404 - /". The Apache Tomcat/8.0.35 logo is visible at the bottom of the browser window.



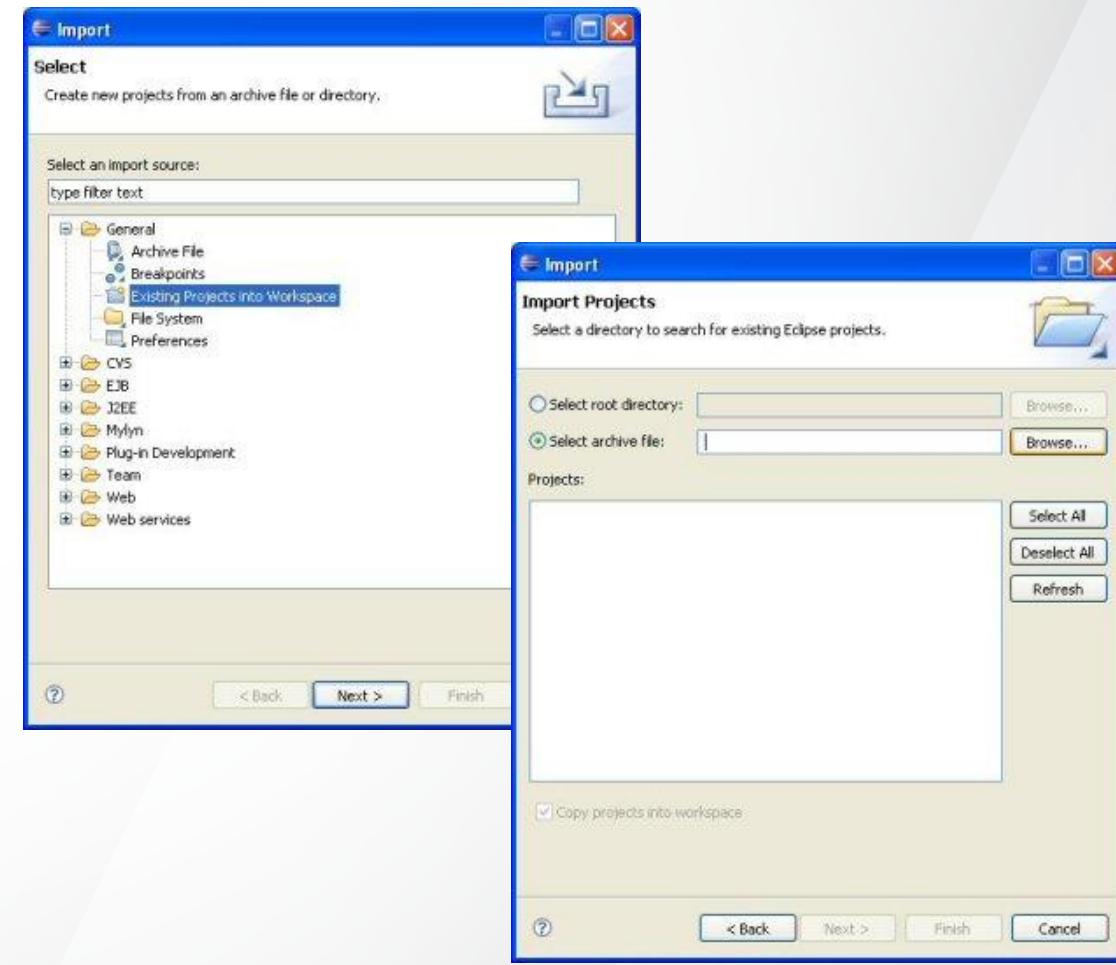
# Atelier 1 : Déploiement TOMCAT

- Copiez le répertoire root de tomcat se trouvant sous :  
`<TomcatInstallDir>\webapps\ROOT`
- vers le workspace de eclipse qui ressemble a ceci :
  - `<eclipseWorkSpace>\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps`
- Retestez l'URL



# Déploiement TOMCAT

- Dans eclipse Importez le fichier récupéré qui s'appelle
  - test-app8.zip
- Vérifiez le projet importé
- Déployez le projet
- Testez les liens suivants :
  - <http://localhost:8080/test-app/>
  - <http://localhost:8080/test-app/hello.html>
  - <http://localhost/test-app/hello.jsp>
  - <http://localhost/test-app/hello>
  - <http://localhost/test-app/test1>
  - <http://localhost/test-app/test2>



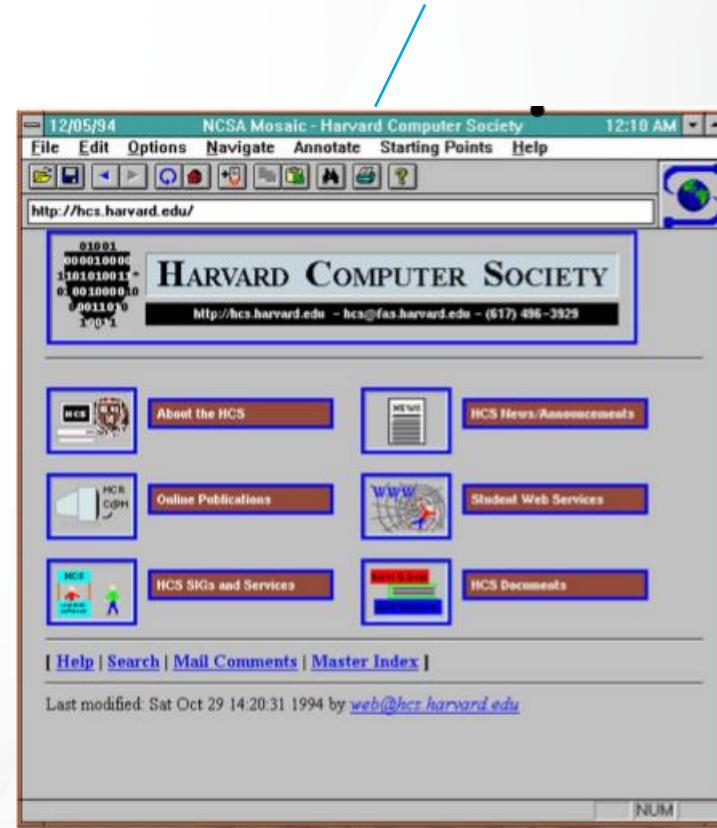
# Sommaire

- Chapitre 1 : INTRODUCTION
  - Chapitre 2 : PLATE-FORME JAVA EE
  - **Chapitre 3 : SERVLET / JSP**
  - Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE
  - Chapitre 5 : JSF
  - Chapitre 6 : PRIMEFACES
- 
- 

# Intro : Internet et HTTP

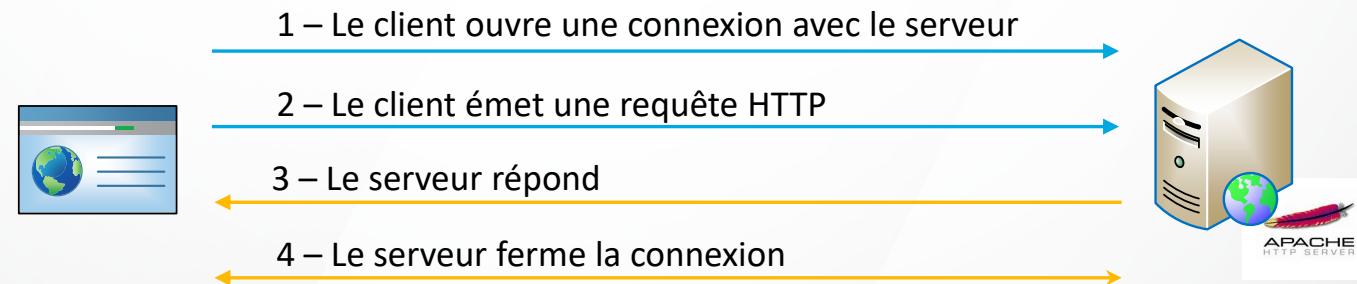
- A l'origine Internet a vocation de diffuser de l'information statique
  - HTTP (déconnecté)
  - HTML (langage de description de document)
- Déploiement
  - Universel (protocoles standards et réseau standard)
  - Un navigateur suffit
- Pas prévu pour embarquer des applications
  - Afficher des données en temps réel

Le premier navigateur pour Internet : Mosaic



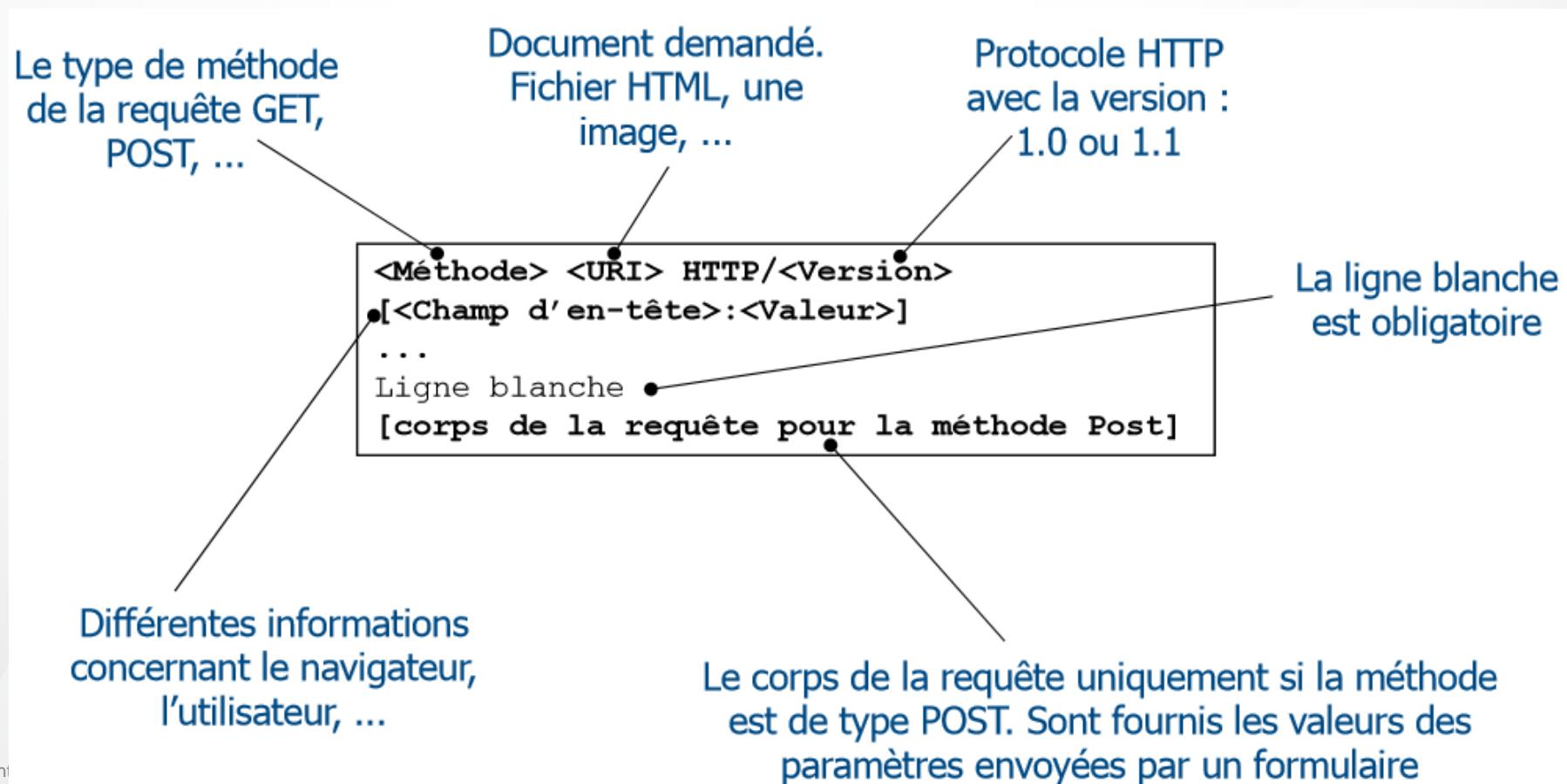
# Intro : Protocole HTTP

- Hyper Text Transfer Protocol
- Protocole Client/Serveur sans état
  - Impossibilité de conserver des informations issu du client
- La conversation HTTP est initialisée lorsque l'URL est saisie dans le navigateur



# Intro : Protocole HTTP : requête

## ➤ Requête envoyée par le client (navigateur) au serveur web



# Intro : HTTP : en-têtes de requête

- Correspond aux formats de documents et aux paramètres pour le serveur
  - Accept = type MIME visualisable par le client (text/html, text/plain, ...)
  - Accept-Encoding = codage acceptées (compress, x-gzip, x-zip)
  - Accept-Charset = jeu de caractères préféré du client
  - Accept-Language = liste de langues (fr, en, de, ...)
  - Authorization = type d'autorisation
    - BASIC nom:mot de passe (en base64)
    - Transmis en clair, facile à décrypter
  - Cookie = cookie retourné
  - From = adresse email de l'utilisateur

# Intro : HTTP : type de méthodes

- Lorsqu'un client se connecte à un serveur et envoie une requête, cette requête peut-être de plusieurs types, appelés **méthodes**
- Requête de type GET
  - Pour extraire des informations (document, graphique, ...)
  - Intègre les données de formatage à l'URL (chaîne d'interrogation)
    - www.exemple.com/hello?key1=titi&key2=raoul&...
- Requête de type POST
  - Pour poster des informations secrètes, des données graphiques, ...
  - Transmis dans le corps de la requête

```
<Méthode> <URI> HTTP/<Version>
[<Champ d'en-tête>:<Valeur>]
...
Ligne blanche
[corps de la requête pour la méthode Post]
```

# Intro : HTTP : réponse

- Correspond aux informations concernant le serveur WWW
  - Accept-Range = accepte ou refus d'une requête par intervalle
  - Age = ancienneté du document en secondes
  - Set-Cookie = créé ou modifie un cookie sur le client
  - WWW -Authenticate = système d'authentification. Utiliser en couple avec l'en-tête requête Authorization

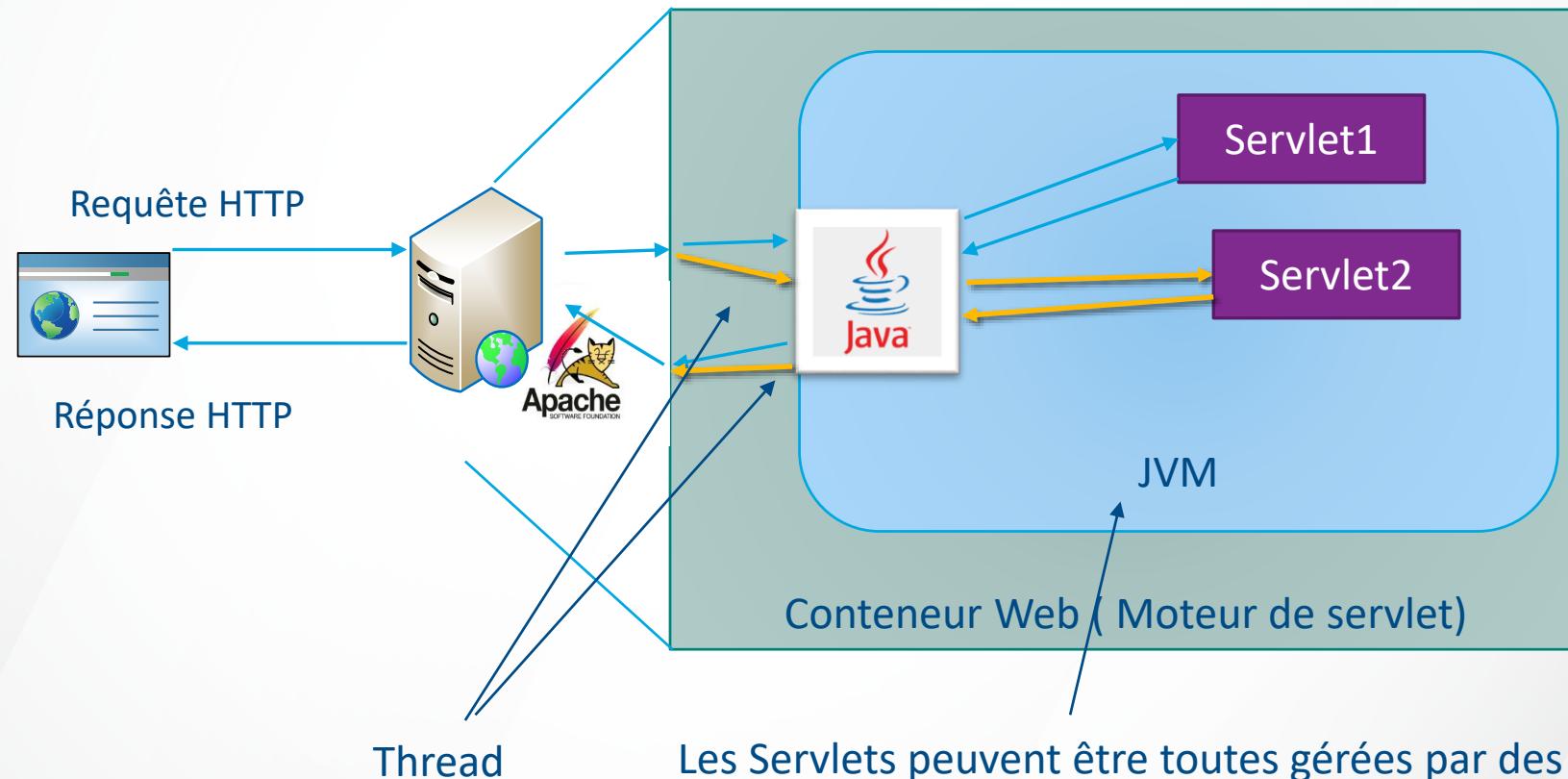
# Intro : HTTP : status des réponses

- Réponse du serveur au client <Status><Commentaire>
  - 100-199 : Informationnel
    - **100** : Continue (le client peut envoyer la suite de la requête), ...
  - 200-299 : Succès de la requête client
    - **200** : OK, **204** : No Content (pas de nouveau corps de réponse)
  - 300-399 : Re-direction de la requête client
    - **301** : Redirection, **302** : Moved Temporarily
  - 400-499 : Erreur client
    - **401** : Unauthorized, **404** : Not Found (ressource non trouvée)
  - 500-599 : Erreur serveur
    - **503** : Service Unavailable (serveur est indisponible)

# Intro : Qu'est ce qu'une Servlet

- Composant logiciel écrit en Java fonctionnant du côté serveur
- Au même titre nous trouvons
  - CGI (Common Gateway Interface)
  - Langages de script côté serveur PHP, ASP (Active Server Pages)
- Permet de gérer des requêtes HTTP et de fournir au client une réponse HTTP
- Une Servlet s'exécute dans un **moteur de Servlet** ou **conteneur de Servlet** permettant d'établir le lien entre la Servlet et le serveur Web
- Une Servlet s'exécute par l'intermédiaire d'une machine virtuelle

# Intro : Architecture d'une Servlet



# Servlet, à quoi ça sert ?

- Créer des pages HTML dynamiques, générer des images, ...
- Effectuer des tâches de type CGI qui sont des traitements applicatifs côté serveur WEB
  - Manipulation d'une base de données
  - Gestion d'un système de surveillance,...
- Respecter les principes d'une architecture : écrire une application en Java dont l'interface utilisateur est dans le client
  - Applet(SWING)
  - Téléphone portable
  - Navigateur(HTML)

# Puissance des Servlets

- Portabilité
  - Technologie indépendante de la plate-forme et du serveur
  - Un langage (Java) et plusieurs plate-forme (.NET plusieurs langages et une plate-forme)
- Puissance
  - Disponibilité de l'API de Java
  - Manipulation d'images, connectivité aux bases de données (JDBC), ...
- Efficacité et endurance
  - Une Servlet est chargée une seule fois (CGI chargée puis déchargée après utilisation)
  - Une Servlet conserve son état (connexions à des bases de données)
- Sûreté
  - Typage fort de Java
  - Gestion des erreurs par exception

# API : Servlet

- Une Servlet doit implémenter l'interface `javax.servlet.Servlet` et `javax.servlet.ServletConfig`

`Servlet << Interface >>`

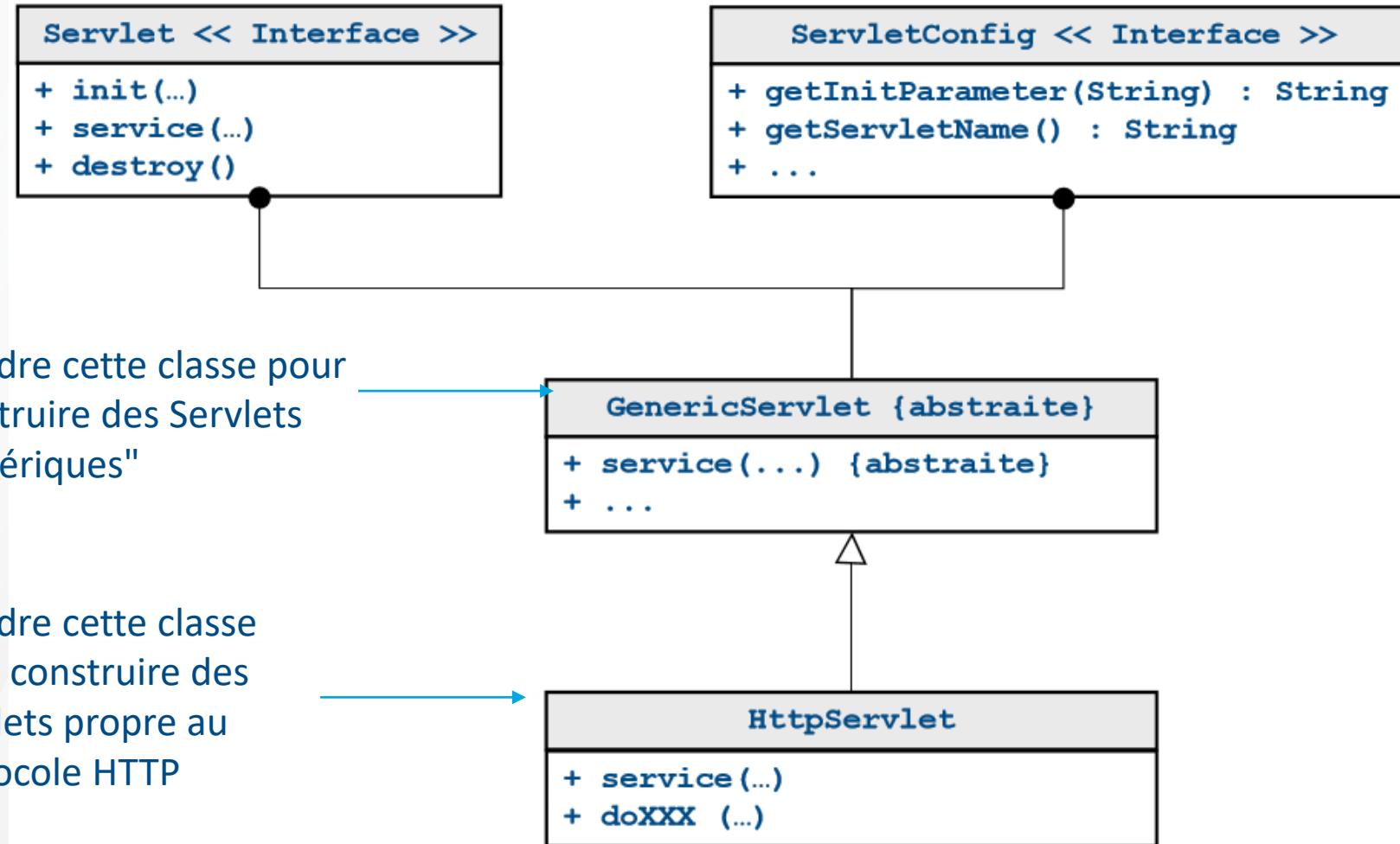
- + `init(...)`
- + `service(...)`
- + `destroy()`

`ServletConfig << Interface >>`

- + `getInitParameter(String) : String`
- + `getServletName() : String`
- + ...

- Plus simplement l'API Servlet fournit deux classes qui proposent déjà une implémentation
  - `GenericServlet` : pour la conception de Servlets indépendantes du protocole
  - `HttpServlet` : pour la conception de Servlets spécifiques au protocole HTTP

# API : Servlet (2)

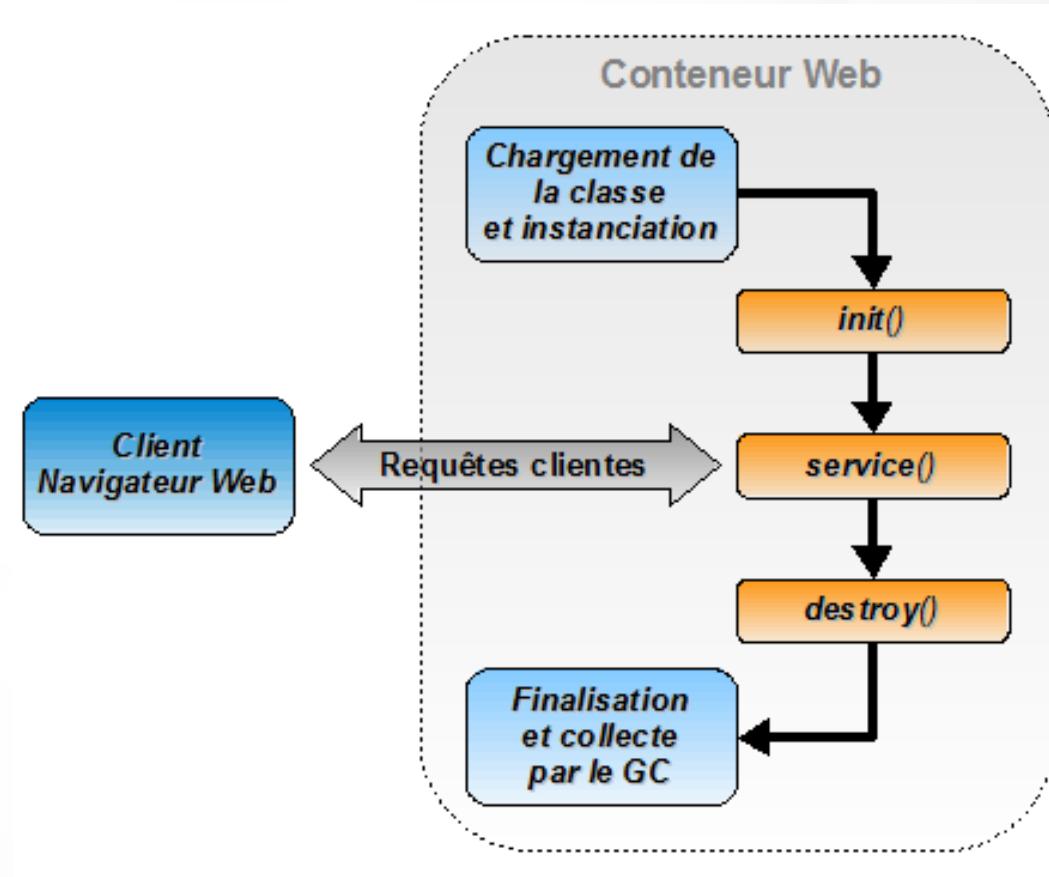


# Cycle de vie

- Le cycle de vie d'une servlet est assuré par le conteneur de servlet.
- Ainsi afin d'être à même de fournir la requête à la servlet, récupérer la réponse ou bien tout simplement démarrer/arrêter la servlet, cette dernière doit posséder une interface (un ensemble de méthodes prédéfinies voir slide précédent) déterminée par le JSDK afin de suivre le cycle de vie suivant :

# Cycle de vie

- 1. Chargement de la classe
- 2. Instanciation du servlet
  - constructeur par défaut
- 3. Appel de init()
- 4. Appel(s) de service()
  - 1 thread par requête
- 5. Appel de destroy()



# Cycle de vie (2)

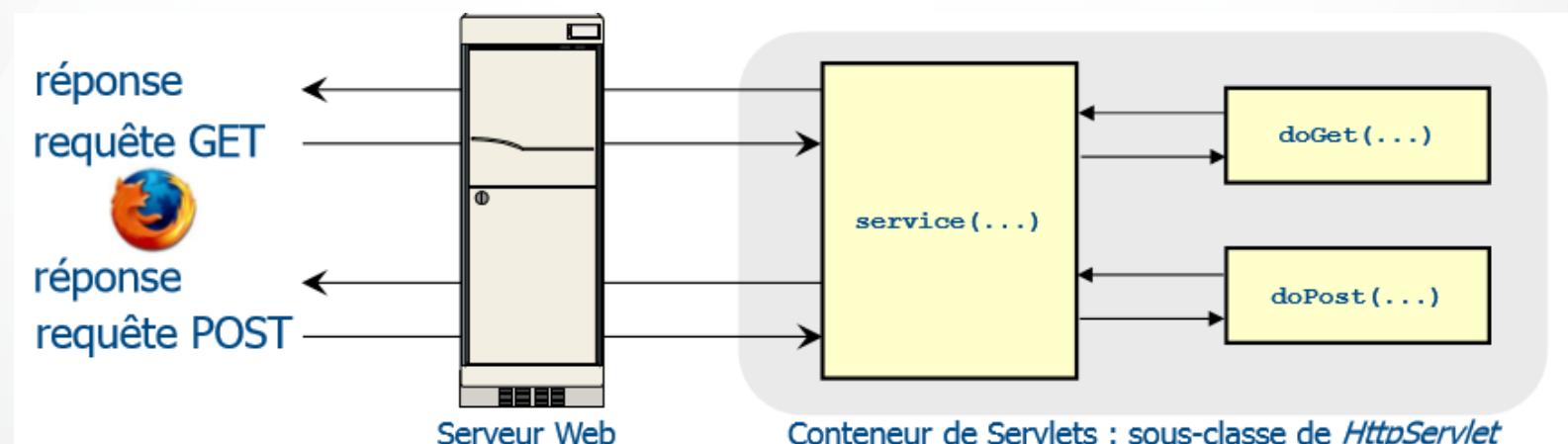
- – La servlet classe est chargée :  
Le chargeur de classe est responsable de charger la classe de servlet . La classe servlet est chargée lorsque la première demande de la servlet est reçue par le conteneur Web.
- – Création de l'instance Servlet :  
Le conteneur Web crée l'instance d'une servlet après le chargement de la classe de servlet . L'instance servlet est créée seulement une fois dans le cycle de vie de la servlet.
- – La méthode `Init()` :  
le conteneur appelle la méthode `Init()` une seule fois après la création de l'instance servlet et crée les objets `Request` et `Response` spécifiques à la requête

# Cycle de vie (3)

- La méthode service()
  - Lors de la réception d'une requête, le conteneur crée:
    - un objet ServletRequest (la requête)
    - un objet ServletResponse (la réponse)
  - Le conteneur appelle ensuite la méthode `service()` avec ces deux objets en paramètres pour permettre au servlet de répondre à la requête du client.
  - Chaque requête est dans une nouvelle thread
- La méthode Destroy() :
  - La méthode `destroy()` est appelée juste avant que le serveur ne détruise la servlet, cela donne l'occasion de libérer des ressources.

# L'API Servlet : la classe HttpServlet

- Dans la suite du cours nous allons utiliser uniquement des Servlets qui réagissent au protocole HTTP d'où l'utilisation de la classe HttpServlet
- HttpServlet redéfinit la méthode service(...)
  - service(...) lit la méthode (GET, POST, ...) à partir de la requête
  - Elle transmet la requête à une méthode appropriée de HttpServlet destinée à traiter le type de requête (GET, POST, ...)



# HttpServlet : méthodes de traitement de requêtes

- Plusieurs méthodes sont fournies pour traiter les différents types de requêtes (GET, POST, ...).
- Elles sont appelées **méthodes de traitement de requêtes**
- Elles ont un en-tête identique **doXXX(...)** où XXX correspond au type de requête
  - doPost(...) est la méthode pour traiter les requêtes de type POST
  - doGet(...) est la méthode pour traiter les requêtes de type GET
  - doHead(...) , doTrace(...) , ...
- Selon le type de requête (GET ou POST) le concepteur redéfinit la méthode concernée

# HttpServlet : objet requête

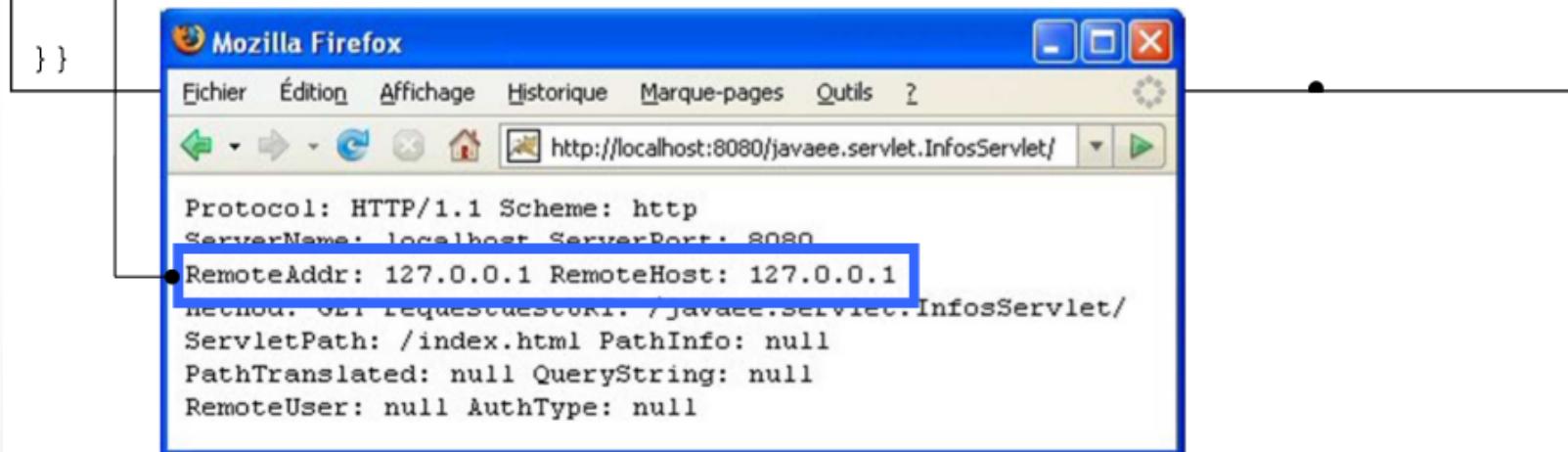
## HttpServletRequest

- HttpServletRequest hérite de ServletRequest
- Cet objet encapsule la requête HTTP et fournit des méthodes pour accéder aux informations
  - du client
  - de l'environnement du serveur
- Exemples de méthodes
  - String getMethod() : retourne le type de requête
  - String getServerName() : retourne le nom du serveur
  - String getParameter(String name) : retourne la valeur d'un paramètre
  - String[] getParameterNames() : retourne le nom des les paramètres
  - String getRemoteHost() : retourne l'IP du client
  - String getServerPort() : retourne le port sur lequel le serveur écoute
  - String getQueryString() : retourne la chaîne d'interrogation

# HttpServlet : objet requête HttpServletRequest

- Exemple : Servlet qui affiche un certains nombre d'informations issues de HttpServletRequest

```
public class InfosServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 response.setContentType("text/plain");
 PrintWriter out= response.getWriter();
 out.println("Protocol: " + request.getProtocol());
 out.println("Scheme: " + request.getScheme());
 out.println("ServerName: " + request.getServerName());
 out.println("ServerPort: " + request.getServerPort());
 out.println("RemoteAddr: " + request.getRemoteAddr());
 out.println("RemoteHost: " + request.getRemoteHost());
 out.println("Method: " + request.getMethod());
 } }
```



# HttpServletRequest : objet réponse

## HttpServletResponse

- HttpServletResponse hérite de ServletResponse
- Cet objet est utilisé pour construire un message de réponse HTTP renvoyé au client, il contient
  - les méthodes nécessaires pour définir le type de contenu, en-tête et code de retour
  - un flot de sortie pour envoyer des données (par exemple HTML) au client
- Exemples de méthodes
  - void setStatus (int) : définit le code de retour de la réponse
  - void setContent-Type (String) : définit le type de contenu MIME
  - ServletOutputStream getOutputStream () : flot pour envoyer des données binaires au client
  - void sendRedirect (String) : redirige le navigateur vers l'URL

# HttpServlet : objet réponse

## HttpServletResponse

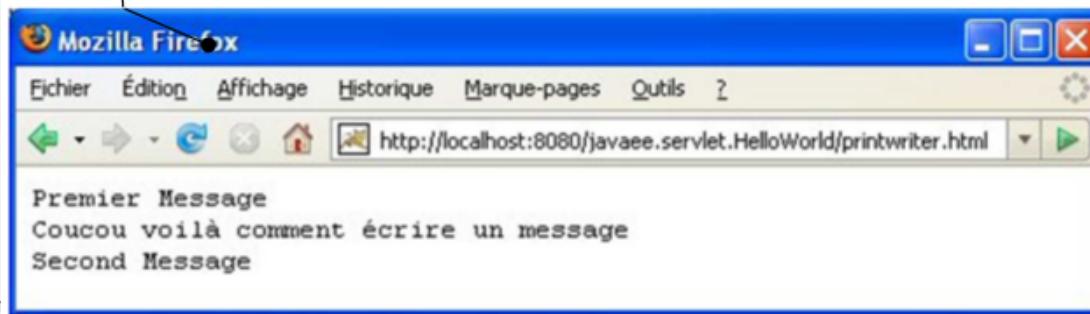
- Exemple : écriture d'un message de type TEXT au client

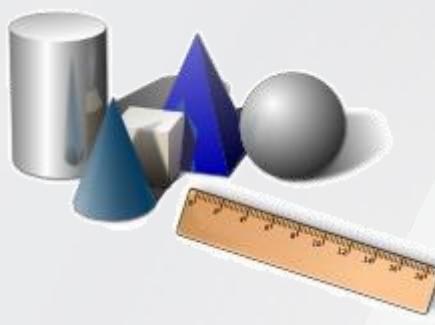
```
public class HelloWorldPrintWriter extends HttpServlet {

 public void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/plain");

 PrintWriter out = res.getWriter();

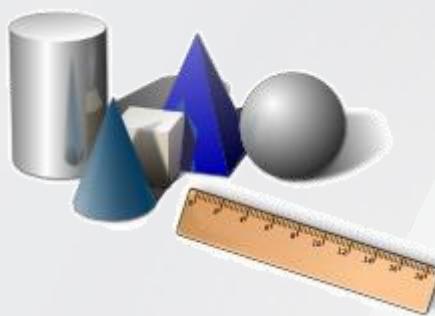
 out.println("Premier Message");
•out.println("Coucou voilà comment écrire un message");
 out.println("Second Message");
 }
}
```





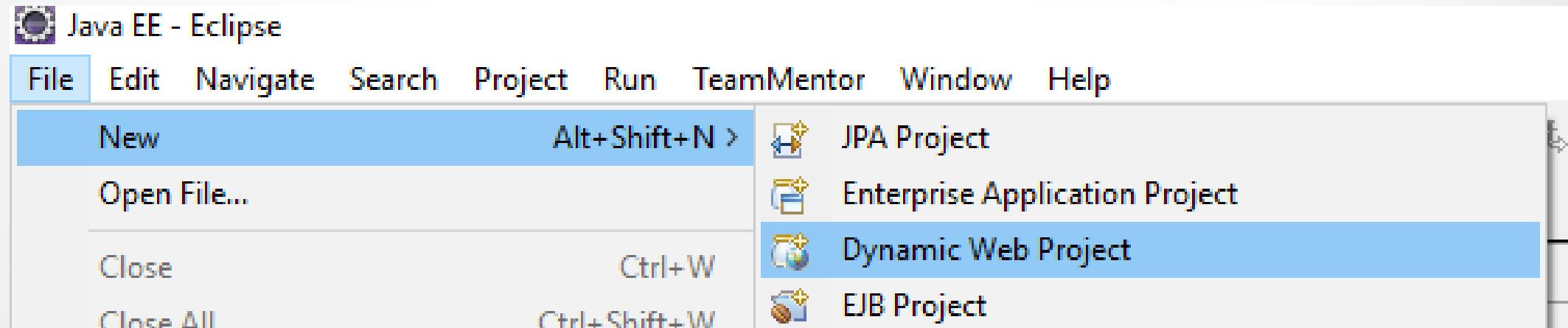
# Workshop 1 : Création d'un projet web sous eclipse

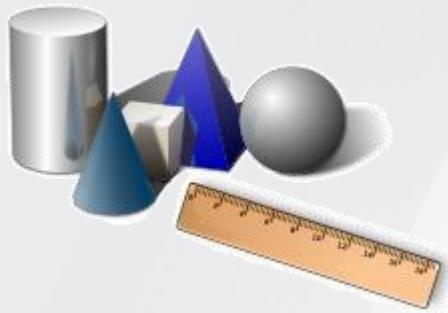
- Un projet Web est un type de projet spécial sous Eclipse.
  - Spécial, en ce sens qu'il est attaché à un serveur d'applications (ici Tomcat, ce pourrait être un autre serveur),
  - et qu'il expose des fonctionnalités particulières pour créer des servlets, des pages, ou des services web (que nous ne présenterons pas ici).
- La création d'un projet Web débute comme un projet Eclipse normal.
  - Comme il ne s'agit pas d'un projet Java classique, on sélectionne la rubrique *Project...* du sous-menu *New*.



# Workshop 1 : Création d'un projet web sous eclipse

- Dans le panneau qui s'ouvre, on sélectionne alors *Dynamic Web Project*, sous le nœud *Web*.

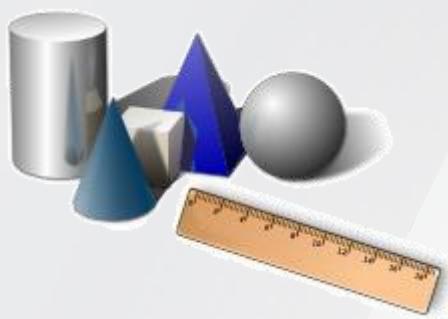




# Workshop 1 : Création d'un projet web sous eclipse

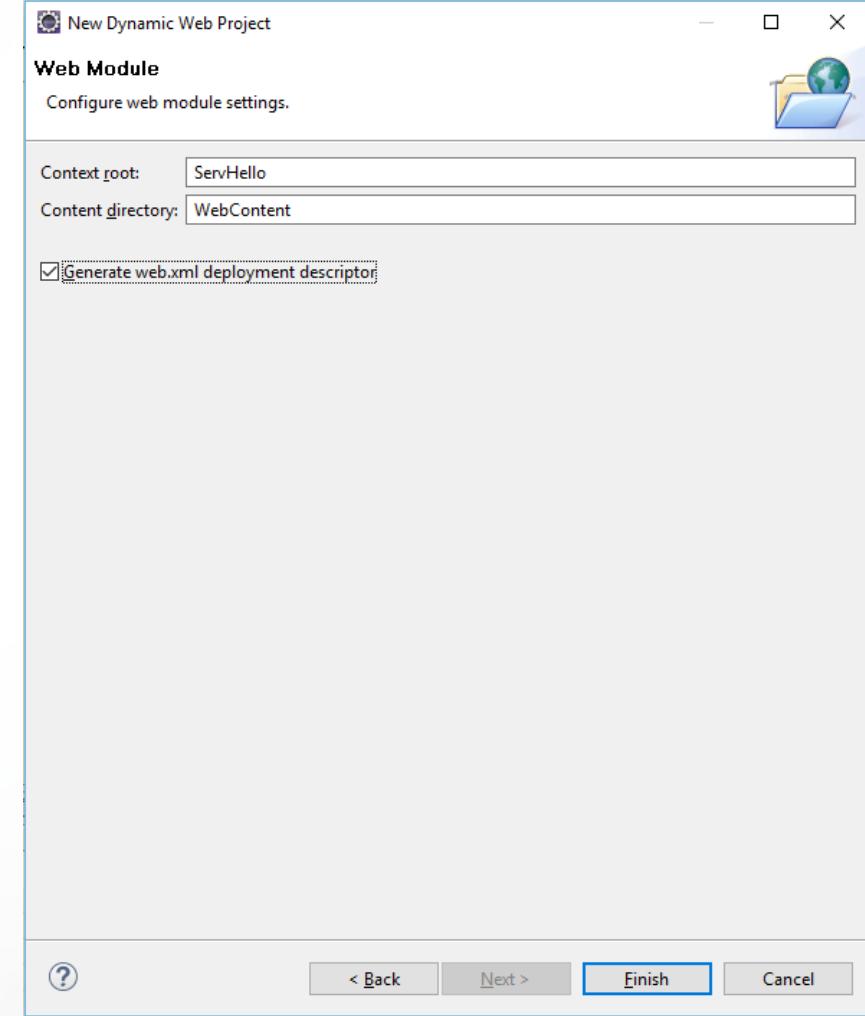
- Le panneau qui s'ouvre alors est un peu plus compliqué que celui d'un projet Java classique.
- En particulier, il nous demande de choisir le serveur que ce projet va utiliser pour fonctionner.
- Dans notre exemple, on sélectionne le serveur Tomcat que l'on vient de créer.
- La version de l'API Servlet est sélectionnée automatiquement : 3.1, version supportée par Tomcat v9.
- Le panneau suivant nous demande dans quel répertoire on souhaite ranger nos fichier source. On conserve le choix par défaut : le répertoire src.

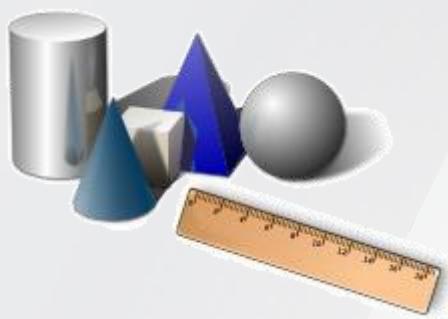
The screenshot shows the Eclipse IDE interface. On the left, the 'New Dynamic Web Project' dialog is open, prompting for project details like name, location, runtime, and configuration. The project name is set to 'ServHello'. The target runtime is 'Apache Tomcat v8.0'. The dynamic web module version is '3.1'. The configuration is 'Default Configuration for Apache Tomcat v8.0'. On the right, the Java perspective is visible, showing a 'src' folder in the package explorer and a 'Default output folder: build\classes' setting in the properties view.



# Workshop 1 : Création d'un projet web sous eclipse

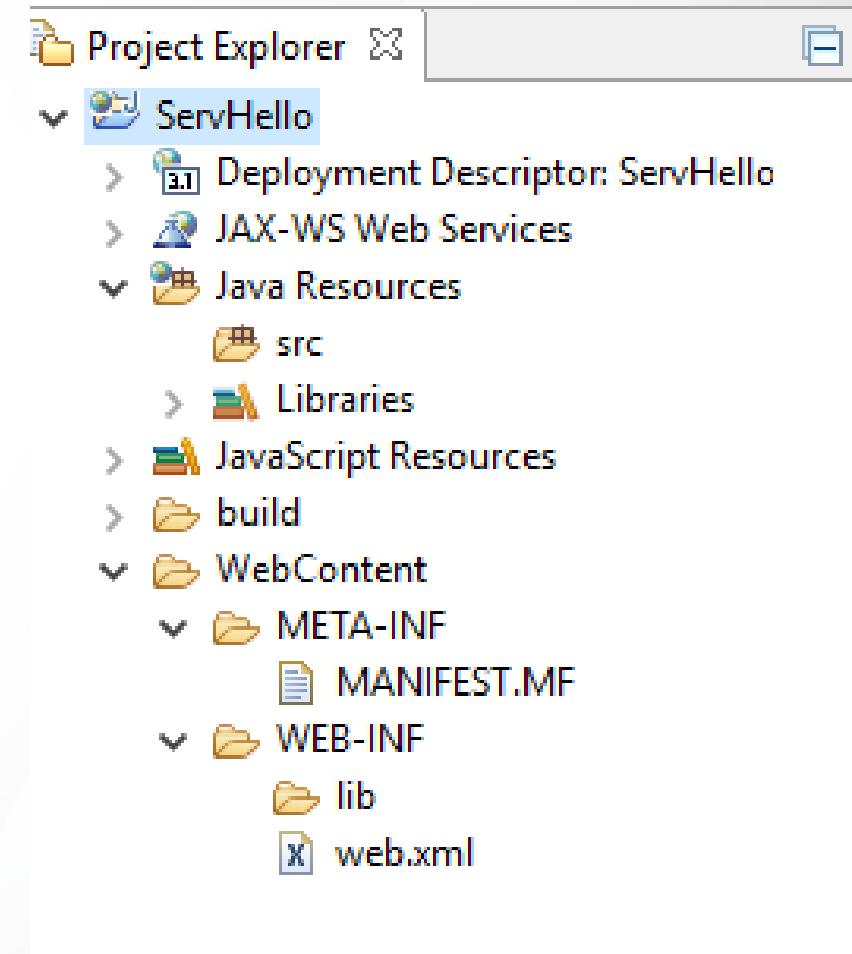
- Le dernier panneau nous demande dans quel répertoire on souhaite ranger les ressources de ce projet web :
  - le répertoire **WebContent**. C'est dans ce répertoire que l'on pourra ranger les différentes ressources de notre projet : pages HTML, tout fichier multimedia, fichier Javascript, etc...
  - On peut également générer automatiquement le fichier **web.xml** de notre application web, ce qui une commodité offerte par Eclipse.
    - Ce fichier sera rangé automatiquement dans **WebContent/WEB-INF**.

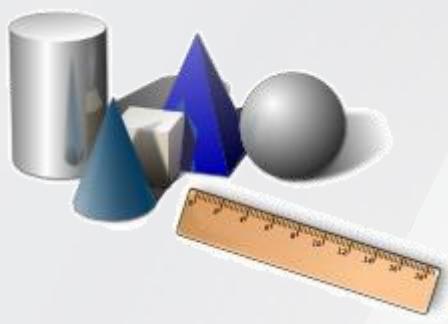




# Workshop 1 : Création d'un projet web sous eclipse

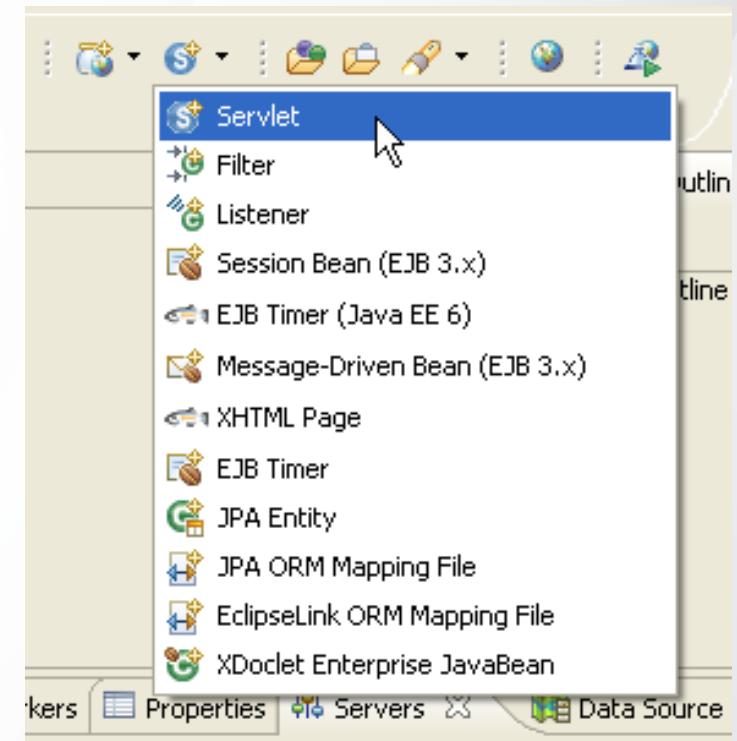
- Une fois notre projet créé, Eclipse nous demande si l'on souhaite utiliser la *perspective Java EE*. Cette perspective est bien adaptée au développement des projets Java EE (elle est même faite pour ça !), on accepte donc cette proposition.
- Sur la partie gauche de cette perspective, se trouve la vue *Project Explorer* (qui ressemble beaucoup à *Package Explorer*), dans laquelle notre nouveau projet apparaît.
- → Notre projet est maintenant créé.

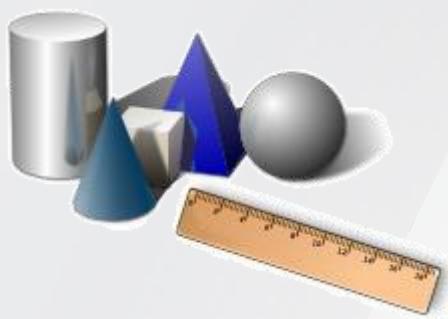




# Workshop 1 : Création d'un projet web sous eclipse

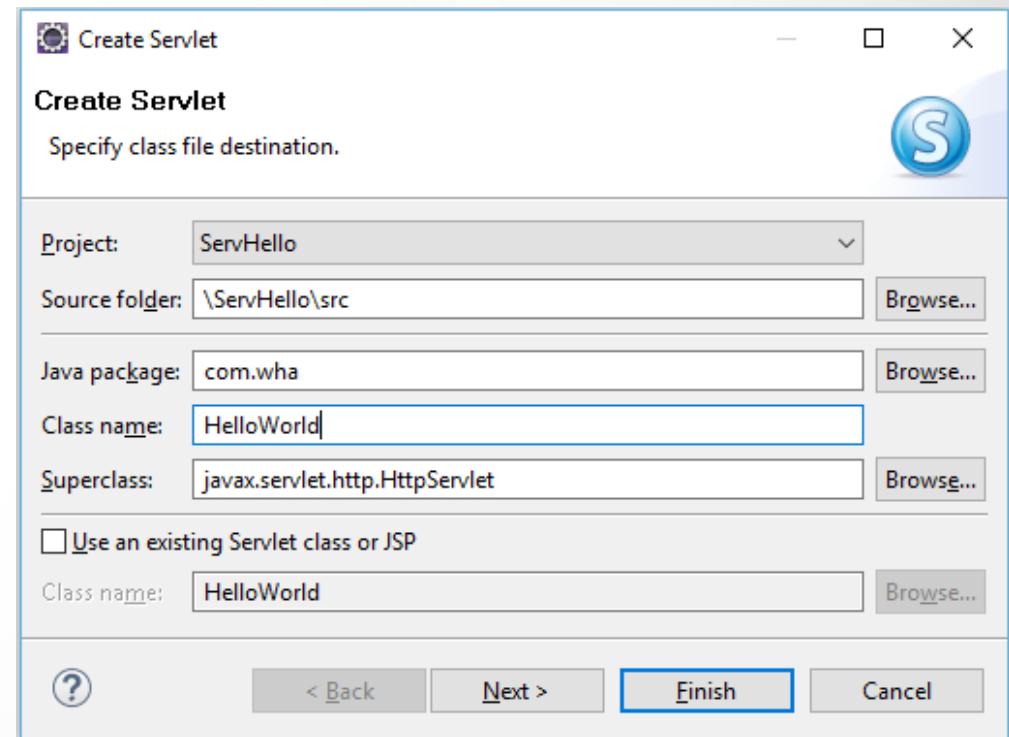
- La perspective Java EE nous montre une frise d'icône qui nous permet de créer directement des classes Java de servlet. C'est que nous allons utiliser pour notre première servlet.
- La première rubrique du menu qui s'ouvre est celle qui nous permet de créer une servlet.

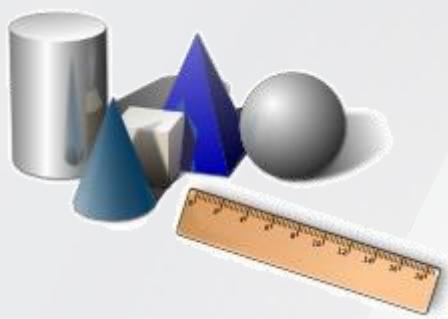




# Workshop 1 : Création d'un projet web sous eclipse

- Un panneau s'ouvre alors, qui nous permet de spécifier la classe de cette servlet : son nom, et le nom du package dans lequel cette classe sera placée.
- Une servlet étend nécessairement HttpServlet, on ne doit donc prendre garde à ce point si l'on modifie la classe qu'étend notre servlet.

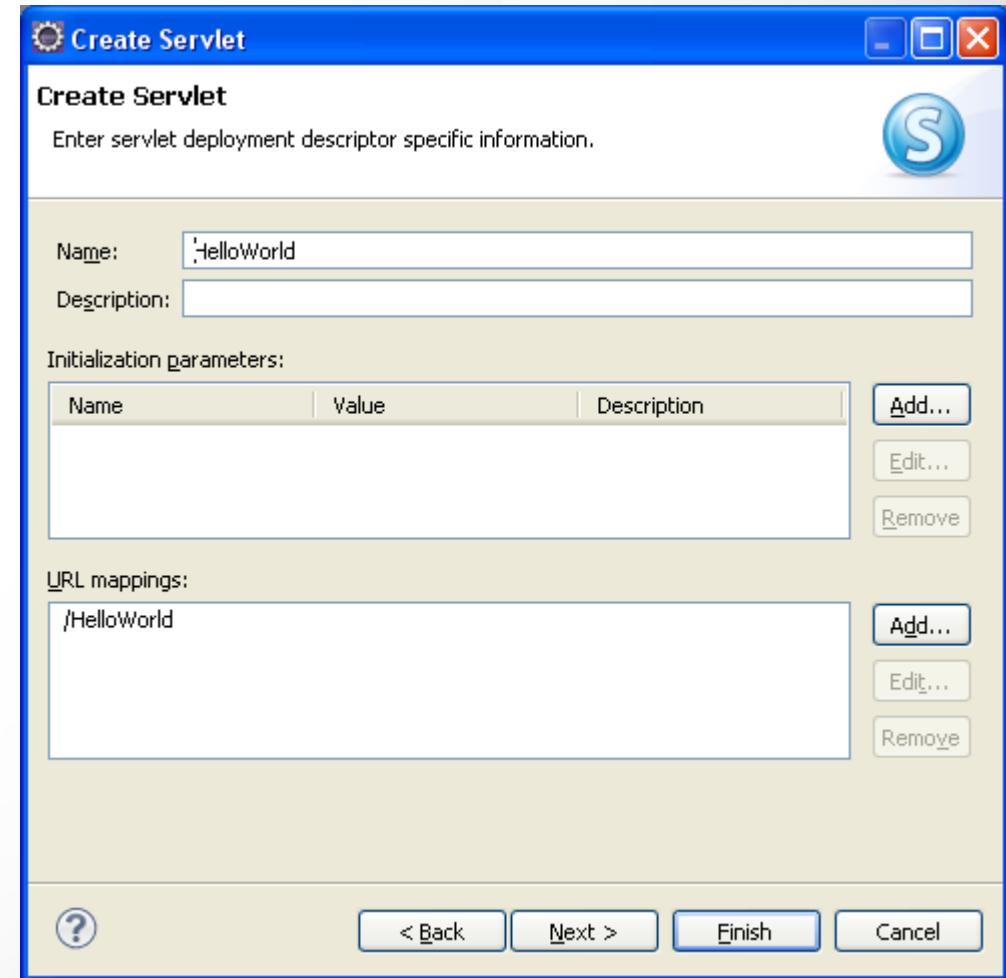


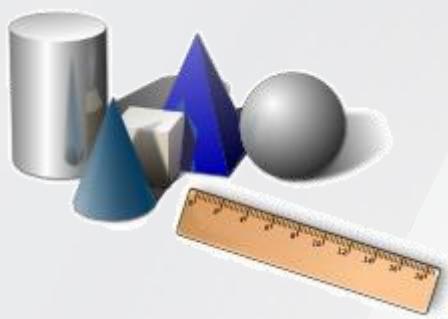


# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

Le panneau suivant nous permet de fixer les informations sur cette servlet, qui seront écrites dans le fichier `web.xml` de notre application.

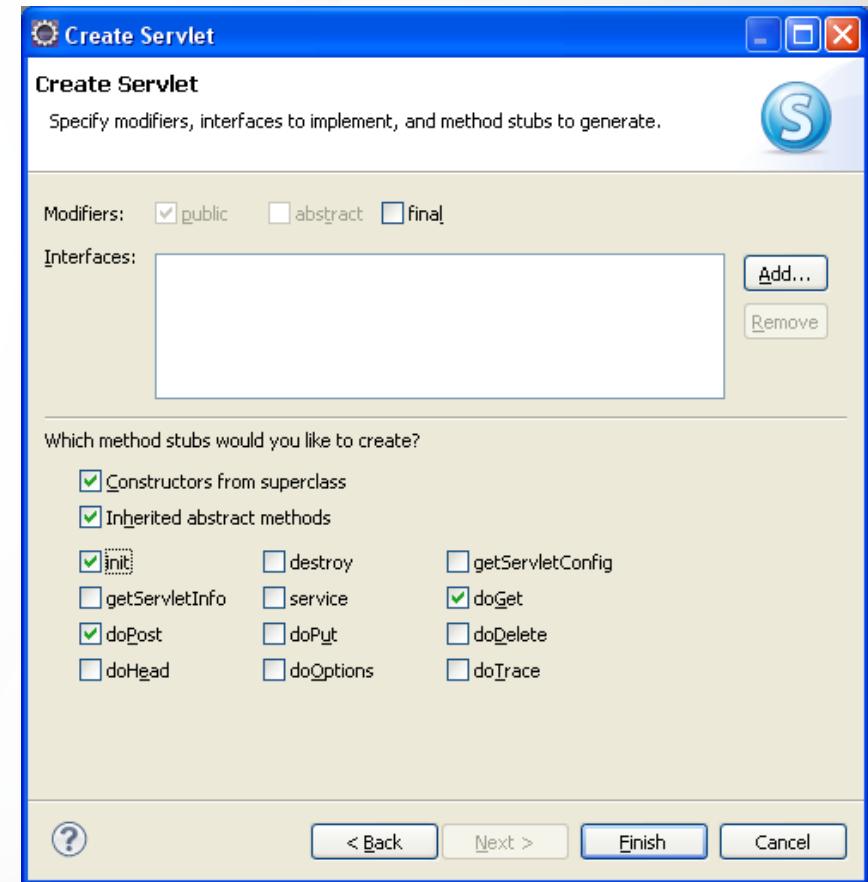
- Le nom de notre servlet, utilisé pour s'y référer dans le fichier `web.xml`.
- Ses paramètres d'initialisation.
- Le (ou les) paramètres *URL mappings*, qui fixeront l'URL à laquelle notre servlet sera disponible.

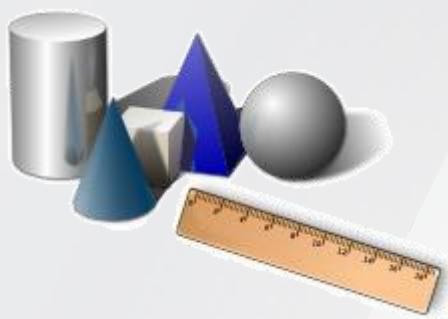




# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

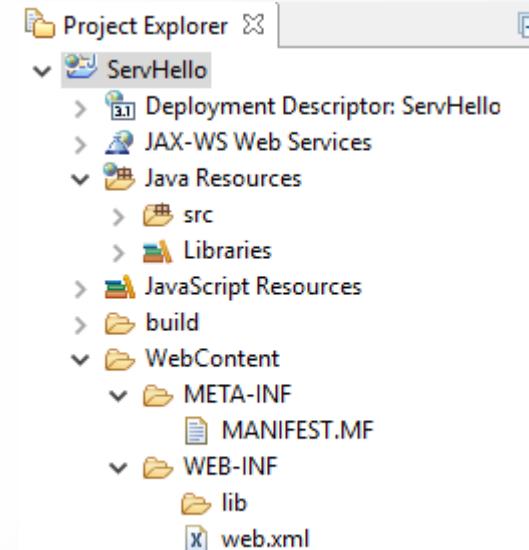
- Le dernier panneau de création précise le contenu de la classe qu'Eclipse va nous générer.
- Par commodité, Eclipse peut créer pour nous différentes méthodes standard de notre servlet, telles que **init()**, **doGet()** ou **doPost()**.



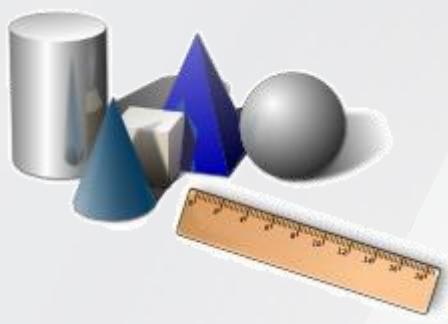


# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

- Une fois ces étapes déroulées, notre classe de servlet est créée, et apparaît dans la structure de notre projet
- Le fichier web.xml a bien été modifié, et prend bien en compte la servlet qui vient d'être créée.



A partir de JEE 6 (Servlet 3.0), le déclaration d'une servlet peut se faire dans la servlet avec l'annotation **@WebServlet**



# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

- Notre servlet est très simple, son code est le suivant.

```
/*
 * Servlet implementation class HelloWorld
 */
@WebServlet("/HelloWorld")
public class HelloWorld extends HttpServlet {
 private static final long serialVersionUID = 1L;

 protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 PrintWriter pw = response.getWriter();
 pw.write("Hello world !");

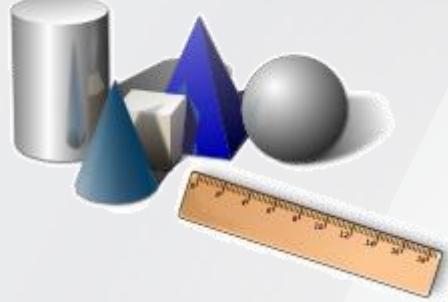
 }

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

 public HelloWorld() {
 super();
 }

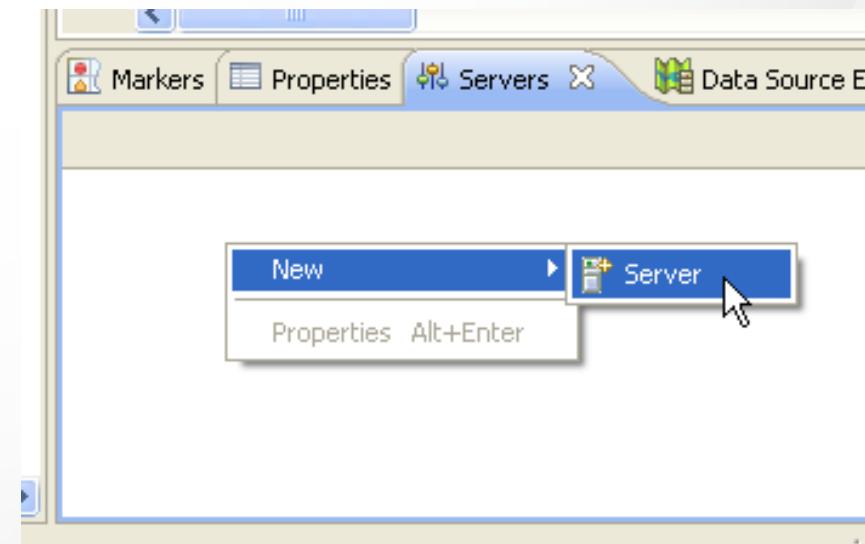
 public void init(ServletConfig config) throws ServletException {
 }
}
```

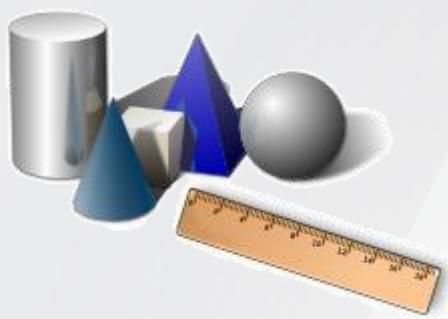
- Une fois cette classe créée, il nous reste à la faire fonctionner dans Tomcat, ce qui est l'objet de la prochaine étape.



# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

- Une servlet doit être exécutée dans un serveur d'applications, dans notre exemple il s'agit de Tomcat. La première étape consiste à créer une instance de serveur Tomcat, associée à notre espace de travail.
- La création d'une instance de Tomcat (comme de tout autre serveur d'application) peut se faire directement dans la vue Servers, présente dans le bas de la perspective Java EE. Le menu contextuel de cette vue permet de créer cette nouvelle instance.



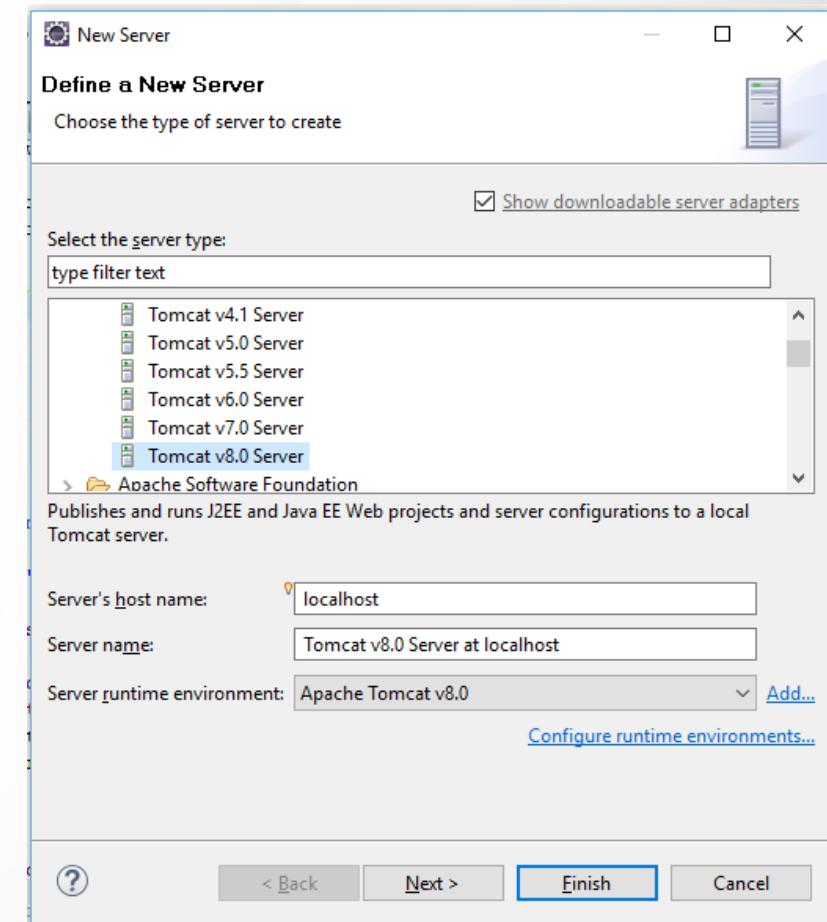


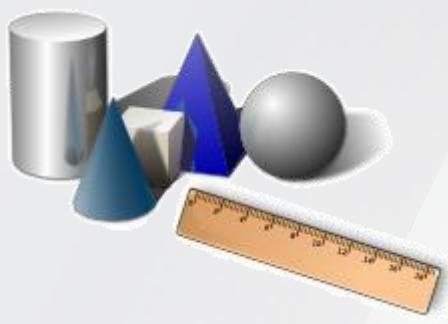
# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

Le panneau qui s'ouvre alors permet de sélectionner le serveur qui va être créé.

Pour cela, il faut renseigner les différents éléments du panneau.

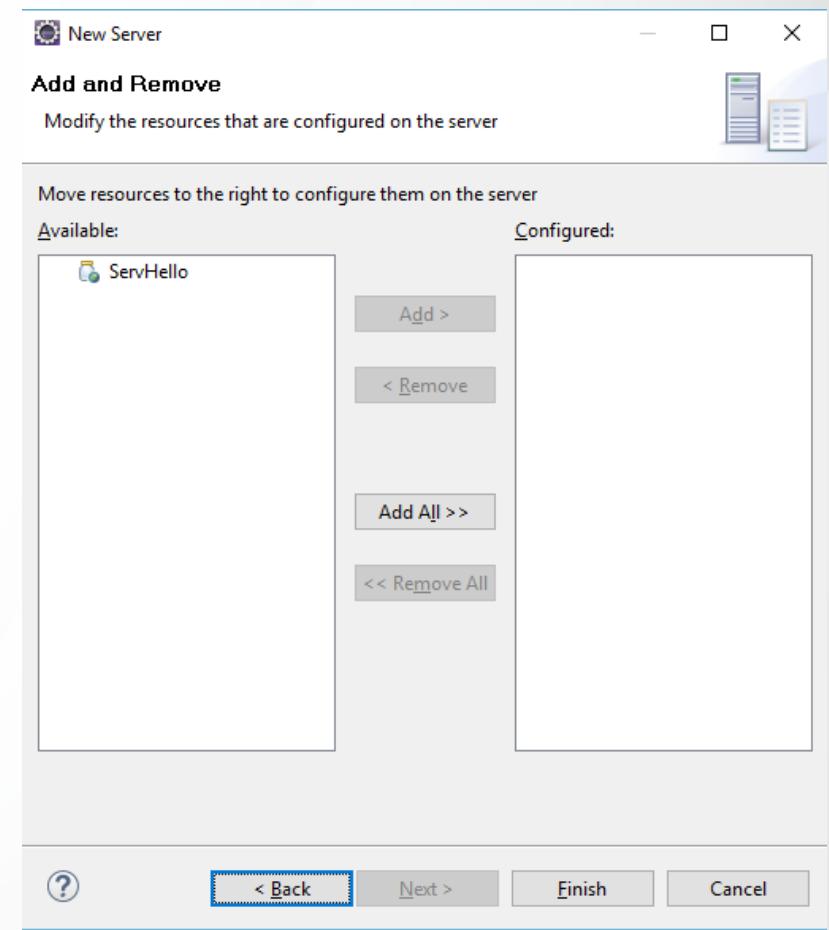
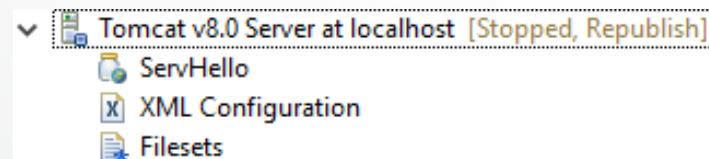
- Le type de serveur : ici Tomcat v6.
- Le *Server host name* : ici localhost.
- Le *Server name* : ici Tomcat v8.0 Server at localhost. Ce nom est un nom logique utilisé par Eclipse.
- Le *Server runtime environment* : c'est ici que l'on sélectionne le Tomcat que l'on a installé précédemment.
- On peut aussi installer de nouveaux serveur en cliquant sur le lien *Add...* qui se trouve à côté de ce sélecteur.

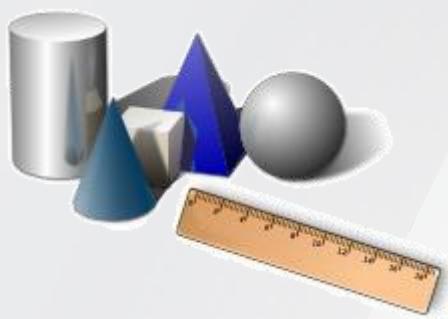




# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

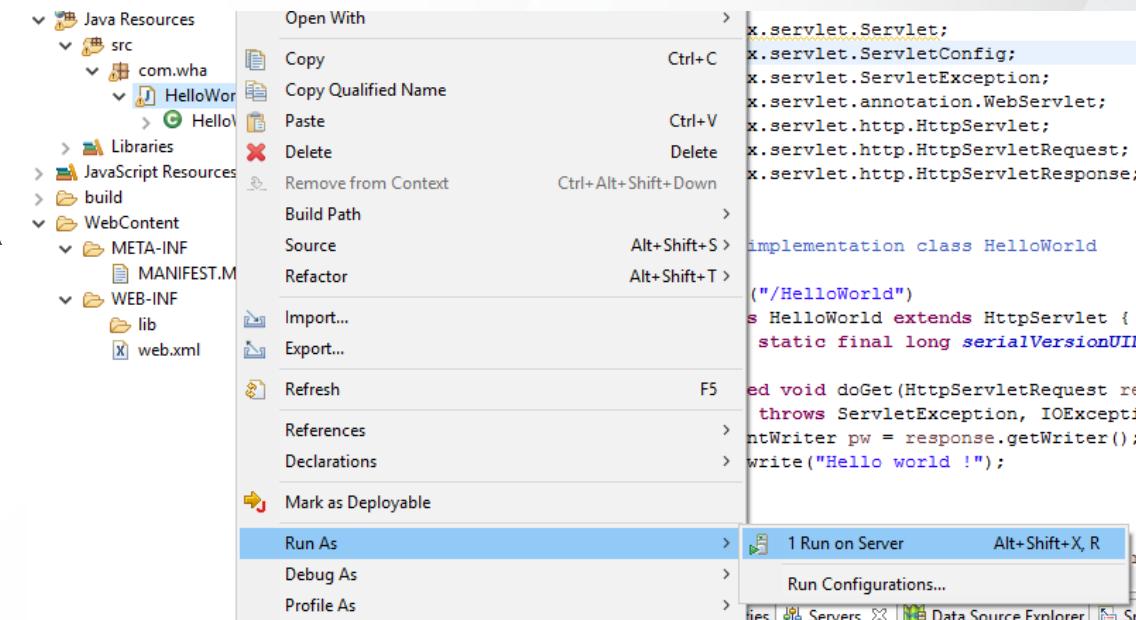
- Une fois l'instance de serveur créé, il faut lui ajouter des projets.
  - Ici nous avons créé un serveur Tomcat, capable d'exécuter des servlets, on peut donc lui ajouter des projets qui contiennent des servlets. Eclipse ne nous propose ici que les projets web. Si nous avions d'autres projets Java dans notre espace de travail, ils n'apparaîtraient pas dans cette liste.
- Cette étape était la dernière : notre serveur est maintenant créé, et notre projet web lui a été ajouté. Notons que l'on peut aussi ajouter des projets à un serveur en les glissant / déposant à partir de la vue *Project explorer*.
- Notre vue *Servers* a maintenant l'allure suivante.

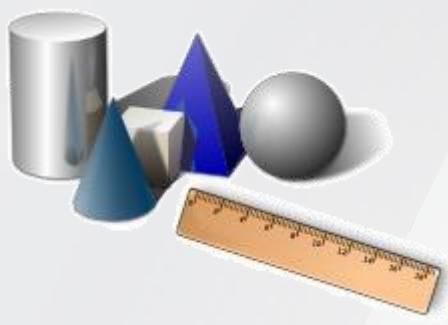




# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

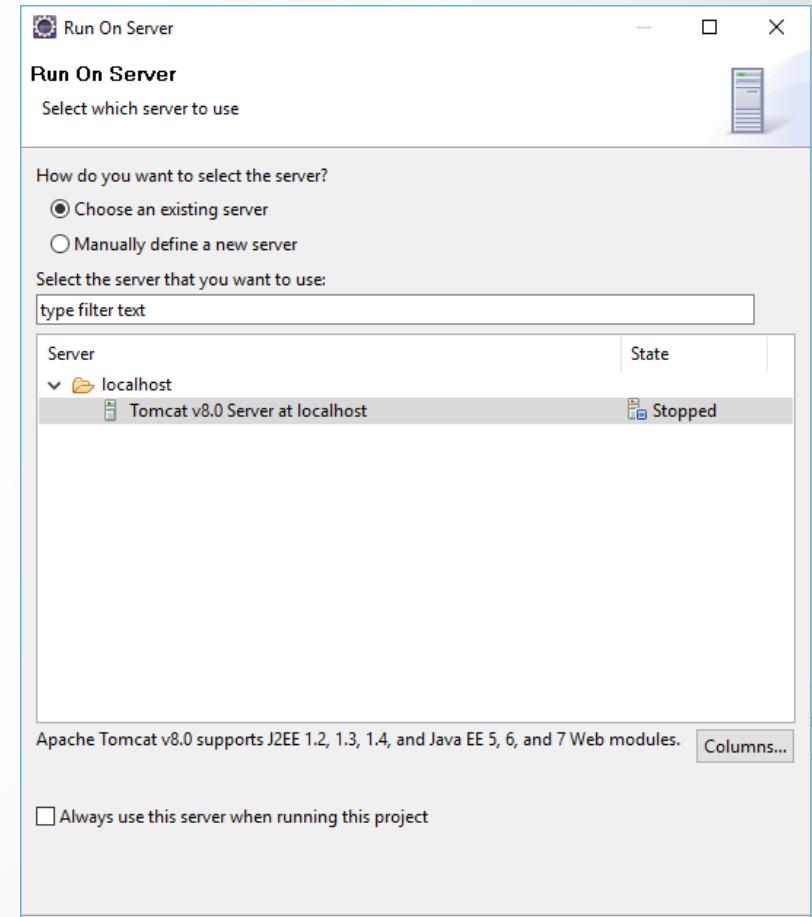
- Lorsque toute la configuration précédente a été effectuée, l'exécution d'une servlet devient assez simple.
- Il suffit de sélectionner la traditionnelle option *Run as...* du menu contextuel de la casse que l'on veut exécuter. Comme cette classe ne comporte pas de méthode *main()*, la rubrique *Java Application* n'est pas présente. En revanche, nous avons la rubrique *Run on Server*, que nous allons sélectionner.

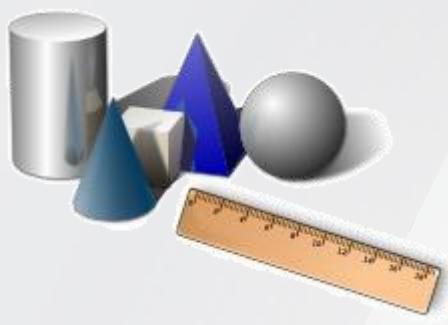




# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

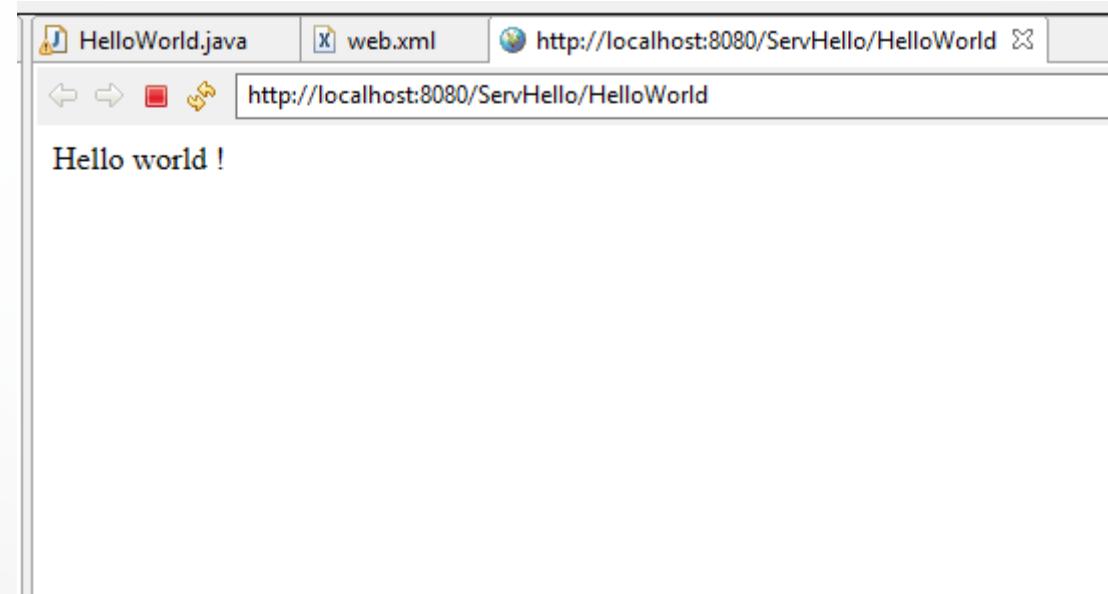
- Eclipse nous demande alors quel serveur va exécuter cette servlet. Ici nous n'en avons qu'un, dans une configuration complexe, il pourrait y en avoir plusieurs.





# Workshop 2 : CRÉATION D'UNE PREMIÈRE SERVLET

- Eclipse nous ouvre enfin une vue *Navigateur*, sur l'URL associée à notre servlet. Cette URL est composée de deux éléments principaux : l'hôte sur lequel on a installé notre serveur, et l'élément d'URL auquel est associée notre servlet. Le port choisi par Eclipse est le port 8080, par défaut. Nous verrons dans la suite comment le changer.



## ➤ MISE EN OEUVRE DE SERVLETS



# Servlets et formulaires : du côté HTML

- Utilisation de la balise <FORM> </FORM>
  - Option METHOD : type de requête GET ou POST
  - Option ACTION : l'URL où envoyer les données
- Utilisation de composants IHM pour saisir des informations
  - Contenu à l'intérieur de la balise FORM

The screenshot shows a portion of a web page with a form. At the top, there is a text area containing the text "blabla". Below it is a text input field containing the number "0549498070". To the right of these are two buttons: a checked checkbox labeled "case à cocher" and an unchecked radio button labeled "bouton radio". Further to the right is a dropdown menu with the text "Element 1" and a small arrow indicating it can be expanded.

- Chaque composant est défini au moyen d'une balise particulière SELECT, INPUT, TEXTAREA, ...
- A l'intérieur de chaque balise du composant (SELECT par exemple) plusieurs options et notamment une (NAME) qui permet d'identifier le composant : NAME="mon\_composant"
- Les données sont envoyées quand l'utilisateur clique sur un bouton de type SUBMIT

# Servlets et formulaires : du côté HTML

```
<body>
<p>Formulaire de satisfaction du cours Servlet/JSP </p>
<form name="form1" method="get" action="form.html">
<p>
 Nom : <input type="text" name="nom">
 Prénom : <input type="text" name="prenom">
</p>
<p>
 Sexe :
 <input type="radio" name="radio1" value="sexe" checked>mASCULIN
 <input type="radio" name="radio1" value="sexe">féMININ
</p>
<p>Commentaire :

 <textarea name="textarea" cols="50" rows="10"> </textarea>

 <input type="submit" name="Submit" rows="5" value="Soumettre">
</p>
</form>
```

*index.html du projet*

Le formulaire appelle une Servlet avec une requête de type GET

The screenshot shows a Mozilla Firefox window titled "Formulaire de Satisfaction - Mozilla Firefox". The address bar shows the URL <http://localhost:8080/javaee.servlet.UIForm>. The page content is a satisfaction survey form. The "Nom" field contains "BARON", the "Prénom" field contains "Mickaël", and the "Sexe" field has the radio button for "mASCULIN" selected. The "Commentaire" text area contains the text: "Ce cours est magnifiquement réalisé, toutefois les étudiants discutent un peu trop". A "Soumettre" (Submit) button is at the bottom.

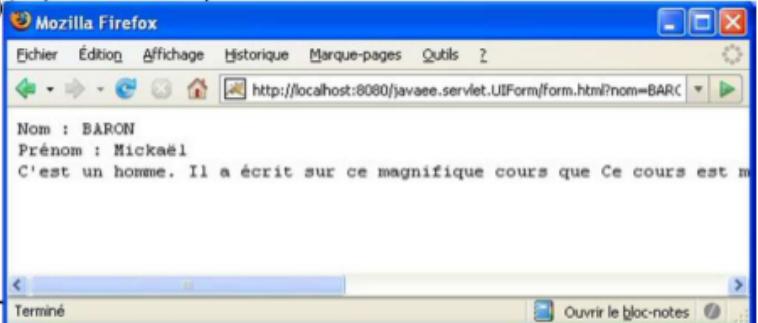
# Servlets et formulaires : du côté Servlet

- Pour lire les données du formulaire : traiter la requête
- Accéder par l'intermédiaire de l'objet HttpServletRequest aux paramètres
  - `String getParameter(String p)` : retourne la valeur du paramètre p
  - `String[] getParamterValues(String p)` : retourne les valeurs du paramètre p

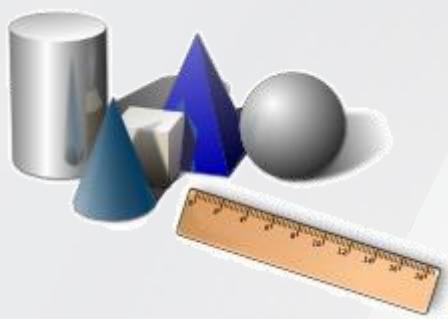
```
public class UIFormServlet extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/plain");
 PrintWriter out = res.getWriter();
 out.println("Nom : " + req.getParameter("nom"));
 out.println("Prénom : " + req.getParameter("prenom"));

 if (req.getParameterValues("radio1") [0].equals("mas"))
 out.print("C'est un homme. Il");
 } else {
 out.print("C'est une femme. Elle");
 }

 out.print(" a écrit sur ce magnifique cours que ");
 out.println(req.getParameter("textarea"));
 }
}
```

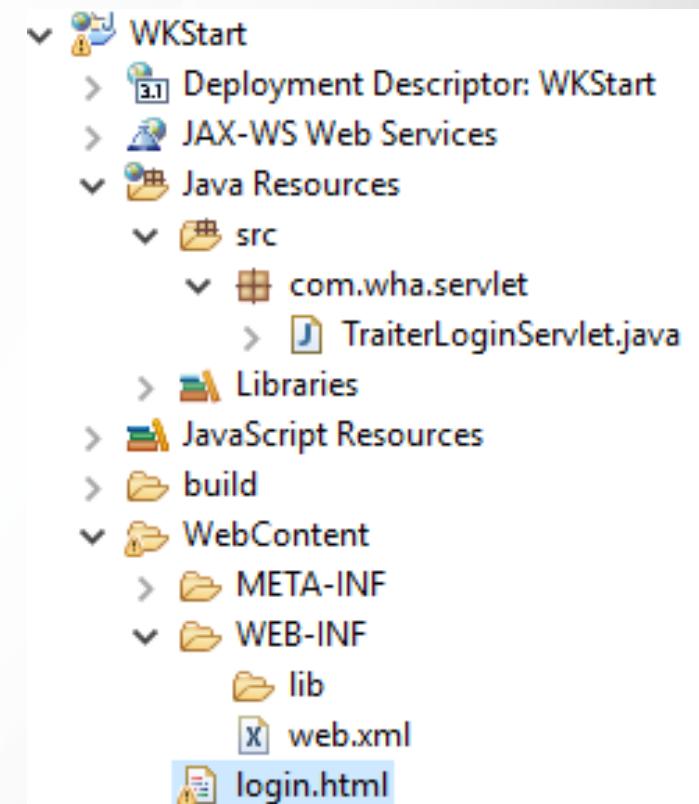


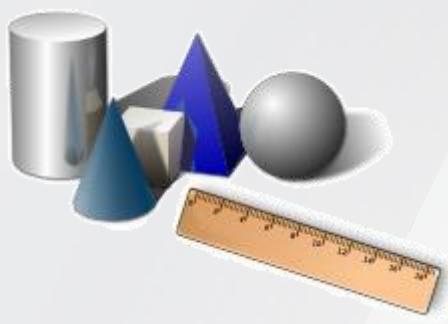
The screenshot shows a Mozilla Firefox window with the URL `http://localhost:8080/javaee.servlet.UIForm/form.html?nom=BARON&prenom=Micka l`. The page content is:  
Nom : BARON  
Prénom : Micka l  
C'est un homme. Il a  crit sur ce magnifique cours que Ce cours est magnifique.  
A 'Termin ' button is visible at the bottom.



# WorkShop 3 : Page d'authentification GestiBank

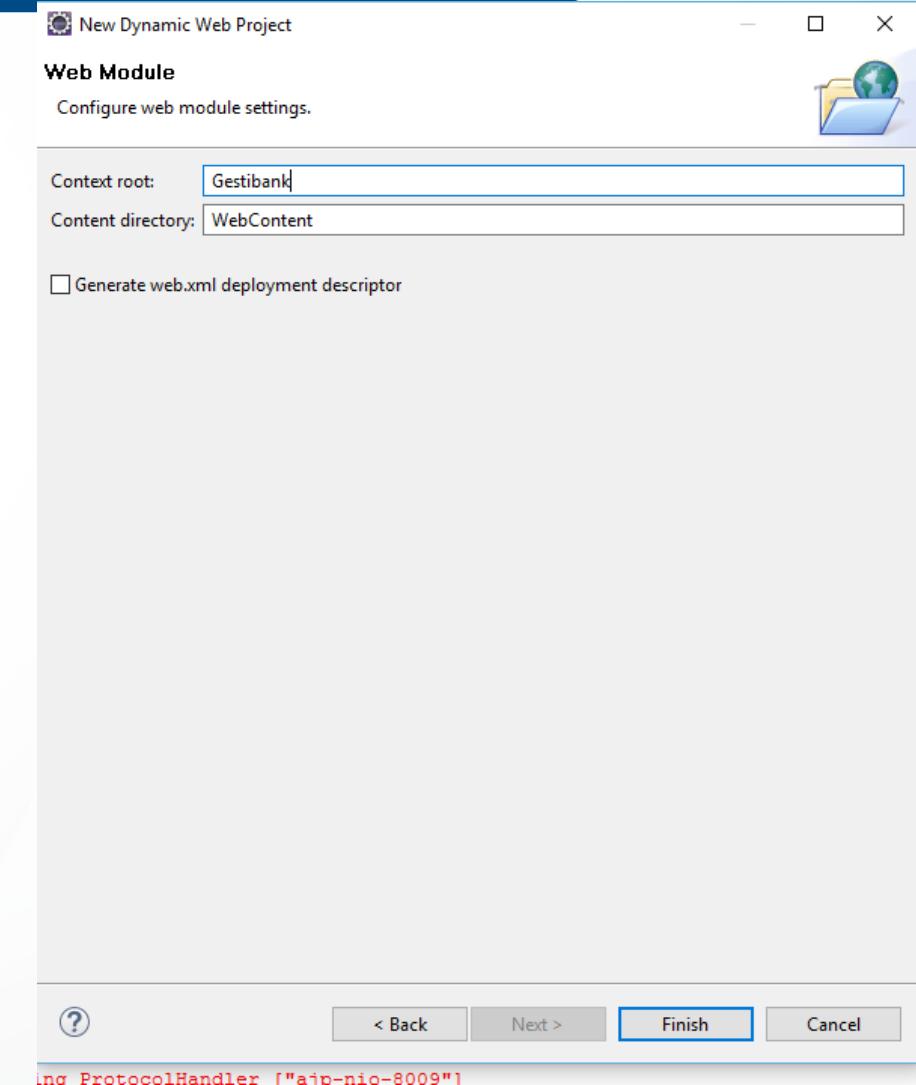
- Le Projet GestBank est un mini projet de gestion bancaire qui sera enrichie au fur et à mesure qu'on avance dans ce cours.
- On va créer dans un premier temps la page de login et la servlet de vérification.

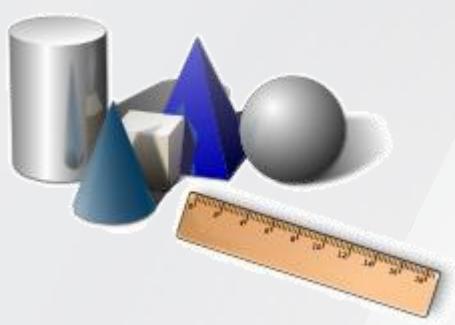




# WorkShop 3 : Page d'authentification GestiBank

- Sous Eclipse on va créer un nouveau projet web dynamique .
  - Nom du projet : WkStart
  - Attention : dans Context root tapez : Gestibank



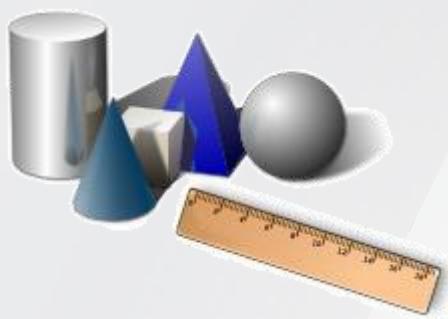


# WorkShop 3 : Page d'authentification Gestibank

## ➤ Creation de la page de login : login.html

```
<html>
 <head>
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
 <title></title>
 </head>
 <body>
 <h1>Bienvenue sur Gestibank</h1>

 <form action="http://localhost:8080/Gestibank/traiterLogin" method="POST">
 <table border="0">
 <tr>
 <td align="center" width="200">Identifiant :</td>
 <td><input type="text" name="login" maxlength="20" size="20" value=""></td>
 </tr>
 <tr>
 <td align="center">Mot de Passe :</td>
 <td><input type="password" name="motDePasse" maxlength="20" size="20" value=""></td>
 </tr>
 <tr>
 <td align="right"><input type="reset" value="Reinitialiser"></td>
 <td align="right"><input type="submit" value="Se connecter"></td>
 </tr>
 </table>
 </form>
 </body>
</html>
```



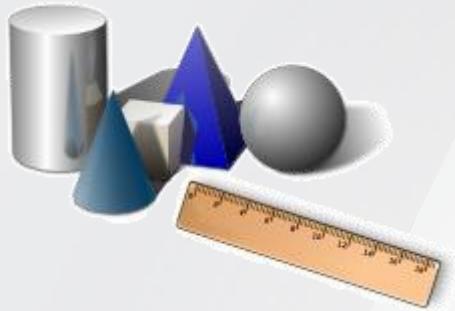
# WorkShop 3 : Page d'authentification GestiBank

## ➤ Creation de la servlet :

- Package : com.wha.servlet
- Nom : TraiterLoginServlet
- Url mapping : **/traiterLogin**
- Generation de web.xml cochée

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
 String identifiant = (String) request.getParameter("login");
 String motDePasse = (String) request.getParameter("motDePasse");

 if (identifiant.equals("user") && motDePasse.equals("pwd")) {
 System.out.println("Connexion réussie , Bonjour :" + identifiant);
 } else {
 System.out.println("connection impossible");
 }
}
```



# WorkShop 3 : Page d'authentification GestiBank

- Une fois terminer
- Déployer et tester l'application en exécutant
- Run as → Run on Server sur le fichier login.html

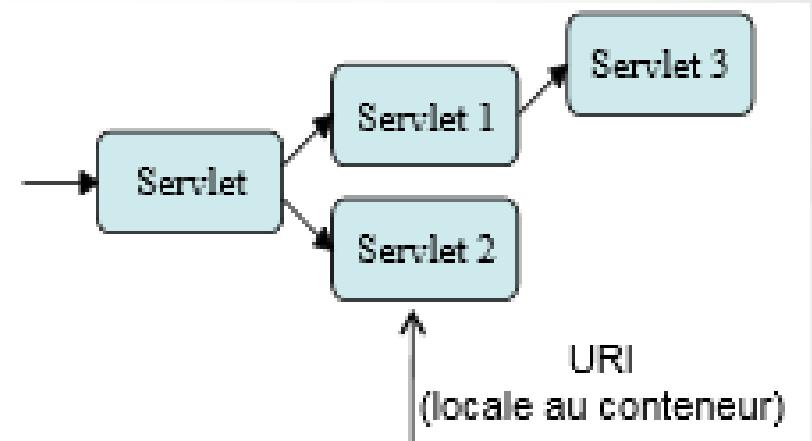
The screenshot shows a web browser window with the following details:

- Address Bar:** http://localhost:8080/Gestibank/login.html
- Title:** Bienvenue sur GestiBank
- Form Fields:**
  - Identifiant:
  - Mot de Passe:
  - Buttons: Reinitialiser, Se connecter
- Console Output (red text):**

```
INFO: Server startup in 716 ms
connection impossible
Connexion réussie , Bonjour :user
```

# Collaboration de Servlets

- Les Servlets s'exécutant dans le même serveur peuvent dialoguer entre elles
- Ceci pour avoir :
  - Meilleure modularité
  - Meilleure réutilisation
- Deux principaux styles de collaboration
  - **Partage d'information** : un état ou une ressource.
    - Exemple : un magasin en ligne pourrait partager les informations sur le stock des produits ou une connexion à une base de données
  - **Partage du contrôle** : une requête.
    - Exemple : Réception d'une requête par une Servlet et laisser l'autre Servlet une partie ou toute la responsabilité du traitement



# Collaboration de Servlets : partage d'information

- La collaboration est obtenue par l'interface `ServletContext`
- L'utilisation de `ServletContext` permet aux applications web de disposer de son propre conteneur d'informations unique
- Une **Servlet** retrouve le `ServletContext` de son application web par un appel à `getServletContext()`
- Exemples de méthodes
  - `void setAttribute(String name, Object o)` : lie un objet sous le nom indiqué
  - `Object getAttribute(String name)` : retrouve l'objet sous le nom indiqué
  - `Enumeration getAttributeNames()` : retourne l'ensemble des noms de tous les attributs liés
  - `void removeAttribute(String name)` : supprime l'objet lié sous le nom indiqué

# Partage d'information

- Exemple : Servlets qui vendent des pizzas et partagent une spécialité du jour

```
public class PizzasAdmin extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/plain");
 PrintWriter out = res.getWriter();
 ServletContext context = this.getServletContext();
 context.setAttribute("Specialite", "Jambon Fromage");
 context.setAttribute("Date", new Date());
 out.println("La pizza du jour a été définie.");
 }
}
```

*PizzasAdmin.java du projet*  
**ServletContext**

**Création de deux attributs**

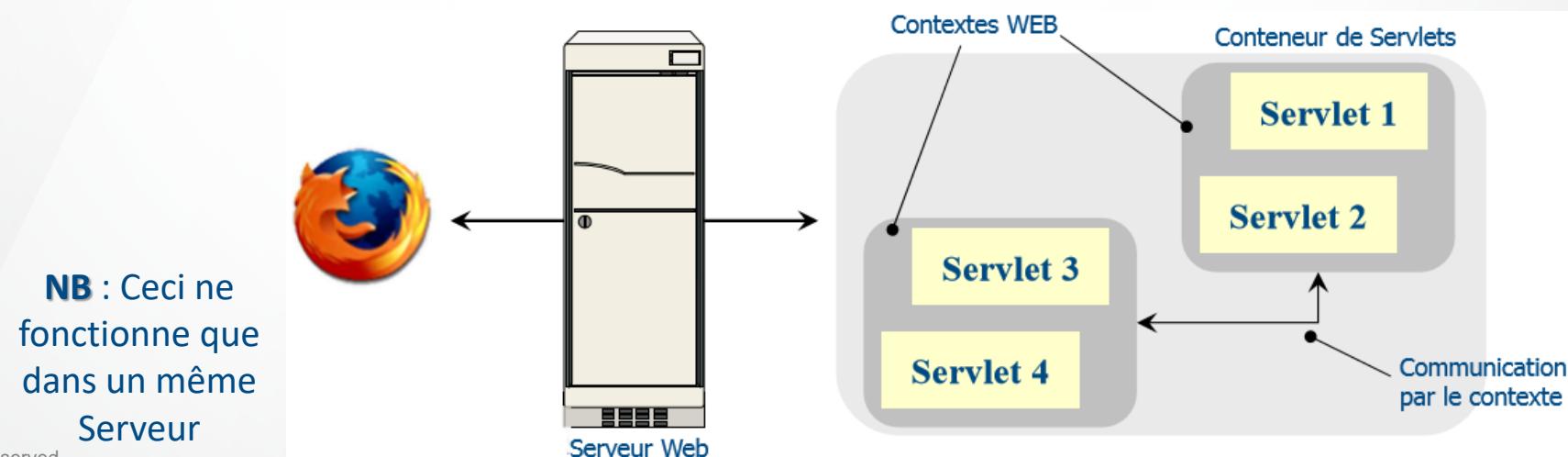
```
public class PizzasClient extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 ...
 ServletContext context = this.getServletContext();
 String pizza_spec = (String)context.getAttribute("Specialite");
 Date day = (Date)context.getAttribute("Date");
 DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM);
 String today = df.format(day);
 out.println("Aujourd'hui (" + today + "), notre specialite est : " + pizza_spec);
 }
}
```

*PizzasClient.java du projet*  
**ServletContext**

**Lecture des attributs**

# Partage d'information

- Possibilité de partager des informations entre contextes web
- Première solution : utilisation d'un conteneur d'informations externes (une base de données par exemple ou un annuaire)
- Seconde solution : la Servlet recherche un autre contexte à partir de son propre contexte
  - `ServletContext getContext (String uripath)` : obtient le contexte à partir d'un chemin URI (`uripath = chemin absolu`)



# Partage d'information

- Exemple : permet d'afficher la spécialité du jour de l'application web précédente

```
public class ReadSharePizzas extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/html");
 PrintWriter out = res.getWriter();

 ServletContext my_context = this.getServletContext();
 ServletContext pizzas_context = my_context.getContext("/ServletContext");
 String pizza_spec = (String)pizzas_context.getAttribute("Specialite");
 Date day = (Date)pizzas_context.getAttribute("Date");

 DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM);
 String today = df.format(day);

 out.println("Aujourd'hui (" + today + "), notre specialite est : " + pizza_spec);
 }
}
```

Contextes de l'application  
web précédente

*ReadSharePizzas.java* du projet  
**CrossServletContext**

# Collaboration de Servlets : partage du contrôle

- Les Servlets peuvent partager ou distribuer le contrôle de la requête
- Deux types de distribution
  - Distribuer un renvoi : une Servlet peut renvoyer une requête entière
  - Distribuer une inclusion : une Servlet peut inclure du contenu généré
- Les avantages sont
  - La délégation de compétences
  - Une meilleure abstraction et une plus grande souplesse
  - Architecture logicielle MVC (Servlet = contrôle et JSP = présentation)

# Collaboration de Servlets : partage du contrôle

- Le support de la délégation de requête est obtenu par l'interface RequestDispatcher
- Une Servlet obtient une instance sur la **requête**
  - RequestDispatcher getRequestDispatcher(String path) : retourne une instance de type RequestDispatcher par rapport à un composant
  - Un composant peut-être de tout type : Servlet, JSP, fichier statique, ...
  - path est un chemin relatif ou absolu ne pouvant pas sortir du contexte
- Pour distribuer en dehors du contexte courant il faut :
  - Identifier le contexte extérieur (utilisation de getContext() )
  - Utiliser la méthode getRequestDispatcher(String path)
  - Le chemin est uniquement en absolu

# Partage du contrôle : distribuer un renvoi

- La méthode `forward(...)` de l'interface `RequestDispatcher` renvoie une requête d'une Servlet à une autre ressource sur le serveur
  - `void forward(ServletRequest req, ServletResponse res)` : redirection de requête

```
RequestDispatcher disp =
 req.getRequestDispatcher("/index.html");
disp.forward(req, res);
```

- Possibilité de transmettre des informations lors du renvoi
  - en attachant une chaîne d'interrogation (au travers de l'URL)
  - en utilisant les attributs de requête via la méthode `setAttribute(...)`
- Les choses à ne pas faire ...
  - ne pas effectuer de modification sur la réponse avant un renvoi
  - ne rien faire sur la requête et la réponse après une distribution d'un renvoi

# Partage du contrôle : distribuer un renvoi

## ➤ Exemple : distribuer un renvoi de Emetteur à Recepteur

```
public class SenderServlet extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 req.setAttribute("seconde", "au revoir"); • Transmission d'informations par attributs

 RequestDispatcher dispat = req.getRequestDispatcher("/recepteur.html?mot=bonjour"); • Le chemin est absolu par rapport au contexte de l'application web
 dispat.forward(req,res); Transmission d'informations par chaîne d'interrogation
 // Ne rien faire sur req et res
 }
}
```

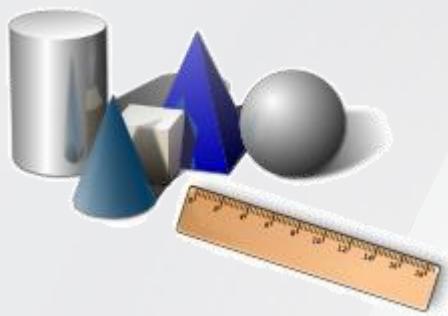
*SenderServlet.java* du projet  
**ForwardInclude**

```
public class ReceiverServlet extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/plain");
 PrintWriter out = res.getWriter();
 out.println(req.getParameter("mot")); • Affichage des informations stockées dans la requête
 out.println(req.getAttribute("seconde"));
 }
}
```

*ReceiverServlet.java* du projet  
**ForwardInclude**

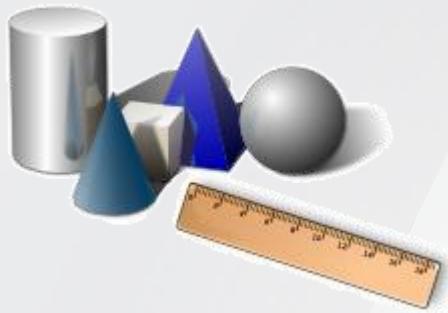
# Partage du contrôle : distribuer un renvoi

- Nous avons vu au début de cette partie qu'il existait une méthode de redirection
  - `sendRedirect(...)` est une redirection effectuée par le client
  - `forward(...)` est une redirection effectuée par le serveur
- Est-il préférable d'utiliser `forward(...)` ou `sendRedirect(...)` ???
  - `forward(...)` est à utiliser pour la partage de résultat avec un autre composant sur le même serveur
  - `sendRedirect(...)` est à utiliser pour des redirections externes car aucune recherche `getContext(...)` n'est nécessaire



## WorkShop 4 : Ajout des redirections

- Reprenez le projet WKStart
- On va mettre en place les redirections après succès ou échec d'authentification
- On va créer 3 pages html
  - erreur.html : page d'erreur en cas d'erreur d'authentification
  - pagePrincipale.html : page privée une fois la connexion est réussie
  - fin.html : page de déconnection



# WorkShop 4 : Ajout des redirections

pagePrincipale.html

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Erreur de connection</title>
</head>
<body>
 <h1>Erreur de connection</h1>
 Se reconnecter
</body>
</html>
```

erreur.html

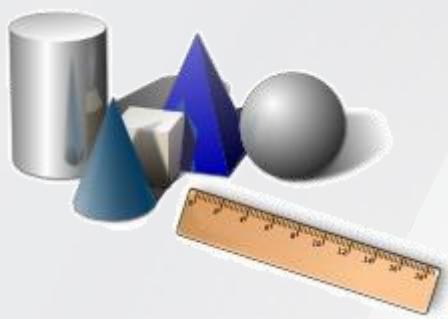
```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
 <title>Menu BANKONET</title>
</head>
<body>
 <h2> Bonjour </h2>
 <p> Opérations disponibles </p>

 Compte Courant

 Déconnection
</body>
</html>
```

fin.html

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Fin</title>
</head>
<body>
 <h1>Vous êtes maintenant déconnecté</h1>
 Retour à la page d'accueil
</body>
</html>
```



# WorkShop 4 : Ajout des redirections

- Il ne reste qu'a modifier le servlet

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
String identifiant = (String) request.getParameter("login");
String motDePasse = (String) request.getParameter("motDePasse");

if (identifiant.equals("user") && motDePasse.equals("pwd")) {
 response.sendRedirect(request.getContextPath() + "/pagePrincipale.html");
 System.out.println("Connexion réussie , Bonjour :" + identifiant);
} else {
 response.sendRedirect(request.getContextPath() + "/erreur.html");
 System.out.println("Connexion impossible");
}

}
```

# Suivi de session

- Le protocole HTTP est un protocole sans état
- Impossibilité alors de garder des informations d'une requête à l'autre (identifier un client d'un autre)
- Obligation d'utiliser différentes solutions pour remédier au problème d'état
  - Authentifier l'utilisateur
  - Champs de formulaire cachés
  - Réécriture d'URL
  - Cookies persistants
  - Suivi de session

# Cookies persistants : Cookie

- Un cookie est une information envoyée au navigateur (client) par un serveur WEB qui peut ensuite être relue par le client
- Lorsqu'un client reçoit un cookie, il le sauve et le renvoie ensuite au serveur chaque fois qu'il accède à une page sur ce serveur
- La valeur d'un cookie pouvant identifier de façon unique un client, ils sont souvent utilisés pour le suivi de session
- Les cookies ont été introduits par la première fois dans Netscape Navigator
  - [home.netscape.com/newsref/std/cookie\\_spec.html](http://home.netscape.com/newsref/std/cookie_spec.html)
  - [www.cookie-central.com](http://www.cookie-central.com)

# Cookies persistants : Cookie

- L'API Servlet fournit la classe javax.servlet.http.Cookie pour travailler avec les Cookies
  - `Cookie(String name, String value)` : construit un cookie
  - `String getName()` : retourne le nom du cookie
  - `String getValue()` : retourne la valeur du cookie
  - `setValue(String new_value)` : donne une nouvelle valeur au cookie
  - `setMaxAge(int expiry)` : spécifie l'âge maximum du cookie
- Pour la création d'un nouveau cookie, il faut l'ajouter à la réponse ( `HttpServletResponse` )
  - `addCookie(Cookie mon_cook)` : ajoute à la réponse un cookie
- La Servlet récupère les cookies du client en exploitant la réponse ( `HttpServletRequest` )
  - `Cookie[] getCookies()` : récupère l'ensemble des cookies du site

# Cookies persistants : Cookie

- Code pour créer un cookie et l'ajouter au client

```
Cookie cookie = new Cookie("Id", "123");
res.addCookie(cookie);
```

- Code pour récupérer les cookies

```
Cookie[] cookies = req.getCookies();
if (cookies != null) {
 for (int i = 0; i < cookies.length; i++) {
 String name = cookies[i].getName();
 String value = cookies[i].getValue();
 }
}
```

# Cookies persistants : Cookie

- Exemple : gestion de session (identifier un client d'un autre) par l'intermédiaire des cookies persistants

```
public class CookiesServlet extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 ...
 String sessionId = null;
 Cookie[] cookies = req.getCookies();
 if (cookies != null) {
 for (int i = 0; i < cookies.length; i++) {
 if (cookies[i].getName().equals("sessionid")) {
 sessionId = cookies[i].getValue();
 } } }
 if (sessionId == null) {
 sessionId = new java.rmi.server.UID().toString();
 Cookie c = new Cookie("sessionid", sessionId);
 res.addCookie(c);
 out.println("Bonjour le nouveau");
 } else {
 out.println("Encore vous"); ... }
 }
}
```

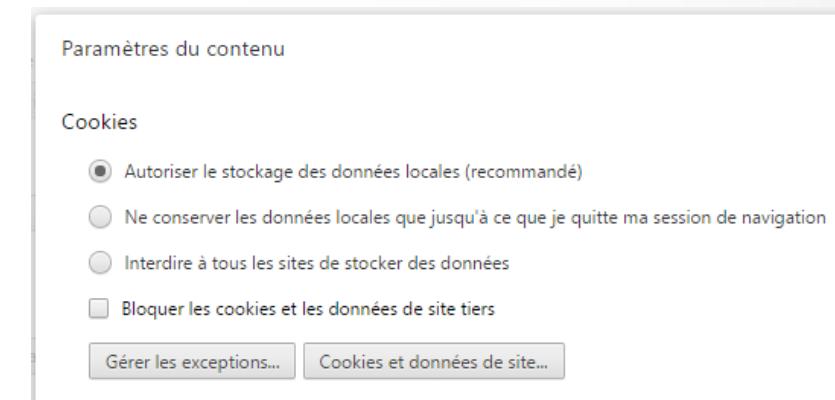
Génère un identifiant unique pour chaque client

*CookiesServlet.java du projet  
Cookies*

# Suivi de session : HttpSession

- Le plus gros problème des cookies est que les navigateurs ne les acceptent pas toujours

➤ L'utilisateur peut configurer son navigateur pour qu'il refuse ou pas les cookies



- Les navigateurs n'acceptent que 20 cookies par site, 300 par utilisateur et la taille d'un cookie peut être limitée à 4096 octets

# Suivi de session : HttpSession

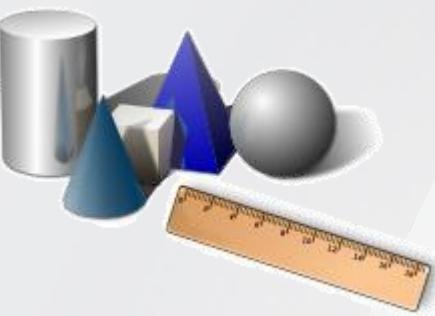
- Solutions : utilisation de l'API de suivi de session HttpSession
- Méthodes de création liées à la requête ( HttpServletRequest )
  - HttpSession getSession() : retourne la session associée à l'utilisateur
  - HttpSession getSession(boolean p) : création selon la valeur de p
- Gestion d'association( HttpSession )
  - Enumeration getAttributNames() : retourne les noms de tous les attributs
  - Object getAttribut(String name) : retourne l'objet associé au nom
  - setAttribut(String na, Object va) : modifie na par la valeur va
  - removeAttribut(String na) : supprime l'attribut associé à na
- Destruction ( HttpSession )
  - invalidate() : expire la session
  - logout() : termine la session

# Suivi de session : HttpSession

## ➤ Exemple : suivi de session pour un compteur

```
public class HttpSessionServlet extends HttpServlet {
 protected void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException {
 res.setContentType("text/plain"); PrintWriter out = res.getWriter();
 HttpSession session = req.getSession();
 Integer count = (Integer)session.getAttribute("count");
 if (count == null)
 count = new Integer(1);
 else
 count = new Integer(count.intValue() + 1);
 session.setAttribute("count", count);
 out.println("Vous avez visité cette page " + count + " fois.");
 }
}
```





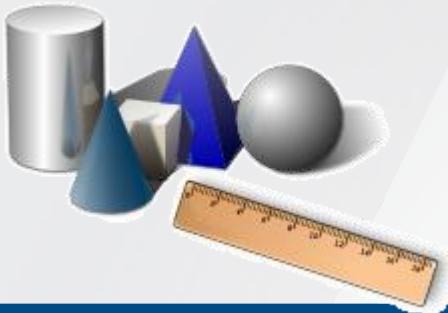
# WorkShop 5 : Gestion des sessions

## ➤ Reprenons notre projet :

- On va ajouter la création de session dans la servlet : TraiterLoginServlet.jsp

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 String identifiant = (String) request.getParameter("login");
 String motDePasse = (String) request.getParameter("motDePasse");

 if (identifiant.equals("user") && motDePasse.equals("pwd")) {
 HttpSession session = request.getSession(true);
 session.setAttribute("utilisateur", identifiant);
 response.sendRedirect(request.getContextPath() + "/pagePrincipale.jsp");
 System.out.println("Connexion réussie , Bonjour :" + identifiant);
 } else {
 response.sendRedirect(request.getContextPath() + "/erreur.html");
 System.out.println("Connexion impossible");
 }
}
```



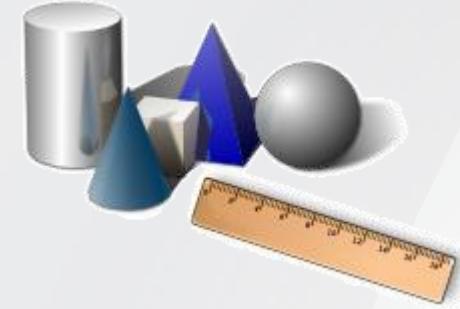
# WorkShop 5 : Gestion des sessions

- On va modifier un peut notre page principale :
  - On renomme : pagePrincipale.html en pagePrincipale.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" %>
<!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>Menu GESTIBANK</title>
 </head>
 <body>
 <% String pseudo = (String) session.getAttribute("utilisateur"); %>
 <h2> Bonjour : <%= pseudo %></h2>
 <p> Opérations disponibles </p>

 Compte Courant

 <a href="<%="request.getContextPath() + "/fin"%>">Déconnection
 </body>
</html>
```



# WorkShop 5 : Gestion des sessions

- Testez l'application
- Il ne reste que la mise en place de la déconnection
  - Ajouter un nouveau servlet : DeconnectionServlet.jsp

```
@WebServlet("/fin")
public class DeconnectionServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;

 /**
 * @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)
 */
 protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

 HttpSession session = request.getSession(false);
 session.invalidate();
 System.out.println("session invalidée");
 response.sendRedirect(request.getContextPath() + "/fin.html");
 System.out.println("redirigé vers fin.html");
 }
}
```

# Problèmes avec le cache du navigateur

- La plupart des navigateurs utilisent un cache pour les pages et les images
  - L'utilisateur peut voir "l'ancien état d'une page"
    - Peut paraître pour un bug, surtout dans le cas d'une page de login
- Pour éviter cela, il faut désactiver le cache dans la réponse HTTP :

```
response.setHeader("Pragma", "No-cache");
response.setDateHeader("Expires", 0);
response.setHeader("Cache-Control", "no-cache");
```

# LES CONTEXTES

- Il existe 3 contextes dans lesquels nous pouvons temporairement stocker de l'information lorsque cela est requis :
  - Le contexte commun à toutes les servlets d'une application web (objet ServletContext)
  - Le contexte de session (objet HttpSession)
  - Le contexte de la requête (objet HttpServletRequest)
- Ces 3 contextes proposent les mêmes méthodes setAttribute et getAttribute.
- Choisir le plus adéquat Pour ne pas conserver l'information plus que nécessaire Pour libérer les ressources (mémoire) dès que possible

# LES CONTEXTES : CYCLE DE VIE DES ATTRIBUTS

- Un attribut est un objet
  - Classiquement utilisé pour le transfert d'informations du contrôleur (servlet) vers la vue (page JSP).
- Enregistrer et récupérer des informations
  - `void setAttribute(String, Object)`
  - `Object getAttribute(String)`
- Supprimer une entrée dans un contexte
  - `void removeAttribute(String)`

# LES CONTEXTES : CONTEXTE DE REQUÊTE

- Les informations présentes dans le contexte de requête sont utilisables pendant un aller-retour client serveur.
  - Après, la requête est automatiquement détruite par le serveur d'applications.
  - Très utilisé pour communiquer de la servlet vers la page jsp.
  - Chaque client a son propre contexte de requête.

```
public void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
 ...
 req.setAttribute("erreurMessage", "Un message d'erreur");
}
```

# LES CONTEXTES : CONTEXTE DE SESSION

- Les informations présentes dans le contexte de session sont utilisables jusqu'à la suppression de la session utilisateur.
  - Par déconnexion ou par timeout.
  - Pour garder de bonnes performances, une bonne pratique est de stocker un minimum de choses dans la session.
  - Chaque client a son propre contexte de session.

```
public void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
 ...
 HttpSession session = req.getSession(true);
 session.setAttribute("livre", monLivre);
}
```

# LES CONTEXTES : CONTEXTE DE SERVLET

- Les informations présentes dans le contexte de servlet sont utilisables tant que l'application est lancée.
  - Permet de stocker globalement des informations.
  - Durée de vie
    - jusqu'à l'arrêt du serveur
    - jusqu'au rechargement du contexte (en cas de modification)
  - Le contexte de servlet est partagé par tous les utilisateurs

```
public void service(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
 //...
 req.getServletContext().setAttribute("now", new Date());
}
```

# LES CONTEXTES : RÉCAPITULATIF

Où a-t-on besoin de l'information ?

	Plusieurs pages	Les pages liées par le traitement d'une même requête
Tous les utilisateurs	ServletContext	ServletContext
L'utilisateur courant	HttpSession	HttpServletRequest
Qui utilise l'information ?		

# Exemple de déclaration de Servlet

```
<servlet>
 <servlet-name>Faces Servlet</servlet-name>

 <servlet-class>
 javax.faces.webapp.FacesServlet
 </servlet-class>

 <load-on-startup>1</load-on-startup>

</servlet>
<servlet-mapping>
 <servlet-name>Faces Servlet</servlet-name>
 <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

# Pages de démarrage

```
<welcome-file-list>
 <welcome-file>
 faces/MenuComptesBancaires.xhtml
 </welcome-file>
 <welcome-file>
 faces/index.xhtml
 </welcome-file>
</welcome-file-list>
```

# A quoi servent les écouteurs ?

- Par exemple on peut déclarer une classe qui implémente ServletContextListener et dont la méthode contextInitialized sera appelée au déploiement de l'application
  - Ex : remplir la base de données avec des données de test, créer des comptes pour l'admin, etc
- Ou encore compter le nombre de session ouvertes pour afficher combien de personnes sont « online » sur le site
  - On va utiliser un SessionListener

# Déclaration dans le web.xml

```
<listener>
 <description>
 Classe pour remplir la base au déploiement
 </description>
 <listener-class>
 tools.InitApplication
 </listener-class>
</listener>
```

# Classe de remplissage de la BD au déploiement

```
public class InitApplication implements ServletContextListener {

 @Override
 public void contextInitialized(ServletContextEvent sce) {

 System.out.println("##### BASE CREE #####");
 compteBancaireFacade.creerComptesDeTest();
 }

 @Override
 public void contextDestroyed(ServletContextEvent sce) { ... }
}
```

# SessionListener : compter les personnes online. Dans le web.xml

```
<listener>
 <listener-class>
 org.kodejava.servlet.examples.SessionCounter
 </listener-class>
</listener>
```

# Classe écouteur

```
public class SessionCounter implements HttpSessionListener {
 private List sessions = new ArrayList();

 public SessionCounter() { }

 public void sessionCreated(HttpSessionEvent event) {
 HttpSession session = event.getSession();
 sessions.add(session.getId());
 session.setAttribute("counter", this);
 }
 public void sessionDestroyed(HttpSessionEvent event) {
 HttpSession session = event.getSession();
 sessions.remove(session.getId());
 session.setAttribute("counter", this);
 }
 public int getActiveSessionNumber() { return sessions.size(); }
}
```

# Affichage dans une page JSP

```
<html>
<head>
<title>Session Counter</title>
</head>
<body>
 Nombre de personnes online sur le site :
 ${counter.activeSessionNumber}
</body>
</html>
```

## REMARQUES :

- \${counter.activeSessionNumber} va chercher une propriété activeSessionNumber dans la page, la requête, la session, le contexte, jusqu'à la trouver.
- Appelle getActiveSessionNumber()

# Exemple de paramètres d'initialisation

```
<init-param>
 <param-name>compressionThreshold</param-name>
 <param-value>10</param-value>
</init-param>

<init-param>
 <param-name>debug</param-name>
 <param-value>0</param-value>
</init-param>
```

# Récupération des initParams, ici dans un écouteur, au déploiement

```
public void contextInitialized(ServletContextEvent event) {
 this.context = event.getServletContext();
 String comp =
 context.getInitParameter(" compressionThreshold ");
 log("contextInitialized()");
 log(<< compression= " + comp);
}
```

# Ici dans une Servlet

```
public void init() {
 response.setContentType("text/html");
 PrintWriter writer = response.getWriter();
 ServletContext context = getServletContext();

 String logPath = context.getInitParameter("LOG.PATH");
 writer.println("Log Path: " + logPath + "
");

 Enumeration enumeration = context.getInitParameterNames();
 while (enumeration.hasMoreElements()) {
 String paramName = (String) enumeration.nextElement();
 String paramValue = context.getInitParameter(paramName);
 writer.println("Context Init Param: [" + paramName + " = " + paramValue +
"
"]
");
 }
}
```

# Servlets 3.0

- Comme on est pas là pour faire du « XML Sitting », le modèle 3.0 a apporté de nombreuses annotations qui évitent de remplir le fichier web.xml.
- Pour des applications simples il est facultatif
- Nécessaire encore pour certaines tâches

# Annotations Servlets 3.0

```
@WebServlet(asyncSupported = false,
 name = "HelloAnnotationServlet",
 urlPatterns = {"/helloanno"},
 initParams = {@WebInitParam(name="param1",
 value="value1"),
 @WebInitParam(name="param2",
 value="value2") })
public class HelloAnnotationServlet extends HttpServlet {...}
```

Récupération des init params dans doGet() ou doXXX() par :  
getInitParameter("param1");

# Annotations pour file upload

```
@WebServlet("/upload.html")
@MultipartConfig(location="c:\\tmp", fileSizeThreshold=1024*1024,
maxFileSize=1024*1024*5, maxRequestSize=1024*1024*5*5)
public class FileUploadServlet extends HttpServlet {
 @Override
 protected void doPost(HttpServletRequest req,
 HttpServletResponse resp) throws ServletException, IOException {
 resp.setContentType("text/html");
 PrintWriter out = resp.getWriter();
 Collection<Part> parts = req.getParts();
 out.write("<h2> Total parts : "+parts.size()+"</h2>");

 for(Part part : parts) {
 printPart(part, out);
 part.write("samplefile");
 }
 }
}
```

# Code qui créer le fichier

```
private void printPart(Part part, PrintWriter pw) {
 StringBuffer sb = new StringBuffer();
 sb.append("<p>");
 sb.append("Name : "+part.getName());
 sb.append("
");
 sb.append("Content Type : "+part.getContentType());
 sb.append("
");
 sb.append("Size : "+part.getSize());
 sb.append("
");

 for(String header : part.getHeaderNames()) {
 sb.append(header + " : "+part.getHeader(header));
 sb.append("
");
 }
 sb.append("</p>");
 pw.write(sb.toString()); }
}
```

# Code du formulaire d'envoi

```
<html>
<body>
 <p>Commons File Upload Example</p>
 <form action="uploadServlet1"
 enctype="multipart/form-data" method="POST">
 <input type="file" name="file1">

 <input type="Submit" value="Upload File">

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

# Compléments sur l'envoi de fichiers en multipart (1)

- Attention, si le formulaire d'envoi possède des champs classiques ou « hidden » en plus du ou des fichiers, leurs valeurs ne seront pas accessible dans la Servlet par `request.getParameter(nomDuParam)`
  - Valeur null renvoyée, faire `getPart()` à la place !
- A la place :

```
private String getParamFromMultipartRequest(HttpServletRequest request, String paramName) throws IOException, ServletException {
 Part part= request.getPart(paramName);
 Scanner scanner = new Scanner(part.getInputStream());
 String myString = scanner.nextLine();
 return myString;
}
```

# Compléments sur l'envoi de fichiers en multipart (2)

- Si on ne précise pas le paramètre location dans l'annotation :  
`@MultipartConfig(location="...")`  
Les fichiers écrits par part.write() iront dans le chemin où est exécuté le projet
- Si on veut que les fichiers aient un URL valide (soit affichables dans une page)
  - Location = un espace visible par le serveur HTTP, soit le docroot, soit le répertoire web (ou un sous-répertoire) de votre projet

# Annotation ServletFilter

```
@WebFilter(urlPatterns={"/"}, description="Request timer filter")
public class TimerFilter implements Filter {
 private FilterConfig config = null;
 @Override
 public void init(FilterConfig config) throws ServletException {
 this.config = config;
 config.getServletContext().log("TimerFilter initialized");
 }
 @Override
 public void doFilter(ServletRequest req, ServletResponse resp,
 FilterChain chain) throws IOException, ServletException {
 long before = System.currentTimeMillis();
 chain.doFilter(req, resp);
 long after = System.currentTimeMillis();
 String path = ((HttpServletRequest)req).getRequestURI();
 config.getServletContext().log(path + " : "+(after-before));
 }
}
```

# Servlet Conclusion

- Servlets : étendent le comportement des serveurs Web avec des programme Java
  - Portabilité, facilité d'écriture (Java)
  - Définition du code, du paquetage, du déploiement
  - Persistance des données dans les servlets Servlet chargée et instanciée une seule fois
  - Exécutée en // avec des processus légers (threads)
- Mais :
  - Difficile d'écrire du code HTML dans du code Java Introduction de la technologie Java Server Pages (JSP)
  - Pas de mécanisme intégré de distribution Introduction de la technologie Enterprise Java Beans (EJB)

## ➤ COMMUNICATION ENTRE SERVLETS



# Sommaire

- Chapitre 1 : INTRODUCTION
  - Chapitre 2 : PLATE-FORME JAVA EE
  - **Chapitre 3 : SERVLET / JSP**
  - Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE
  - Chapitre 5 : JSF
  - Chapitre 6 : PRIMEFACES
- 
- 

# Java server page - jsp

- Qu'est-ce que c'est ?
  - langage de script qui combine
    - un langage à balises (html ou xml)
    - des fragments de code java
  - chaque page est compilée en une servlet
- Défini par qui
  - Sun puis Oracle

# Exemple de page jsp

```
<%@ page language="java" %>

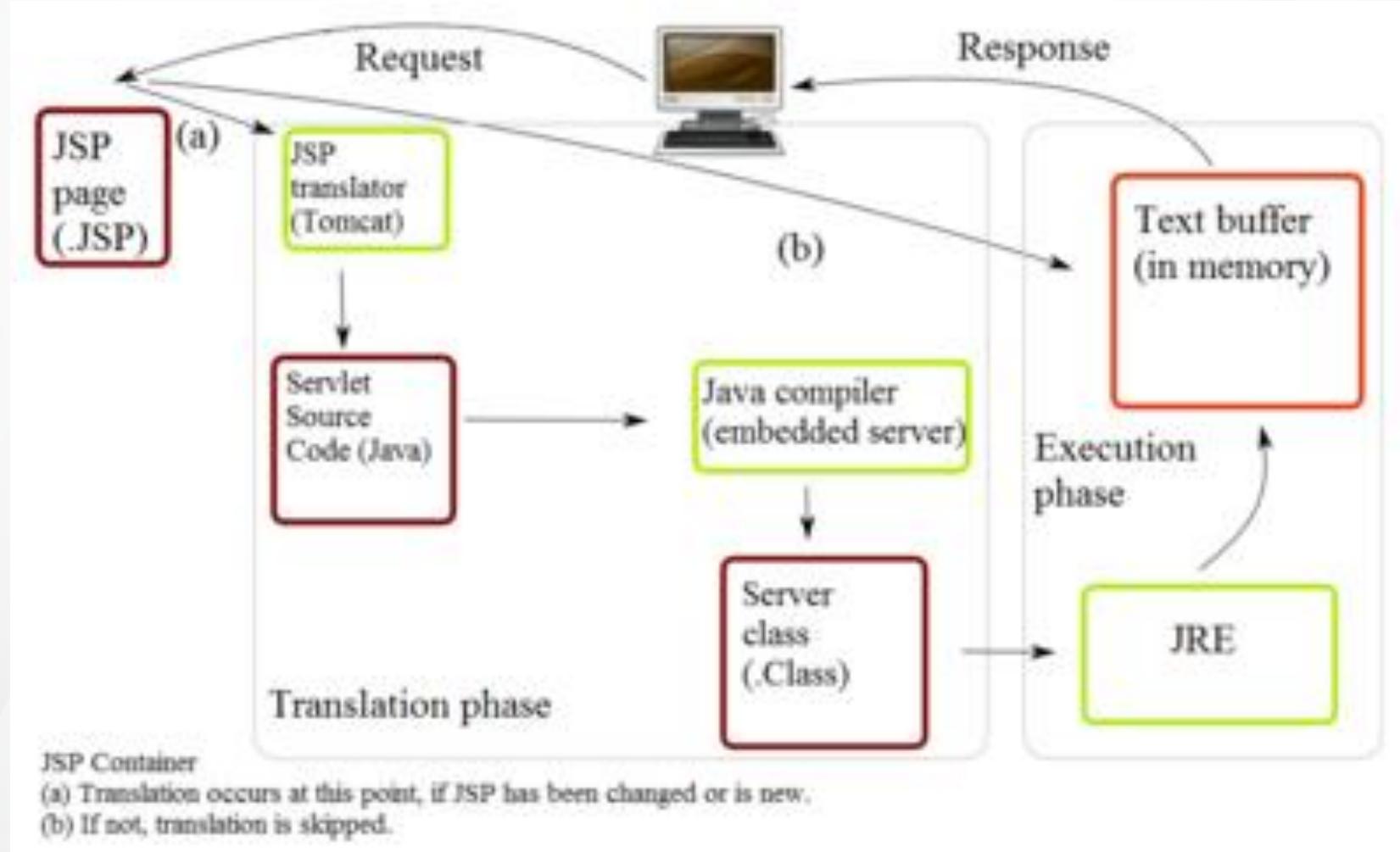
<html>
 <head>
 <title>Untitled</title>
 </head>
 <body>
 <h1>Résultat</h1>
 <% int francs = Integer.parseInt(
 request.getParameter("somme"));%>
 <%= francs + " Frs valent " +
 francs/6.55957 + " Euro"%>
 </body>
</html>
```

# Architecture d'une page JSP

- Une page jsp est un document texte ayant une syntaxe mixte HTML/Java.
  - Code HTML = contenu statique, mise en page
  - Code Java = contenu dynamique
- Une page jsp n'est pas une classe Java
  - Une page jsp est une facilité de développement
  - Le serveur d'application génère une classe java à partir de la page JSP
  - La classe Java générée est compilée, puis exécutée à la demande

# Architecture

## Moteur JSP/Servlet

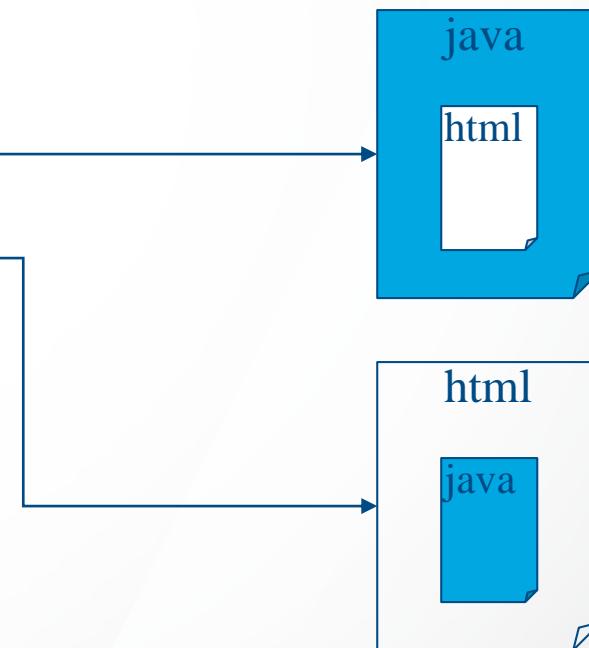


JSP Container

- (a) Translation occurs at this point, if JSP has been changed or is new.
- (b) If not, translation is skipped.

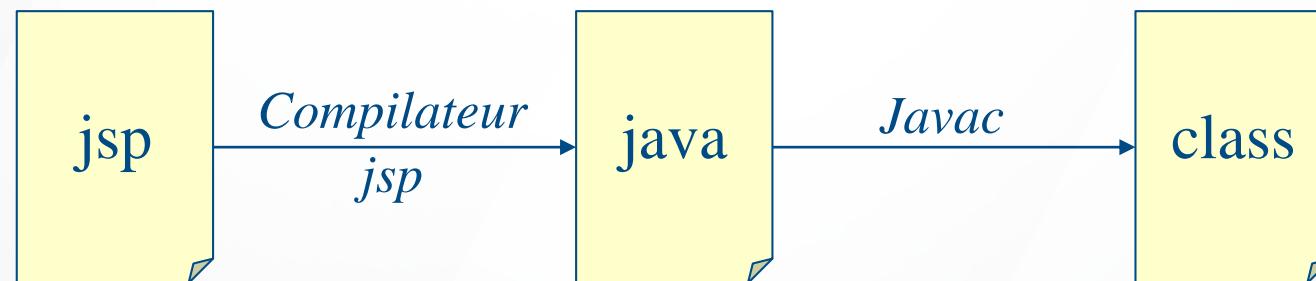
# Servlet versus jsp

- Un même objectif
  - construction côté serveur de pages dynamiques
  - concurrents de php, cgi, asp
- Deux logiques
  - servlet : priorité à java
  - jsp : priorité à html (ou xml)
- Un même avantage
  - ouverture sur l'univers Java



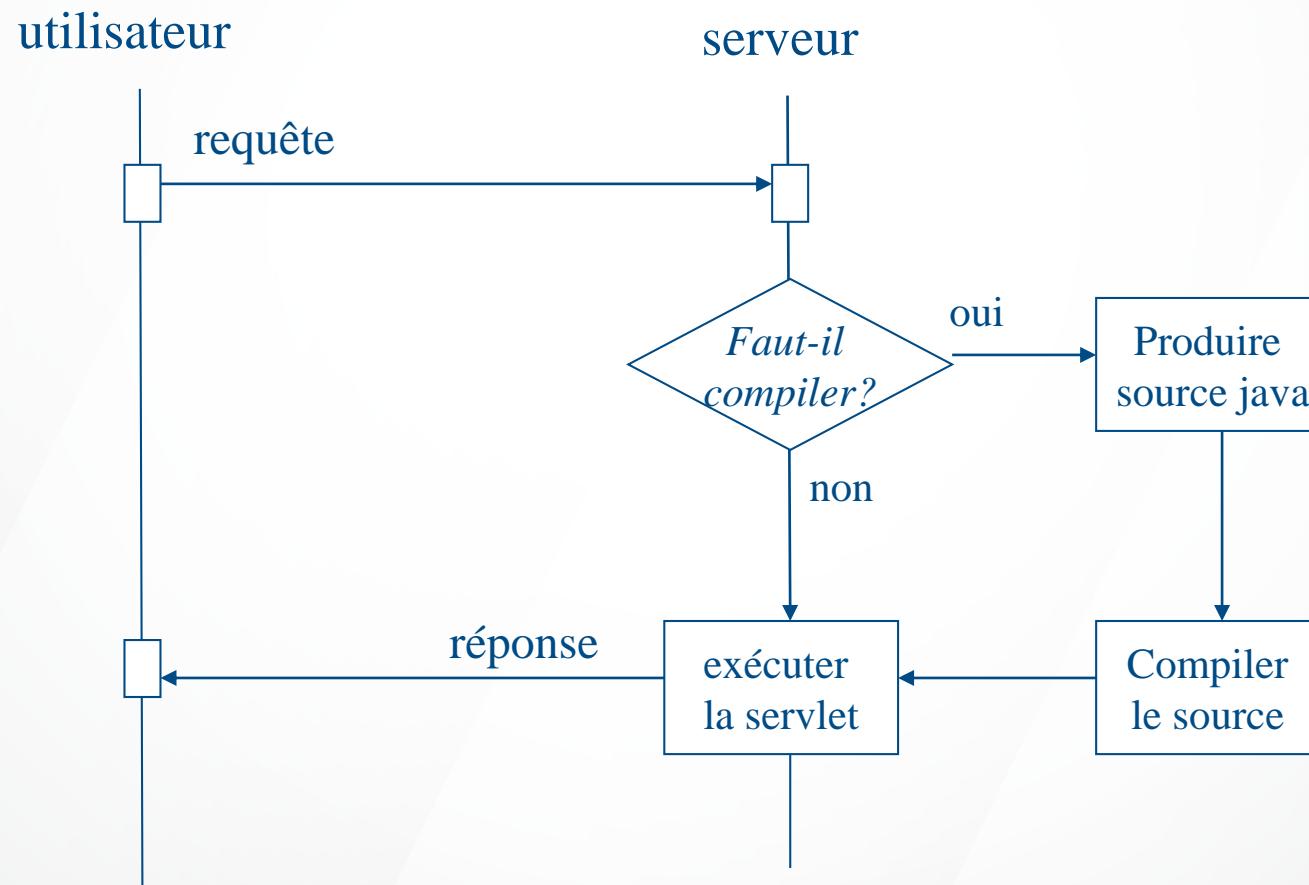
# Jsp comment

- On dépose les pages jsp à coté des pages html
  - Pas de compilation au moment du dépôt
  - Visibilité identique aux pages html
  - url : pas déclaration de service
    - nomDePage.jsp
- Compilation automatique en dynamique sur le serveur WEB



# Déroulement d'un service jsp

- La compilation ne se fait qu'une fois



# Une page jsp s'occupe de la sortie

1. Lire les données expédiées par le demandeur
2. Lire les méta-information sur la demande et le demandeur
3. Calculer la réponse (accès aux SGBD ...)
4. Formater la réponse (généralement en html)
5. Envelopper la réponse http (type de document en retour, cookies ...)
6. Transmettre le document dans un format adapté
  - texte (eg. html)
  - binaire (eg. gif)
  - compressé (eg. gzip)

*Trois étapes simplifiées par jsp*

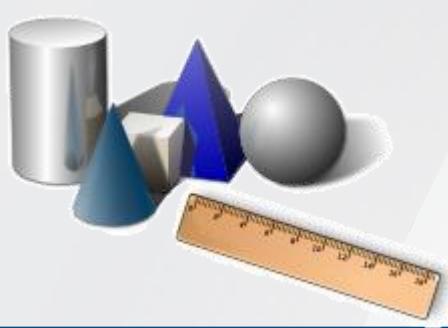
# Qu'est-ce qui est généré ?

---

- Un programme java contenant une classe qui étend l'interface `HttpJspBase`
  - Une interface qui hérite de `JspPage` qui elle même hérite de `Servlet`
  - Son point d'entrée est la méthode `_jspService`

# Interface HttpJspPage

- Le corps de la page jsp correspond au bloc de la méthode `_jspService()`
  - Point d'entrée de la page jsp
  - `void _jspService(HttpServletRequest request, HttpServletResponse response)`
  - les paramètres `request` et `response` sont disponibles au sein de la page
- Les méthodes `jspInit()` et `jspDestroy()` peuvent être redéfinie par l'auteur au sein de la page jsp



# Analyse

- On va voir de près la génération automatique de la page JSP : pagePrincipale.jsp, réalisé dans le Workshop2.
- Pour le faire rendez vous dans le répertoire de travail de Tomcat sous Eclipse qui se trouve généralement sous :
  - <eclipseWorkSpace>\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\work

> .metadata > .plugins > org.eclipse.wst.server.core > tmp0 > work > Catalina > localhost > Gestibank > org > apache > jsp

 pagePrincipale\_jsp.class  
 pagePrincipale\_jsp.java

# Scripts d'une page jsp

## ➤ Elements jsp:

- *Directives*

`<%@ ... %>`

- *Déclarations (attributs et méthodes)*

`<%! Déclarations de variables et de méthodes %>`

- *Scriptlets*

`<% bloc d'instructions %>`

- *Expressions*

`<%= expression java %>`

- *Commentaires (ne sont pas produits en sortie)*

`<%-- expression java --%>`

# Code généré par les scripts JSP

JSP	Code Java généré	Position
<html>	out.println("<html>");	Dans service()
<%! int i = 0; %>	int i = 0;	Méthodes et variable de la servlet (en dehors du service())
<% int i = 0; %>	int i = 0;	Dans service()
<%= i+2 %>	out.println(""+(i+2));	Dans service() afficher la valeur
<%@ ... %>	Include ...	Pour l'ensemble de la page

```
<html>
<head>
<title>titre</title>
</head>
<body>
<%! char c=65; %>
<%
for (int i=0; i<26 ; i++)
{%
Le <%= i+1 %> ième
car est <%= (char)c+i %>.<p>
<% } %>
</body>
```

```
public class exsimple_0002ejspxexsimple_jsp_1
extends HttpJspBase {
char c=65;

public void _jspService(HttpServletRequest
request, HttpServletResponse
response)
throws IOException, ServletException {
response.setContentType("text/html");

out.print(_jspx_html_data[0]);
out.print(_jspx_html_data[1]);

for (int i=0; i<26 ; i++)
{
 out.print(_jspx_html_data[2]);
 out.print(i+1);
 out.print(_jspx_html_data[3]);
 out.print(c+i);
}
out.print(_jspx_html_data[4]);
}
```

# Déclarations

```
<%!

private int i;

public void jspInit() {
 ...
}

public void jspDestroy() {
 ...
}
%>
```

Sert à déclarer des méthodes à l'échelle de la page ( méthode de la servlet générée )

- Les méthodes ne sont exécutées que si elles sont appelées explicitement
- On peut aussi déclarer des attributs

# Scriptlet

```
<%
Iterator i = cart.getItems().iterator();
while (i.hasNext()) {
 ShoppingCartItem item =
 (ShoppingCartItem)i.next();
 BookDetails bd = (BookDetails)item.getItem();
}
%>
```

## Contient du code java

- Tout code Java valide

Se transforme dans la méthode \_jspService() de la servlet

- Les variables sont locales à la méthode
- Les blocs de code s'insèrent dans la servlet

# Scriptlet : La conditionnelle

```
<% if (clock.getHours() < 12) { %>
 matin
<% } else if (clock.getHours() < 14) { %>
 midi
<% } else if (clock.getHours() < 17) { %>
 après-midi
<% } else { %>
 soir
<% } %>
```

# Scriptlet : La conditionnelle (code généré)

```
if (clock.getHours() < 12) {
 out.println("matin");
} else if (clock.getHours()<14) {
 out.println("midi");
} else if (clock.getHours ()<17) {
 out.println("après-midi");
} else {
 out.println("soir");
}
```

# Scriptlet : La boucle

```

<% String[] items = cart.getItems(); %>
 for (int i=0; i<items.length; i++) {
<%>
 <%= items[i] %>
<%>
 }
<%>

```

# Évaluation d'une expression

- Expression dont le résultat est sorti sur le sortie courante de la servlet

```
<%= (12+1)*2 %>
```

- Code généré

```
out.println(""+(12+1)*2);
```

- C'est un raccourci pour le scriptlet : <%out.println(...);%>

# Les commentaires

- Commentaires html ou xml générés dans la page en sortie  
`<!-- blabla -->`
- Commentaires jsp  
Jamais générés dans la page en sortie  
`<%-- blabla --%>`

# Directives

## ➤ Syntaxe

```
<%@ directive attribut="valeur" %>
```

## ➤ Trois directives

page, include, Taglib

## ➤ Exemples

```
<%@ page language="java" %>
<%@ page buffer="5" autoFlush="false" %>
<%@ page import="java.sql.* , cart.*" %>
<%@ include file="foo.jsp" %>
<%@ taglib
 uri="http://java.sun.com/jstl/core"
 prefix="c" %>
```

# La directive page

- Définit des attributs spécifiques à la page
- Chaque attribut ne peut être spécifié qu'une seule fois (sauf l'attribut import)
- Exemple :

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

- Les attributs sont :
  - language (java)
  - Extends
  - import
  - session (true)
  - buffer (8 ko)
  - isErrorPage (false)
  - autoFlush (true)
  - contentType (text/html)
  - isThreadSafe (true)
  - info
  - errorPage

# Objets accessibles automatiquement au sein d'une page jsp

Nom de variable	Type java
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
out	javax.servlet.jsp.JspWriter
pageContext	javax.servlet.jsp.PageContext
config	javax.servlet.ServletConfig
page	java.lang.Object ( <b>HttpJspPage</b> )
exception	java.lang.Throwable

# Stratégie de conception: Limiter le code Java dans les JSP

- Deux options
  - Ecrire 25 lignes de code directement dans une JSP
  - Ecrire ces 25 lignes dans une classe Java à part et 1 ligne dans une JSP pour l'invoquer
- Pourquoi la 2 option est vraiment meilleure?
  - **Développement.** Ecriture de la classe dans un environnement Java et pas HTML
  - **Debugage.** S'il y a des erreurs, elles sont visible à la compilation
  - **Test.** L'accès à la classe Java facilite le test (ex: boucle de test de 100000 itérations sur un serveur...)
  - **Réutilisation.** Utilisation de la même classe dans différentes pages JSP

# Versions JSP

- Extensions de fichiers
  - .jsp, page JSP standard
  - .jspx, fragment de page JSP
  - .jspx, page JSP compatible XML
- Deux syntaxes
  - Standard (JSP 1.2)
  - XML (JSP 2.0)
- Depuis la 2.0 : Extensible à travers des librairies de tag (fichier .tld) importés ou personnels.

# Notion de context en JSP

- On a quatre contextes d'exécution imbriqués les uns dans les autres
- Application>session>requête>page
  - **Page** : Les variables déclarées dans un scriptlet sont locales à la page
  - **Requête** : Pages successives collaborant au traitement d'une même requête (voir "action" <jsp:forward> plus loin)
  - **session** : requêtes collaborant à une même tâche
  - **application** : un ensemble de requêtes regroupées sous un même thème

# "actions" au sein de pages jsp

- **Syntaxe**
  - Eléments XML "actifs" au sein d'une page JSP

```
<prefixe:action attr="..."> ... </prefixe:action>
```
- **Effet**
  - Exécuter du code à chaque passage
- **Deux sortes d'actions**
  - Actions standards (*prefixe jsp*) exemple `<jsp:action>`
  - Actions définies par l'utilisateur : JSP tags (*autres prefixes*)
- **Les actions sont regroupées par librairies**

# Actions JSP standard

## ➤ Modularité

<jsp:include> inclure un fichier en place

<jsp:forward> transférer le contrôle à une autre page

## ➤ Utiliser les beans

<jsp:useBean> trouver (instancier) un bean

<jsp:getProperty> valeur d'un attribut de bean

<jsp:setProperty> initialise un attribut de bean

## ➤ Appel d'une applet

<jsp:plugin> génère du code <OBJECT> ou <EMBED>

# Actions JSP standard : modularité

- A quoi cela sert
  - Récupérer les erreurs
  - Simplifier le contrôle
  - ou simplement séparer contrôle et traitements
  - Modulariser les programmes
- Quels outils sont nécessaires
  - Des outils pour passer le contrôle d'une page à une autre
  - Des outils pour passer de l'information d'une page à l'autre

# Actions JSP standard : modularité : <jsp:forward>

- Action <jsp:forward>  
`<jsp:forward page="page1.jsp" />`
- Attribut page
  - Référence relative à la racine de l'application  
`page="/rep/page1.jsp"`
  - Référence relative à la page d'appel  
`page="rep/page1.jsp"`
- Effet : arrête l'exécution de la page courante, lance l'exécution d'une autre page

# Actions JSP standard : modularité : <jsp:forward> (2)

- Effets (suite)
  - L'URI d'appel reste inchangée (en cas de rechargement dans le navigateur, c'est la page d'appel qui est rechargée ...)
  - Les variables request, result ... restent accessibles
  - Le buffer de sortie est réinitialisé (si le transfert avait déjà commencé, une erreur est générée)
- Passer de l'information entre pages :

```
<jsp:forward page="myPage.jsp">
 <jsp:param name="nom" value="val" />
</jsp:forward>
```

# Actions JSP standard : modularité : <jsp:forward> Récupération d'erreurs

```
<% try {
...
} catch(Exception e) {
%>

<jsp:forward page="erreur.jsp">
 <jsp:param name="titre" value="Erreur pendant la lecture de la
base" />

 <jsp:param name="message"
 value="<%= e.getMessage() %>" />

</jsp:forward>
<% } %>
```

# Actions JSP standard : modularité

## <jsp:include>

- Inclure en ligne la réponse d'une autre page jsp, servlet ou page statique
  - L'URI reste celle de la page d'appel
  - Les informations sur la requête restent inchangées
  - Le buffer de sortie est vidé sur la sortie avant de commencer
  - Des paramètres peuvent être passés

```
<jsp:include page="nav.jsp">
 <jsp:param name="nom" value="val"/>
</jsp:include>
```

# Actions JSP standard : modularité

## <jsp:include> (2)

- Il y a deux mode d'inclusion :
  - INCLUSION STATIQUE
  - INCLUSION Dynamique
- INCLUSION STATIQUE
  - Insère le code source d'une ressource (fichier .html, .jsp, ...) dans la page JSP
    - Juste avant la phase de compilation la page englobante et la page incluse ne constituent qu'une seule servlet générée
    - La page incluse voit les imports java qui ont été déclarés dans la page englobante
- Attribut "file" URL relative de la ressource à insérer

```
<%@ include file="entete.jsp"%>
```

# Actions JSP standard : modularité

## <jsp:include> (3)

### ➤ INCLUSION DYNAMIQUE

- Insère le résultat de l'exécution d'une ressource (fichier .html, .jsp, ...) dans la page JSP

- Lors de la phase de traitement de la requête La jsp incluse génère sa propre servlet
  - La page incluse ne voit pas les imports déclarés dans la page enveloppante

```
<jsp:include page="entete.jsp" flush="true"/>
```

- Syntaxe

- page : URL relative de la ressource à insérer
  - flush (booléen) : s'il est positionné à true, permet un affichage progressif de la page sur le poste client (le buffer est vidé après l'inclusion)

# INCLUSION STATIQUE OU DYNAMIQUE ?

- Les inclusions statiques sont plus performantes
  - Moins de servlets à gérer
- Les inclusions dynamiques se rafraîchissent mieux
- Les inclusions dynamiques permettent un affichage progressif de la page sur le poste client
  - Attribut flush="true"

# Actions JSP standard : modularité

## <jsp:plugin>

- Génération d'un élément <embed> ou <object> (en fonction du navigateur utilisé)

```
<jsp:plugin type="applet"
 code="Clok.class" jreversion="1.2"
 width="160 value="150">
 <jsp:params>
 <jsp:param name="nom" value="val"/>
 </jsp:param>
</jsp:plugin>
```

# Les Beans - conventions

- bean = classe qui respecte des conventions

```
public class MonBean {
 // Attributs privés
 private type XXX;
 // constructeur
 public MonBean() { ... }
 ...
 // Lecture/écriture d'un attribut
 type getXXX() { ... }
 setXXX(type val) { ... }
}
```

*Attribut caché*

*Constructeur  
(sans paramètre)*

*Visibilité  
gérée par des  
méthodes d'accès*

# Les Beans ou JavaBeans

- Les JavaBeans sont des classes Java(POJO) qui suivent certaines conventions:
  - Doivent avoir un constructeur vide (zéro paramètre)
    - => On peut satisfaire cette contrainte soit en définissant explicitement un tel constructeur, soit en ne spécifiant aucun constructeur
  - Ne doivent pas avoir d'attributs publics
    - => Une bonne pratique réutilisable par ailleurs...
  - La valeur des attributs doit être manipulée à travers des accesseurs (getters et setters)
    - => Si une classe possède une méthode getTitle qui retourne un String, on dit que le bean possède une propriété String nommée title
    - => Les propriétés Boolean utilisent isXXX à la place de getXXX

# Les Beans ou JavaBeans

## ➤ Pourquoi il faut utiliser les accessors

- 1) On peut imposer des contraintes sur les données

```
public void setSpeed(double newSpeed) {
 if(newSpeed < 0){
 SendErrorMessage(...);
 newSpeed = Math.abs(newSpeed);
 }
 speed = newSpeed;
}
```

- 2) On peut changer de représentation interne sans changer d'interface

```
public void setSpeed(double newSpeed) {
 speed = convertKMTomiles(newSpeed);
}
```

- 3) On peut rajouter du code annexe

```
public double setSpeed(double newSpeed) {
 speed = newSpeed;
 updateSpeedometerDisplay();
}
```

# Comment utiliser des beans

- Une fois la classe créée, les beans sont des objets Java sur lesquels on peut faire ces actions :
  - instantiation d'un nouveau bean
  - récupération de la valeur d'une propriété du bean
  - affectation/modification de la valeur d'une propriété du bean
- Pour faire cela dans une Servlet, ça ne pose pas de problème particulier,
  - le bean est traité comme un objet Java standard.
- Dans une JSP, l'utilisation des balise de scriptlets n'est pas très « propre », on utilisera plutôt l'une des manière suivantes :
  - balises JSP (JSP 1.2) : <jsp:useBean>, <jsp:getProperty>, <jsp:setProperty>
  - utilisation des EL (JSP 2.0) (si le serveur le supporte) : \${...} (on va voir en détails les EL )

# Actions jsp standards – les "beans"

## ➤ Trois actions standards pour les beans

```
<jsp:useBean id="XXX" class="..."/>
<jsp:getProperty name="XXX" property="..."/>
<jsp:setProperty name="XXX" property="..." value="..."/>
```

## ➤ Trois éléments vides (en général) :

- usage des paramètres pour passer l'information
- Attention `xxx` doit être un identificateur java valide

## ➤ Trois traitements correspondants

- *Déclaration/création d'une instance d'objet*  
(constructeur sans paramètre)
- *Lecture d'un attribut de classe*
- *Écriture d'un attribut de classe*

# Création d'un bean

- Crédit d'un bean dans une page jsp

```
<jsp:useBean id="horloge"
 class="myPackage.DateFormatee"/>
```

- Code Généré : deux fonctions
  - Crédit d'une instance d'objet

```
myPackage.DateFormatee horloge = null;
horloge = new myPackage.DateFormatee();
```

- Réutilisation du bean : le bean est sauvé par défaut dans le contexte de la page courante

# Contexte d'un bean

- Quatre contextes d'exécution imbriqués (rappel)  
Page<requête<session<application
- Action <jsp:useBean>  
<jsp:useBean id="**ident**" scope="**request**"  
    class="lib.Classe"/>
- Si l'objet bean existe déjà dans la portée on utilise celui-ci et on ne crée pas de nouvel objet

Attribut scope



# Portée (attribut scope)

- scope="page"
  - objet est visible dans la page (valeur par défaut)
- scope="request"
  - objet est visible au fil des jsp:forward
- scope="session"
  - objet est visible pour la session
- scope="application"
  - objet est visible pour toute l'application

# Attribut @class ou @type

- Au moins un attribut class ou type doit être présent
  - `class="nom"` : Un nom complet de classe
  - `type="nom"` : Un nom complet de super classe de l'objet ou d'interface implémentée

# Elément <jsp:useBean> avec contenu

Le corps de l'élément est exécuté une seule fois lors de la création de l'instance de l'objet bean

```
<jsp:useBean id="myBean" class="myClass">
 <jsp:setProperty name="myBean"
 property="prop" value="val"/>
</jsp:useBean>
```

# <jsp:useBean> avec contenu

```
<%@ page language="java" %>
<html><body>
 <h1>Utilisation du Bean
 <code>String</code></h1>

<jsp:useBean id="monTexte" class="String">
 <p>On crée une chaîne nommée monTexte</p>
 <%monTexte = "blabla";%>
</jsp:useBean>

<p>monTexte = <%= monTexte%></p>
</body></html>
```



# Exemple de Bean - "date courante formatée"

```
package myPackage;

public class DateFormatee {
 static SimpleDateFormat sdf =
 new SimpleDateFormat(
 "EEEEEEEEE dd MMMMMMMMM YYYY");
 private Date dateCreation;
 public DateFormatee () {
 dateCreation = new Date();
 }
 static public String getDate() {
 return sdf.format(new Date());
 }
 public String getDateCreation() {
 return sdf.format(dateCreation);
 }
}
```

Attention  
« default package » interdit

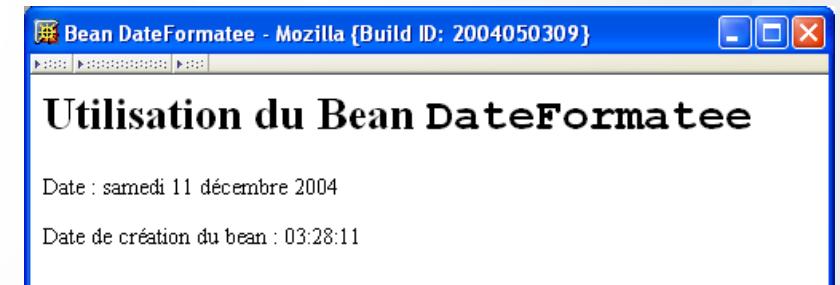
Attention les attributs static  
ne sont pas des propriétés du bean

*Une seule propriété*

# Utilisation du bean

```
<%@ page language="java" %>
<html><body>
 <h1>Utilisation du Bean DateFormatee</h1>
 <p>Date :
 <%= myPackage.DateFormatee.getDate() %>
 </p>
 <jsp:useBean id="horloge"
 class="myPackage.DateFormatee"/>
 <p>Heure de création :
 <jsp:getProperty name="horloge"
 property="dateCreation"/>
 </p>
</body></html>
```

*Attention les attributs static  
ne sont pas des propriétés du bean*



# Elément <jsp:setProperty>

## Trois formes d'appel

- Valeur directe

```
<jsp:setProperty name="XXX" property="..." value="chaîne"/>
```

- Valeur résultante du calcul

```
<jsp:setProperty name="XXX" property="..." value="<%=\n expression %>">/>
```

- Valeur provenant d'un paramètre saisi dans un formulaire

```
<jsp:setProperty name="XXX" property="prop" Param="param"/>\n(prop:=param)
```

# Elément <jsp:setProperty>

```
<jsp:setProperty name="XXX" property="..." value="..."/>
```

- Le type d'une propriété est quelconque
- Si la valeur est une chaîne une conversion implicite est opérée

# Conversions implicites de chaîne vers un type primitif de Java

Type de propriété	Méthode de conversion
Boolean ou boolean	Boolean.valueOf(String)
Byte ou byte	Byte.valueOf(String)
Character ou char	String.charAt(int)
Double ou double	Double.valueOf(String)
Integer ou int	Integer.valueOf(String)
Float ou float	Float.valueOf(String)
Long ou long	Long.valueOf(String)

# Initialisation d'un "bean"

- *Condition* : chaque paramètre récupéré dans le formulaire possède un attribut de même nom dans la classe bean

```
<jsp:useBean id="monObjet"
 class="lib.MaClasse">

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

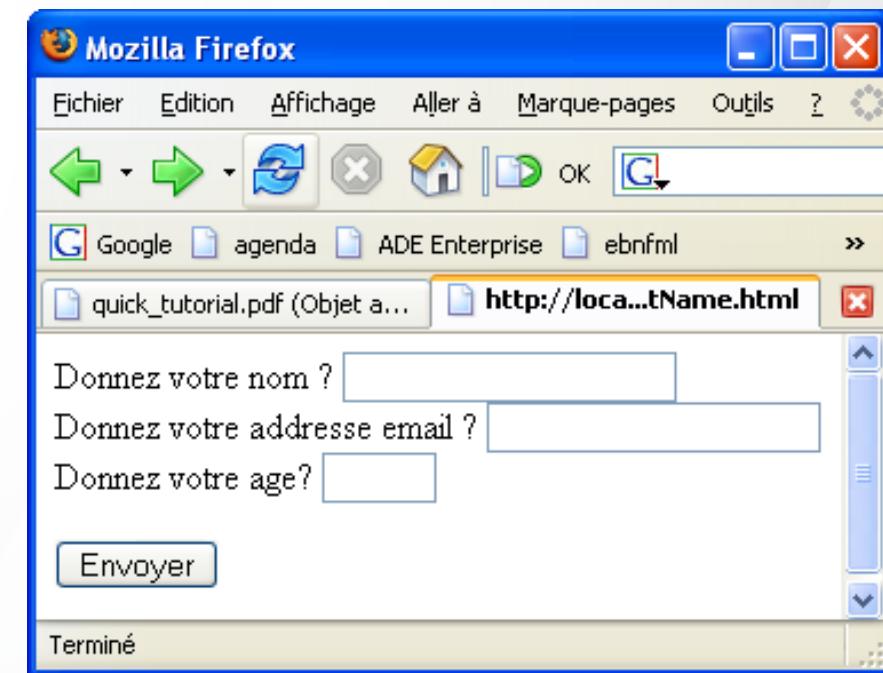
Utilisation de \*

## ExempleFormulaire de saisie : getName.html

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="saveName.jsp">
 Donnez votre nom ? <INPUT TYPE=TEXT
 NAME="username" SIZE=20>

 Donnez votre adresse email ? <INPUT TYPE=TEXT
 NAME="email" SIZE=20>

 Donnez votre age? <INPUT TYPE=TEXT
 NAME="age" SIZE=4>
 <P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```



# Java beans (cf nom des attributs)

```
public class UserData {
 String username;
 String email;
 int age;

 public void setUsername(String value) {
 username = value;
 }
 public void setEmail(String value) {
 email = value;
 }
 public void setAge(int value) {
 age = value;
 }
}
```

```
public String getUsername() {
 return username;
}
public String getEmail() {
 return email;
}
public int getAge() {
 return age;
}
}
```

*Les getters et setters doivent être public, le nom des attribut est le même que dans le formulaire, la classe ne doit pas être dans le paquetage par défaut*

# Récolter les valeurs, page JSP

## saveName.jsp

```
<jsp:useBean id="user" class="test.UserData"
 scope="session"/>

<jsp:setProperty name="user" property="*"/>

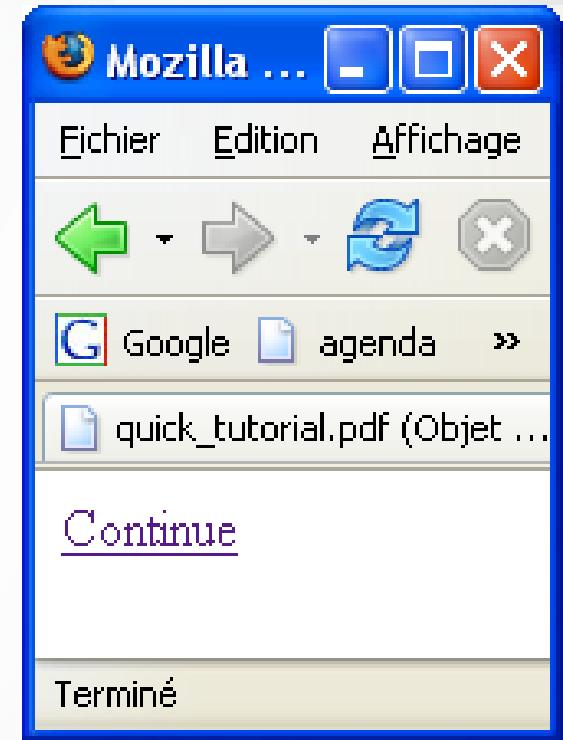
<HTML>

<BODY>

Continue

</BODY>

</HTML>
```



# Afficher les valeurs : nextPage.jsp

```
<jsp:useBean id="user" class="test.UserData"
 scope="session"/>

<HTML>

<BODY>

You entered

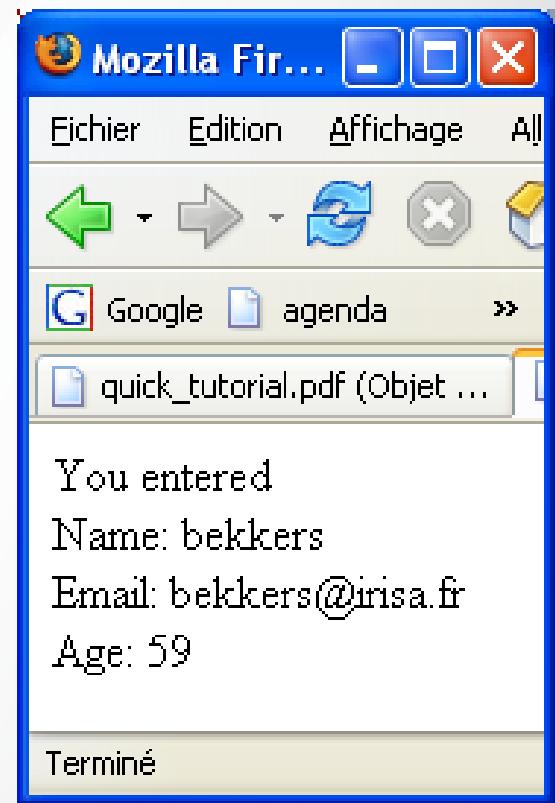
Name: <%= user.getUsername () %>

Email: <%= user.getEmail () %>

Age: <%= user.getAge () %>

</BODY>

</HTML>
```



# Java Standard Tag Library (JSTL)

- De nombreux frameworks facilitent le développement d'application Java EE. **JSTL** propose une librairie standard pour la plupart des fonctionnalités de base d'une application Java EE.
- Le but de la JSTL est de simplifier le travail des auteurs de page JSP, c'est à dire la personne responsable de la couche présentation d'une application web J2EE.
  - En effet, un web designer peut avoir des problèmes pour la conception de pages JSP du fait qu'il est confronté à un langage de script complexe qu'il ne maîtrise pas forcément.
- La JSTL permet de développer des pages JSP en utilisant des balises XML, donc avec une syntaxe proche des langages utilisés par les web designers, et leur permet donc de concevoir des pages dynamiques complexes sans connaissances du langage Java.
- La JSTL se base sur l'utilisation des EL en remplacement des scriptlets Java.
- Il existe différentes versions des JSTL. La plus récente se base sur les JSP 2.x qui intègre un moteur d'EL (Expression Language ).

# Java Standard Tag Library (JSTL)

## ➤ Utilisation:

- Pour utiliser les JSTL, il suffit de copier dans le répertoire WEB-INF/lib de votre application WEB les fichiers **jstl.jar** et **standard.jar** qui se trouvent dans le répertoire lib de la distribution JSTL.
- Les JSTL disposent d'un ensemble de librairies de fonctionnalités différentes
- Pour inclure une librairie jstl à une page JSP il faut ajouter une déclaration en tête de page (directive taglib) de cette façon:  
`<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>`
  - Cette ligne permet d'inclure la librairie "core" de la JSTL qui contient les actions de base d'une application web.

## ➤ Voici une liste des différentes librairies existantes :

Librairie	URI	Préfixe
Core	<a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>	c
Format	<a href="http://java.sun.com/jsp/jstl/fmt">http://java.sun.com/jsp/jstl/fmt</a>	fmt
XML	<a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a>	x
SQL	<a href="http://java.sun.com/jsp/jstl/sql">http://java.sun.com/jsp/jstl/sql</a>	sql
Fonctions	<a href="http://java.sun.com/jsp/jstl/functions">http://java.sun.com/jsp/jstl/functions</a>	fn

# Exemple gestion des variables

## ➤ Initialiser

```
<c:set var="bookId"
 value="${param.Remove}"/>
```

## ➤ Effacer

```
<c:remove var="cart"
 scope="session
```

## ➤ Référencer

`${bookId}`

# Affichage d'une liste

```

<% List bookList = (List) request.getAttribute ("books-list");
 if (bookList!=null) {
 Iterator iterator = bookList.iterator();
 int i = 0;
 while (iterator.hasNext()) {
 Book b = (Book) iterator.next();
 out.print ("<li class='"+(i%2==0?"pair":"impair")+">");
 out.print (b.getName()+" ("+b.getPrice()+" €)");
 if (b.getPrice() < 30.0)
 out.print (" <img src='hot.png'
 alt='Moins de 30 €' />");
 out.println ("");
 i++;
 }
 }
%>

```

# Utiliser la librairie core

```


<c:forEach items="${requestScope['books-list']}" var="book"
 varStatus="status">

 <li class="${status.index%2==0?'pair':'impair'}">
 ${book.name} (${book.price} €)

 <c:if test="${book.price < 30.0}">
 <img src='hot.png'
 alt='Moins de 30 €' />
 </c:if>

</c:forEach>


```

# Utiliser la librairie core (bis)

```
<%
 String [] liste = Personne.listIds();
 pageContext.setAttribute ("liste", liste);
%>
<select name="personneId" size="<%= liste.length %>">
 <optgroup label="">
 <c:forEach items="${liste}" var="item" >
 <option value="${item}">
 <%= Personne.get((String)pageContext.getAttribute("item")).getNom() %>
 </option>
 </c:forEach>
 </optgroup>
</select>
```

# Conditionnelle à choix multiple

```
<c:choose>
 <c:when test="${customer.category=='trial'}" >
 ...
 </c:when>
 <c:when test="${customer.category=='member'}" >
 ...
 </c:when>
 <c:when test="${customer.category=='preferred'}" >
 ...
 </c:when>
 <c:otherwise>
 ...
 </c:otherwise>
</c:choose>
```

# Éléments actifs « xml »

Noyau	out, parse, set
Gestion du contrôle	choose (when, otherwise), forEach , if
Transformation	transform (param) ,

## Evaluations xpath

```
<x:forEach var="book"
select="$applicationScope:booklist/books/*">
...
</x:forEach>
```

# URL et documents xml

## ➤ Importer un document

```
<c:import url="/books.xml" var="xml" />
```

## ➤ Compiler le document

```
<x:parse doc="${xml}" var="booklist" scope="application" />
```

# i18n

```
<h3><fmt:message key="Choose"/></h3>
```

Définir de la locale	setLocale requestEncoding
Gestion des messages	Bundle, message param, setBundle
Nombres et dates	formatNumber, formatDate parseDate, parseNumber setTimeZone, timeZone

# Exemple1

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

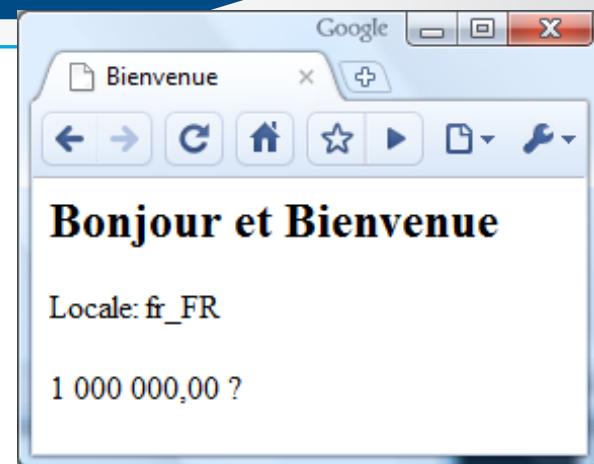
<html><fmt:setBundle basename="message"/>
<head><title><fmt:message key="Welcome" />
</title></head><body>

<h2><fmt:message key="Hello" /> <fmt:message key="and" /> <fmt:message
key="Welcome" /></h2>

Locale:
${pageContext.request.locale.language}_${pageContext.request.locale.country}

<fmt:formatNumber value="1000000" type="currency" />

</body></html>
```



# Properties file

- message.\_fr.properties placé à la racine des sources de type :  
`<clé>=<valeur>`

```
Welcome=Bienvenue
Hello=Bonjour
and=et
```

# Exemple i18n

```
<jsp:useBean id="now" class="java.util.Date" />

<jsp:setProperty name="now"
 property="time" value="${now.time + 432000000}" />

<fmt:message key="ShipDate"/>

<fmt:formatDate value="${now}"
 type="date" dateStyle="full"/>.
```

# EL : Expression Language

- *Manipuler des données plus simplement qu'en Java*

# Utilisation dans les attributs de la librairie standard JSTL

Langage simple et flexible d'accès aux données \${...} pour encadrer les expressions dynamiques

Tout objet présent au moins dans le scope de la JSP peut être utilisé comme variable dans une expression EL

- Pas de typage
- Conversions automatiques (nombres → String)
- Evaluation des expressions à l'exécution

```
<c:out value="${expression}" />
```

# Où utiliser les EL

- Dans les attributs des tags JSP.
- Dans du texte simple de la page JSP.
  - Exemple

```
 ${monBean}
<prefix:actionTag id="newBean" param="${monBean}">
 ${newBean}
</prefix:actionTag>
```

# Quelques exemples

- `${ pageContext.response.contentType }` 
  - affiche le content type de la réponse.
- `${ pageScope["name"] }` 
  - affiche l'attribut "name" du scope page.
- `${ param["page"] }` 
  - affiche la valeur du paramètre "page".
- `${ header["user-agent"] }` 
  - affiche l'header "user-agent" envoyé par le navigateur.

# Les objets implicites

- **pageContext** : Accès à l'objet **PageContext** de la page JSP.
- **pageScope** : Map permettant d'accéder aux différents attributs du scope '**page**'.
- **requestScope** : Map permettant d'accéder aux différents attributs du scope '**request**'.
- **sessionScope** : Map permettant d'accéder aux différents attributs du scope '**session**'.
- **applicationScope** : Map permettant d'accéder aux différents attributs du scope '**application**'.
- **param** : Map permettant d'accéder aux paramètres de la requête HTTP sous forme de **String**.
- **paramValues** : Map permettant d'accéder aux paramètres de la requête HTTP sous forme de **tableau de String**.
- **header** : Map permettant d'accéder aux valeurs du Header HTTP sous forme de **String**.
- **headerValues** : Map permettant d'accéder aux valeurs du Header HTTP sous forme de **tableau de String**.
- **cookie** : Map permettant d'accéder aux différents Cookies.
- **initParam** : Map permettant d'accéder aux **init-params** du web.xml.

# Recherche automatique d'un attribut

- \${ name }
  - Rechercher l'attribut "name" successivement dans les différentes portées (dans l'ordre : **page, request, session, application**).

# Opérateurs

- Arithmétiques +, -, \*, / (ou div), % (ou mod)
- Relationnels == (ou eq), != (ou ne), < (ou lt), > (ou gt), <= (ou le), >= (ou ge)
- Logiques && (ou and), || (ou or), ! (ou not [ . . . ])

# Autres opérateurs

- **Empty [ \* ]**  
**true** si l'opérande est **null**, une chaîne vide, un tableau vide, une Map vide ou une List vide. **false** sinon.
- **(c?v1:v2)**  
**conditionnelle**

# Écritures équivalentes

- \${ bean.name }
- \${ bean["name"] }
- \${ bean['name'] }
- La valeur entre 'crochet' est une chaîne mais aussi n'importe quel objet.
- La méthode **toString()** est utilisé pour forcer le résultat à être une chaîne.
- \${ bean[ config.propertyName ] }

# Exemple d'écritures équivalentes

- \${ person.address.city }
- \${ person['address']['city'] }
- \${ person["address"]["city"] }
- \${ person.address['city'] }
- \${ person["address"].city }

# Listes, tableaux et map

## ➤ Listes et tableaux

- \${ list[0] }
- \${ list[1] }
- \${ list["2"] }
- \${ list["3"] }
- \${ list[ config.value ] }

## ➤ Map

- \${ map["clef1"] }
- \${ map['clef2'] }
- \${ map[ config.key ] }

# Prise en compte des expressions

- Dans une directive **page** ou dans un **tag** .

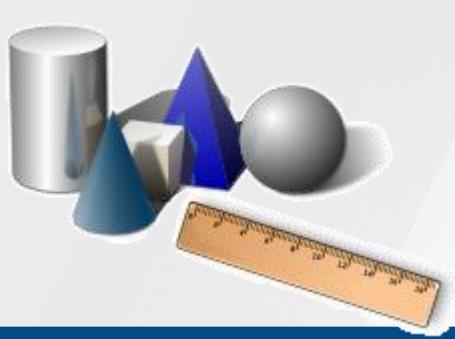
- Utiliser l'attribut **isELIgnored**

```
<%@ page isELIgnored="false" %>
```

**Attention** elles sont ignorées par défaut ...

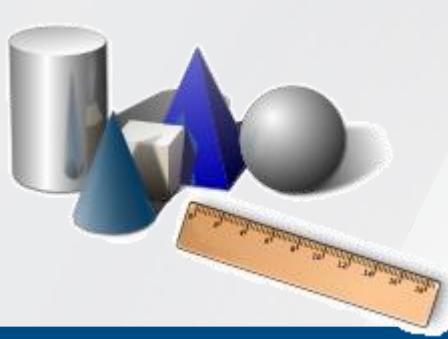
- Utiliser anti-slash pour signifier que ceci n'est pas une expression:

```
\${ ceci n'est pas une EL }
```



## Workshop 5 :

- Dans ce workshop on va déclarer Les javabean et la couche DAO.
- On va aussi utiliser le JSTL dans les pages JSP
  - Pour commencer récupérer le projet : wkshop4\_Start
  - Regarder de près le package Model et l'interface BanqueService.java
  - Ajouter la librairie de JSTL au projet :
    - jstl-1.2\_1.jar



## Workshop 5 :

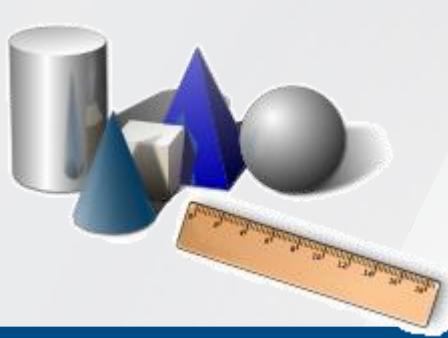
- On va Ajouter un service bouchon qui simule la recherche d'un client par mots de passe

```
public class BanqueServiceBouchon implements BanqueService {

 public Client findClient(String login, String motDePasse) throws GestiBankException {
 Client c = new Client(10, "bouchon", "john");

 ArrayList courantList = new ArrayList();
 courantList.add(new CompteCourant(12, "compte courant bouchon", 200, 1000));
 c.setCompteCourantList(courtantList);

 ArrayList epargneList = new ArrayList();
 epargneList.add(new CompteEpargne(13, "compte epargne bouchon", 1000, 0.5f, 500));
 c.setCompteEpargneList(epargneList);
 return c;
 }
}
```



# Workshop 5 :

## ➤ On va modifier PagePrincipale.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 pageEncoding="ISO-8859-1"%>
<!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>Menu GESTIBANK</title>
 <%@page language="java" import="com.wha.model.Client" %>
</head>
<body>
 <% Client client = (Client) session.getAttribute("client");%>
 <h2> Bonjour <%= client.getPrenom() %> <%= client.getNom() %> </h2>
 <p> Opérations disponibles </p>

 <a href="<%=request.getContextPath()%>/compteCourantListe">Compte Courant
 <% if (!client.getCompteEpargneList().isEmpty()) {%
 <a href="<%=request.getContextPath()%>/compteEpargneListe">Compte Epargne
 %}>

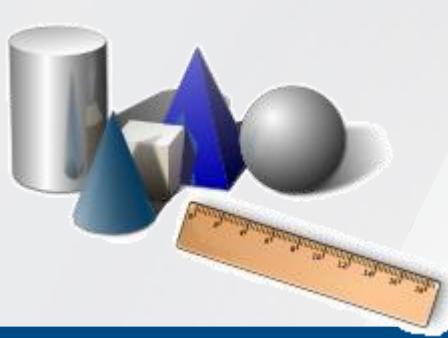
 <a href="<%=request.getContextPath()%>/fin">Déconnection
</body>
</html>
```

**Bonjour john bouchon**

Opérations disponibles

- [Compte Courant](#)
- [Compte Epargne](#)

[Déconnection](#)



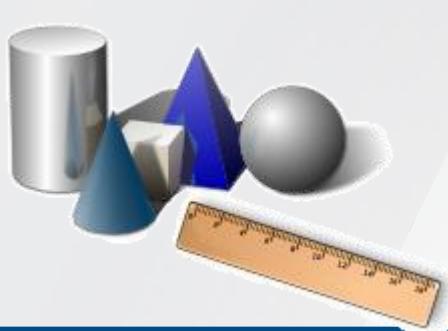
## Workshop 5 :

- Ajouter les 2 servlets d'affichages pour le compte courant et le compte épargne

```
@WebServlet("/compteCourantListe")
public class CompteCourantServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;

 /**
 * @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)
 */
 protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

 HttpSession session = request.getSession(false);
 if (session != null)
 {
 response.sendRedirect(request.getContextPath() + "/compteCourantListe.jsp");
 }
 else
 {
 response.sendRedirect(request.getContextPath() + "/login.jsp");
 }
 }
}
```

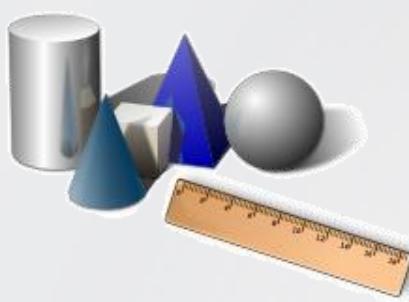


# Workshop 5 :

```
@WebServlet("/compteEpargneListe")
public class CompteEpargneServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;

 /**
 * @see HttpServlet#service(HttpServletRequest request, HttpServletResponse response)
 */
 protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

 HttpSession session = request.getSession(false);
 if (session != null)
 {
 response.sendRedirect(request.getContextPath() + "/compteEpargneListe.jsp");
 }
 else
 {
 response.sendRedirect(request.getContextPath() + "/login.jsp");
 }
 }
}
```



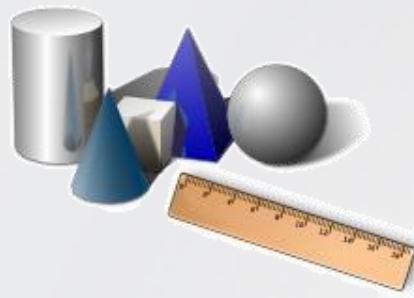
# Workshop 5 :

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 pageEncoding="ISO-8859-1"%>
<!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>
 <%@page language="java" import="com.wha.model.Client" %>
 <%@page language="java" import="com.wha.model.Compte" %>
 <%@page language="java" import="com.wha.model.CompteCourant" %>
 <%@page language="java" import="java.util.Iterator" %>
</head>
<body>
 <h1>Comptes courants</h1>
 <% Client client = (Client) session.getAttribute("client");
 Iterator ccIt = client.getCompteCourantList().iterator();
 %>
 <table cellspacing="5px" style="border:solid 1px black;">
 <tr>
 <td style="border:solid 1px black;">Intitulé</td>
 <td style="border:solid 1px black;">Solde</td>
 <td style="border:solid 1px black;">Découvert autorisé</td>
 </tr>
 <% while (ccIt.hasNext()) {
 CompteCourant cc = (CompteCourant)ccIt.next();%>
 <tr>
 <td style="border:solid 1px black;"> <%= cc.getLibelle()%></td>
 <td style="border:solid 1px black;"> <%= cc.getSolde()%></td>
 <td style="border:solid 1px black;"> <%= cc.getDecouvertAutorise()%></td>
 </tr>
 <% }%>
 </table>
 <A href= "<%=request.getContextPath() + "/pagePrincipale.jsp"%>">Menu principal
</body>
</html>
```

## Comptes courants

Intitulé	Solde	Découvert autorisé
compte courant bouchon	200.0	1000.0

[Menu principal](#)



# Workshop 5 :

## Comptes Epargne

Intitulé	Solde	Taux d'intérêt	Plafond
compte epargne bouchon	1000.0	0.5	500.0

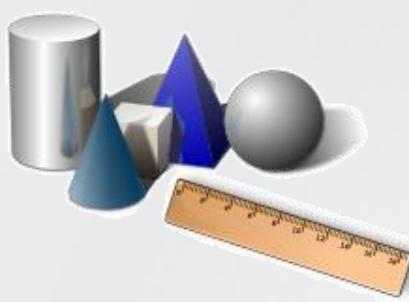
[Menu principal](#)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 pageEncoding="ISO-8859-1"%>
<!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>
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
</head>
<body>
 <h1>Comptes Epargne</h1>
 <table border=1;>
 <tr>
 <td>Intitulé</td>
 <td>Solde</td>
 <td>Taux d'intérêt</td>
 <td>Plafond</td>
 </tr>
 <c:forEach items="${client.compteEpargneList}" var="tempclient">
 <tr>
 <td> <a href= "<c:url value="/detailCompteEpargne">
 <c:param name="CompteId" value="${tempclient.identifiant}" />
 </c:url>">
 <c:out value="${tempclient.libelle}" />

 </td>
 <td> <c:out value="${tempclient.solde}" /> </td>
 <td> <c:out value="${tempclient.tauxInteret}" /> </td>
 <td> <c:out value="${tempclient.plafond}" /> </td>
 </c:forEach>
 </table>

 <a href= "<%=request.getContextPath()%>"+"/pagePrincipale.jsp">Menu principal
 </body>
</html>
```

compteEpargneList.jsp



# Workshop 5 :

- La page compteEpargneList permet d'avoir les détails du compte épargne en appelant la servlet associé

```
@WebServlet("/detailCompteEpargne")
public class DetailCompteEpargneServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;

 protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
 int CompteID= Integer.parseInt(request.getParameter("CompteID"));
 System.out.println("compteID "+CompteID);
 HttpSession session = request.getSession(false);
 Client client = (Client) session.getAttribute("client");
 Iterator ceIt = client.getCompteEpargneList().iterator();
 while (ceIt.hasNext())
 {
 CompteEpargne tempcompte = (CompteEpargne) ceIt.next();
 System.out.println(tempcompte.getIdentifiant());
 if(tempcompte.getIdentifiant()==CompteID)
 {
 session.setAttribute("compte", tempcompte);
 break;
 }
 }
 System.out.println("Compte : "+(CompteEpargne)session.getAttribute("Compte"));
 response.sendRedirect(request.getContextPath() + "/detailCompteEpargne.jsp");
 }
}
```

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!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>
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
 <%@page language="java" import="com.wha.model.Client" %>
 <%@page language="java" import="com.wha.model.Compte" %>
 <%@page language="java" import="com.wha.model.CompteEpargne" %>
</head>
<body>
 <h1>Détail du compte épargne:</h1>
 <table border=1;>
 <tr>
 <td>Identifiant</td>
 <td><c:out value="${compte.identifiant}"></c:out></td>
 </tr>
 <tr>
 <td>libelle</td>
 <td><c:out value="${compte.libelle}"></c:out></td>
 </tr>
 <tr>
 <td>Solde</td>
 <td><c:out value="${compte.solde}"></c:out></td>
 </tr>
 <tr>
 <td>Taux d'intérêt</td>
 <td><c:out value="${compte.tauxInteret}"></c:out></td>
 </tr>
 <tr>
 <td>Plafond des échéances</td>
 <td><c:out value="${compte.plafond}"></c:out></td>
 </tr>
 <c:remove var="compte"/>
 </table>
 <A href= "<%=request.getContextPath() + "/pagePrincipale.jsp" %>">Menu principal
</body>
</html>

```

detailCompteEpargne.jsp

## Détail du compte épargne:

Identifiant	13
libelle	compte epargne bouchon
Solde	1000.0
Taux d'intérêt	0.5
Plafond des échéances	500.0

[Menu principal](#)

# Ajout du client dans la session

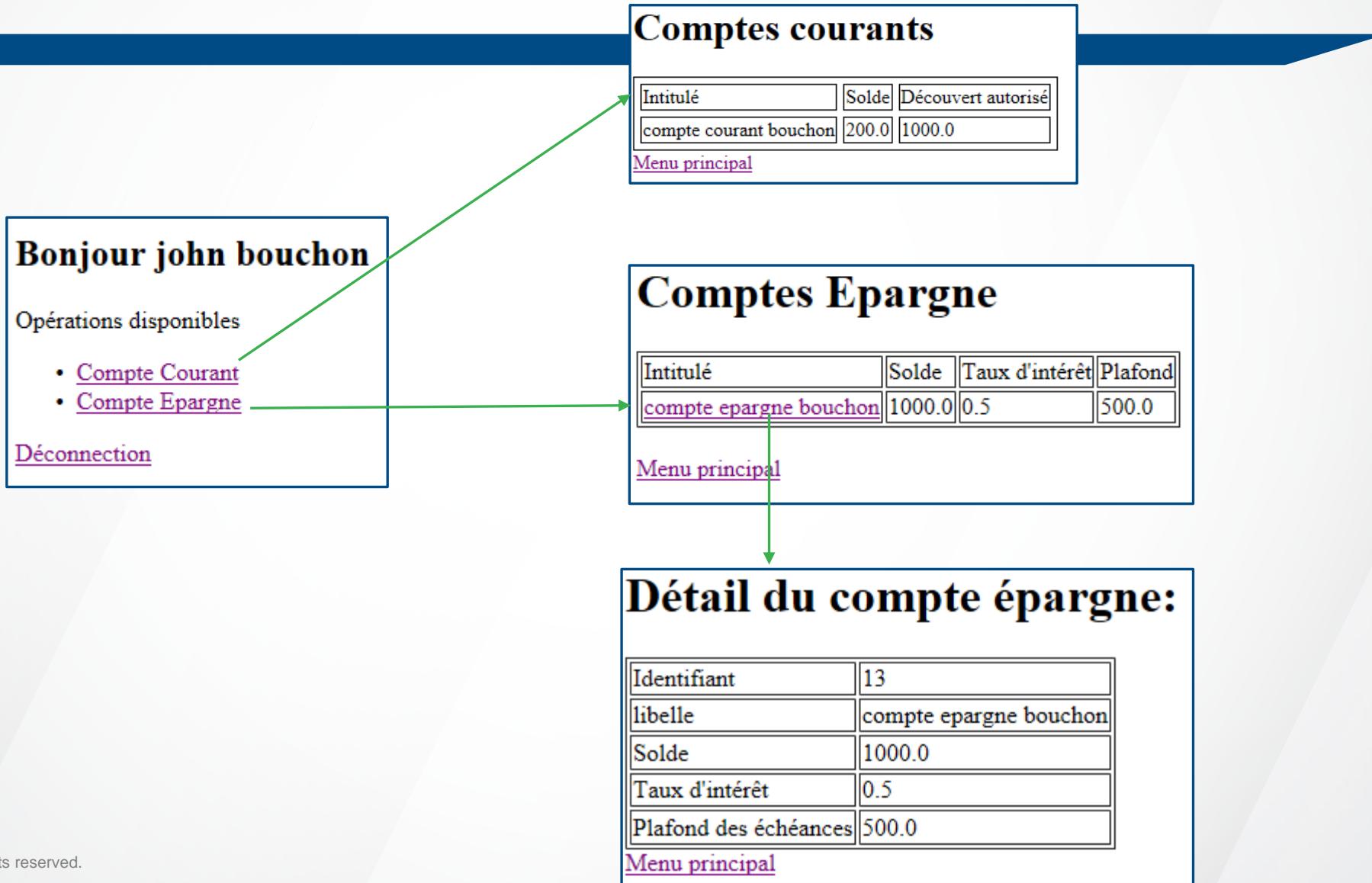
## ➤ Modifier la servlet : TraiterLoginServlet.java

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse
 * response)
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 String identifiant = (String) request.getParameter("login");
 String motDePasse = (String) request.getParameter("motDePasse");

 BanqueServiceBouchon bs = new BanqueServiceBouchon();

 try {
 Client c = bs.findClient(identifiant, motDePasse);
 HttpSession session = request.getSession(true);
 session.setAttribute("client", c);
 response.sendRedirect(request.getContextPath() + "/pagePrincipale.jsp");
 } catch (Exception e) {
 response.sendRedirect(request.getContextPath() + "/erreur.jsp");
 System.out.println("connection impossible");
 }
}
```

# Tester l'application



# LABS

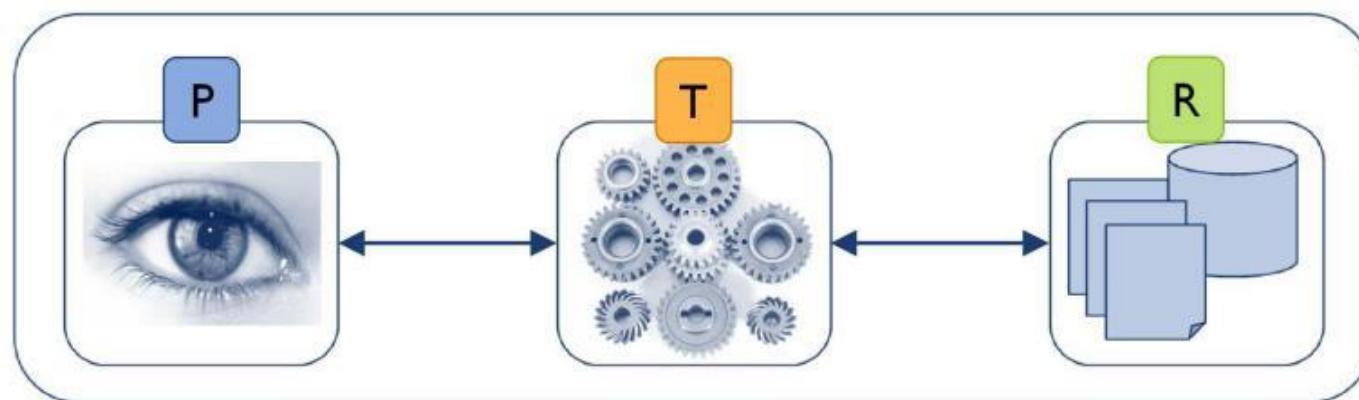
- MISE EN OEUVRE DE SERVLETS,  
JSP ET BEANS : BIBLIOTHEQUE  
EN LIGNE



# Sommaire

- Chapitre 1 : INTRODUCTION
  - Chapitre 2 : PLATE-FORME JAVA EE
  - Chapitre 3 : SERVLET / JSP
  - **Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE**
  - Chapitre 5 : JSF
  - Chapitre 6 : PRIMEFACES
- 
- 

# Principe de conception d'une application



## ➤ 3 types de responsabilités = 3 couches (tiers) principales

Présentation	Interaction avec l'utilisateur
Traitement	Traitements métiers, logique applicative
Ressources	Gestion des ressources, des données

# Concept de persistance

- Dans une architecture n tiers, nous dédions une couche d'accès aux données ou couche de persistance, elle permet notamment :
  - d'ajouter un niveau d'abstraction entre la base de données et l'utilisation qui en est faite.
  - de simplifier la couche métier qui utilise les traitements de cette couche
  - de masquer les traitements réalisés pour la correspondance objets/base de données
  - de faciliter le remplacement de la base de données utilisée
- Il y a plusieurs façons d'assurer la persistance des objets, qui, par sophistication croissante, sont :
  - la simple écriture et lecture des valeurs d'attributs à sauvegarder sur un fichier,
  - la sérialisation des objets,
  - l'interaction avec une base de données

**Persistance d'objets = sauvegarder l'état des objets (valeur des attributs) entre deux exécutions**

# Sérialisation

- Pour être sérialisée, la classe d'un objet doit implémenter l'interface Serializable.

```
public class Formation implements Serializable {
 private String nom;
 private List<Etudiant> etudiants;
 // ... méthodes
}
```

```
public class Etudiant implements Serializable {
 private String nom;
 private String prenom;
 private int age;
 // ... méthodes
}
```

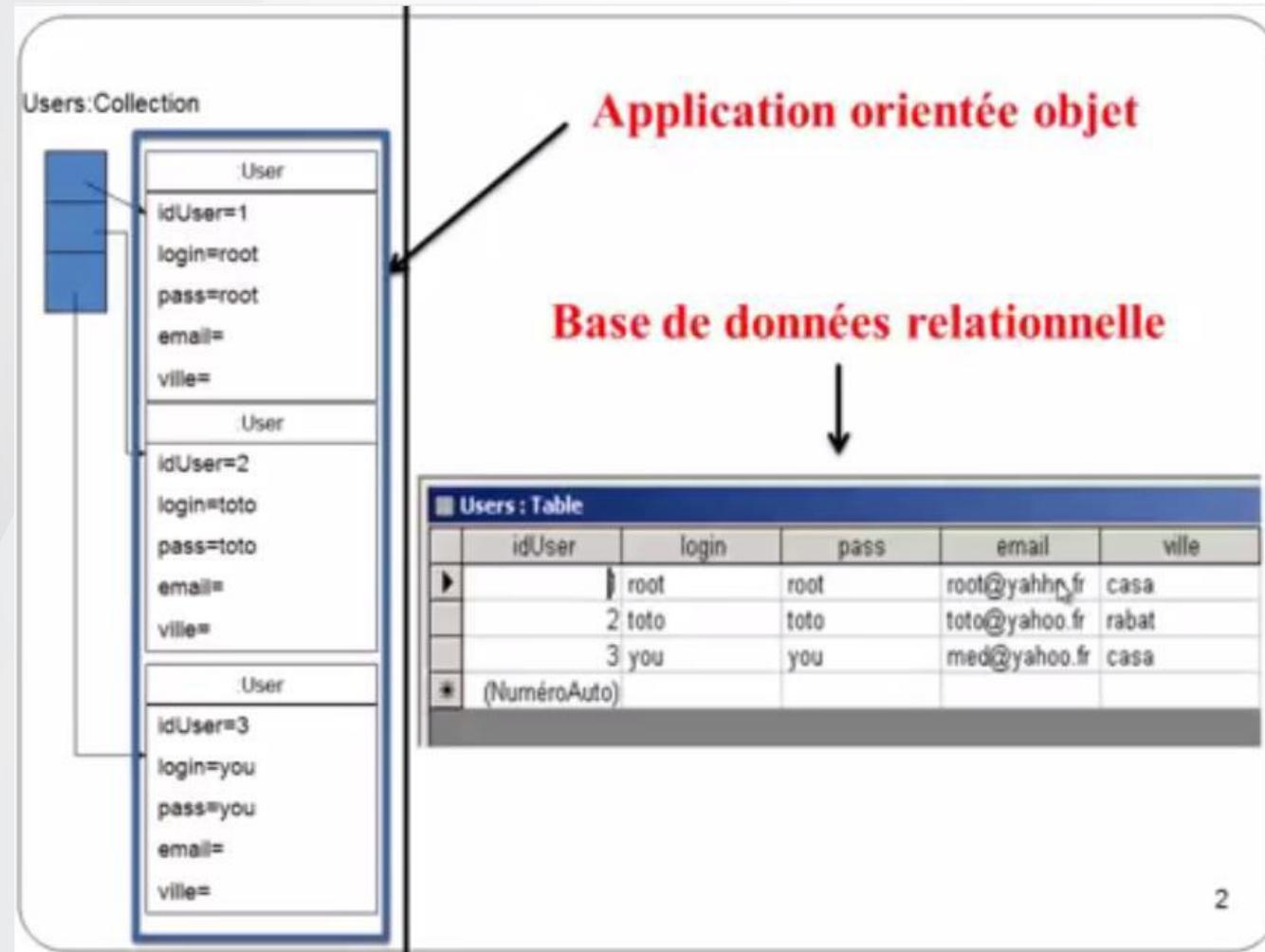
- Sérialisation

```
public static void main(String[] args) throws Exception {
 Scanner clavier = new Scanner(System.in);
 FileOutputStream fos = new FileOutputStream("formation.txt");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 Formation genieLogiciel4 = new Formation("master 1 génie logiciel");
 // ... renseigner les étudiants qui compose la formation
 System.out.println("Sérialisation de l'objet : " + genieLogiciel4);
 oos.writeObject(genieLogiciel4);
 oos.close();
}
```

# Désérialisation

```
public static void main(String[] args) throws Exception {
 FileInputStream fis = new FileInputStream("formation.txt");
 ObjectInputStream ois = new ObjectInputStream(fis);
 Formation formation = (Formation)ois.readObject();
 System.out.println("Déserialiser : " + formation);
 ois.close();
}
```

# Modèle relationnel versus modèle objet



- **Objet** : propose plus de fonctionnalités ; héritage, polymorphisme, ...
- Les **relations** entre les entités des deux modèles sont différentes (clé étrangère, associations/héritage).
- **Relationnel**, chaque occurrence possède un identifiant unique, contrairement aux objets (hormis son adresse mémoire qui varie d'une exécution à l'autre).

# Exemple :

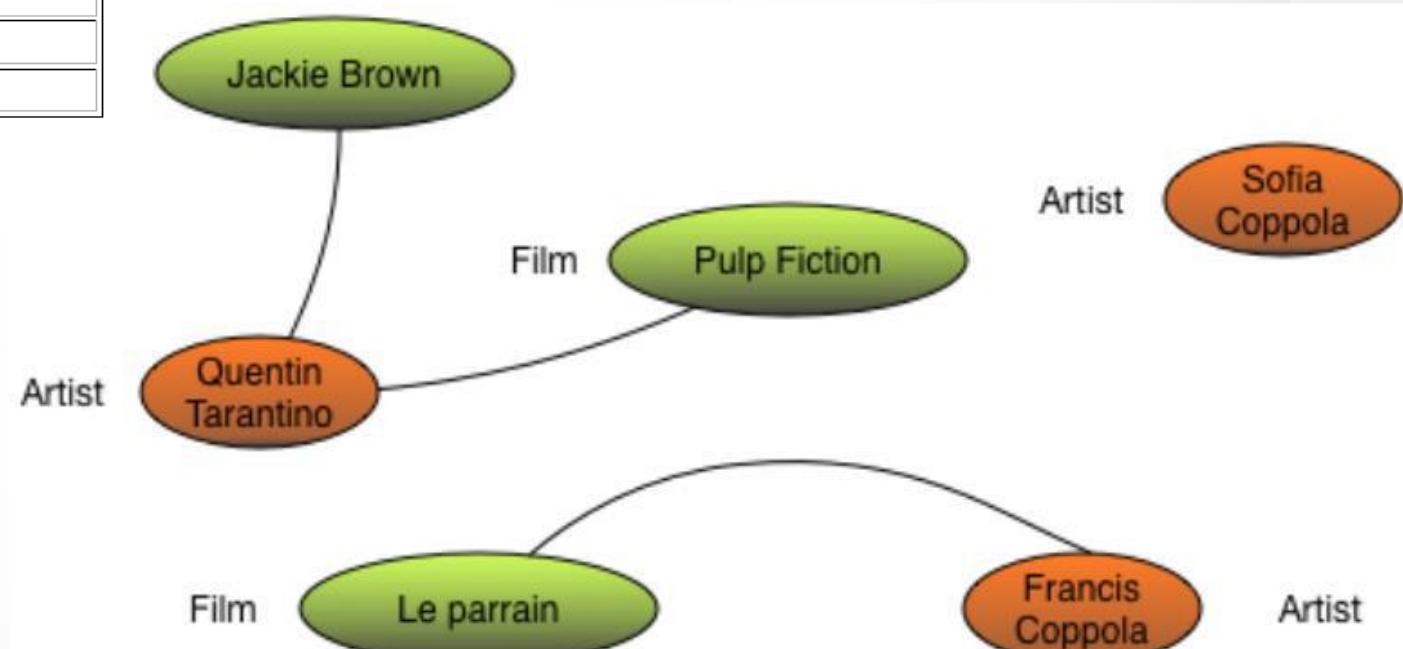
Table des films

<b>id</b>	<b>titre</b>	<b>année</b>	<b>idRéalisateur</b>
17	Pulp Fiction	1994	37
54	Le Parrain	1972	64
57	Jackie Brown	1997	37

Table des artistes

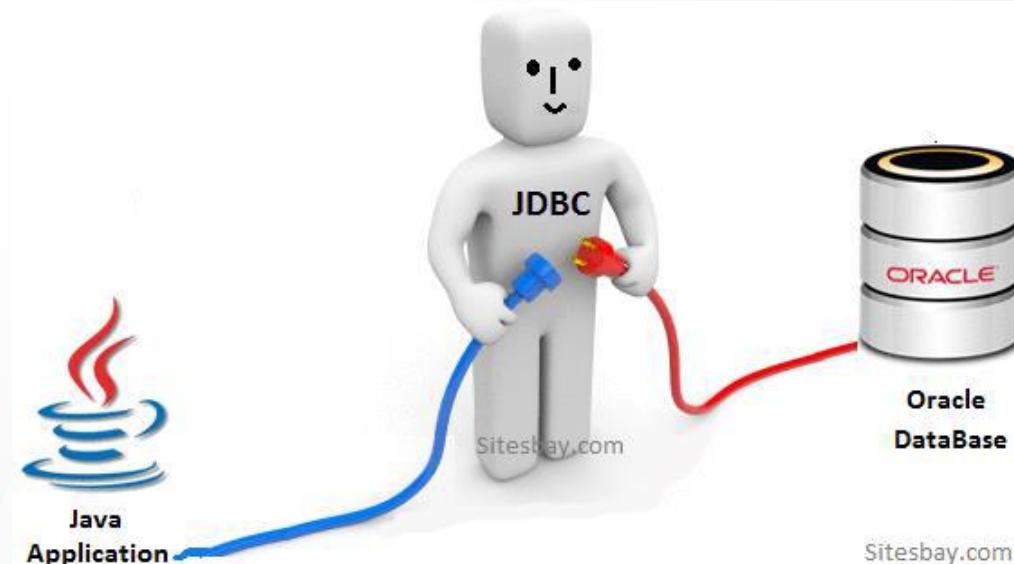
<b>id</b>	<b>nom</b>	<b>prénom</b>	<b>année_naissance</b>
1	Coppola	Sofia	1971
37	Tarantino	Quentin	1963
64	Coppola	Francis	1939

**Remarque** - Il est toujours possible par une requête SQL (jointure) de trouver les films réalisés par un artiste, ou le réalisateur d'un film. En java, le lien peut être unidirectionnel.



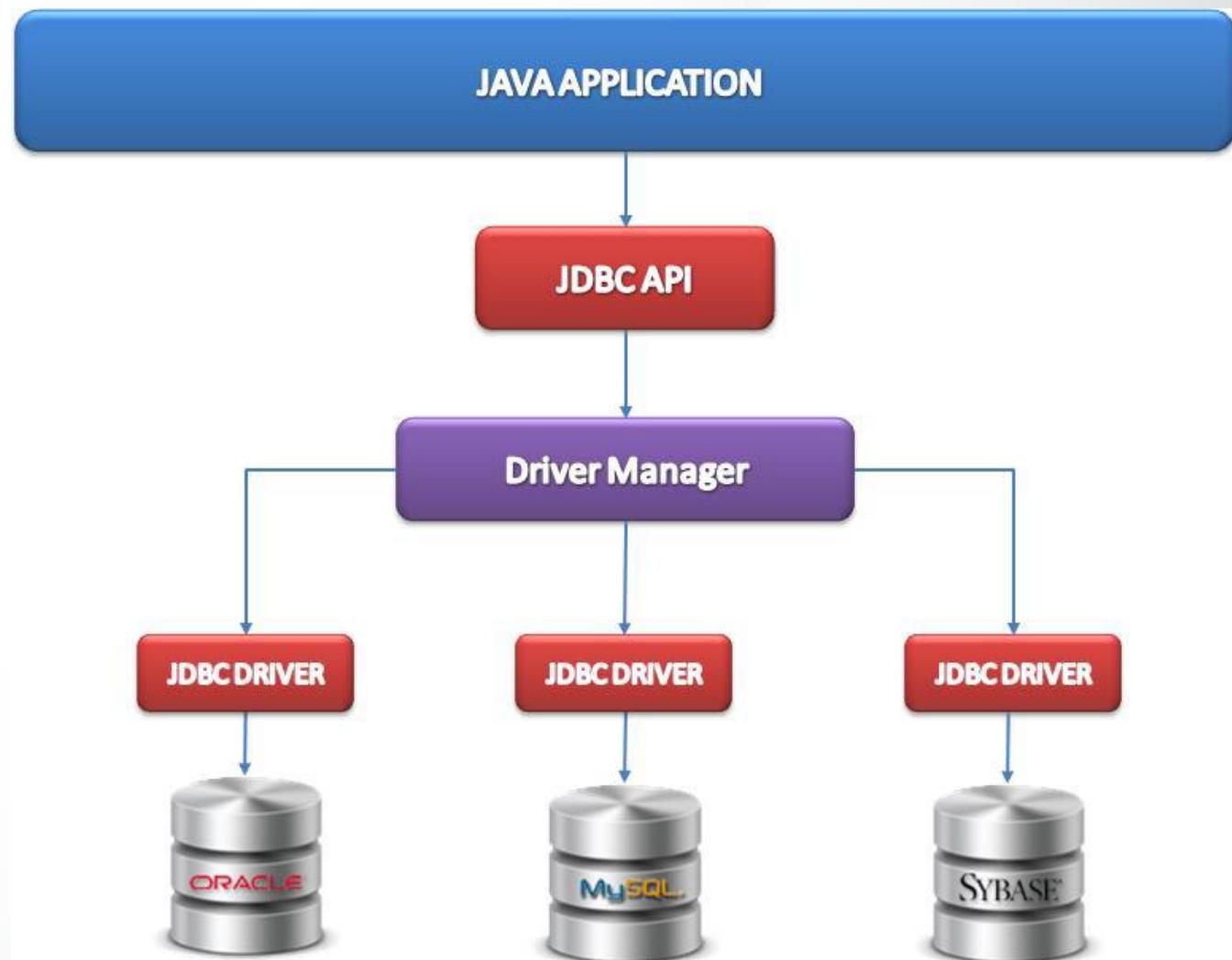
# Java Database Connectivity (JDBC)

- JDBC (Java DataBase Connectivity) est une API (Application Programming Interface) qui permet d'exécuter des instructions SQL. JDBC fait partie du JDK (Java Development Kit), paquetage `java.sql`.
- Permet à un programme Java d'interagir
  - localement ou à distance
  - avec une base de données relationnelle
- Fonctionne selon un principe client/serveur
  - client = le programme Java
  - serveur = la base de données



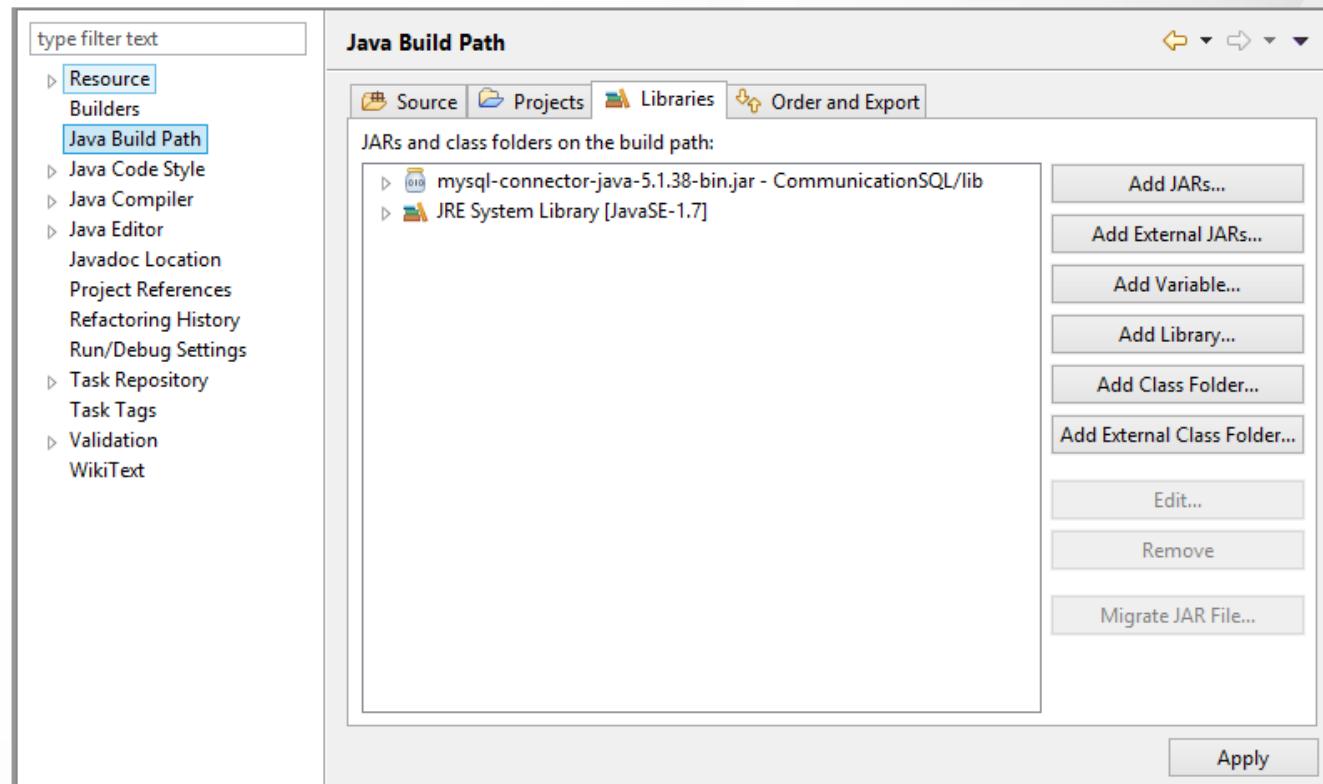
# Architecture JDBC

- Gestionnaire de drivers (driver manager) = gestionnaire de tous les drivers chargés par un programme Java
- Driver JDBC : gère les détails des communications avec un type de SGBD
  - un type de driver par BD (Oracle, MySQL, Postgres, ...)



# JDBC et projet

- Récupérer l'archive correspondant au pilote de la base de données que vous voulez utiliser.
- Par exemple le fichier **mysql-connector-java-xxx-bin.jar**
- Il suffit ensuite de copier ce fichier dans le répertoire **/lib du projet (créer /lib si besoin)**
- **Paramétrer le chemin de construction (build Path) la nouvelle librairie.**



# Chargement du pilote de la base

- Le pilote d'une base de données est un objet type interface `java.sql.Driver`.
- Chaque éditeur de base de données fournit une telle classe (généralement un JAR)
- Le nom complet de cette classe varie donc d'un éditeur à l'autre.
- Dans le cas de MySQL, il s'agit de `com.mysql.jdbc.Driver`.
- Charger une classe Java explicitement dans le code : appel de la méthode `Class.forName()`.
  - 1 chargement du pilote MySQL, récupération d'un objet de type Class :

```
Class.forName("com.mysql.jdbc.Driver");
```

# ouverture de la connexion

```
Connection connect = DriverManager.getConnection(chaineDeConnection, user, passwd);
```

- L'objet chaîneDeConnection est de la classe String. C'est une concaténation des éléments suivants :
  - jdbc:mysql : protocole utilisé pour accéder à la base de données. Ce protocole est propre à chaque base.
  - localhost:3306 : nom d'hôte qui héberge la base de données, ainsi que le port d'accès.
  - nom\_de\_la\_base : nom de la base de données à laquelle on souhaite se connecter.

```
String url = "jdbc:mysql://localhost/inscription";
String user = "root";
String passwd = "";
Connection connect;
Class.forName("com.mysql.jdbc.Driver");
connect = DriverManager.getConnection(url,user, passwd);
```

# Utilisation de JDBC

- Création d'une requête SQL (Statement, PreparedStatement ou CallableStatement)
  - Statement st = cx.createStatement();
- Exécuter la requête SQL (interroger la base (executeQuery()), ou modifier la base (executeUpdate()))
  - Envoi d'une requête SELECT
    - ResultSet rs = st.executeQuery( "SELECT \* FROM etudiant" );
  - Envoi d'une requête CREATE, INSERT ou UPDATE
    - int rs = st.executeUpdate( "INSERT INTO etudiant VALUES ('Coulibaly','Ibrahim',...)" );

# Récupération du résultat

- `rs.next()` retourne vrai tant qu'il reste des enregistrements dans le résultat et positionne le curseur sur l'enregistrement suivant
- `rs.getString(String attribut)` (ex. : `rs.getString("nom")` )
  - retourne la valeur de l'attribut attribut de type String dans le résultat
- `rs.getInt(String attribut)`, `getBoolean`, `getByte`, `getDouble`, `getFloat`
  - idem pour des attributs de type int, boolean, byte, double ou float

# Exemple

```
public static void main(String[] args) {
 Connection cx = null;
 Statement st = null;
 ResultSet rs = null;
 try {
 Class.forName("com.mysql.jdbc.Driver");
 cx = DriverManager.getConnection("jdbc:mysql://localhost/inscription", "root", "");
 st = cx.createStatement();
 rs = st.executeQuery("SELECT * FROM etudiant");
 while (rs.next()) {
 String nom = rs.getString("nom");
 String prenom = rs.getString("prenom");
 System.out.println(nom + " " + prenom);
 }
 rs.close(); st.close(); cx.close();
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

```
public class Etudiant {
 private Long id;
 private String nom;
 private String prenom ;

 public Etudiant() {}
 //...
}
```

# Correspondances types données SQL/Java

- Tous les SGBD n'ont pas les mêmes types SQL

Type SQL	Type Java	Méthode getter
CHAR VARCHAR	String	getString()
INTEGER	int	getInt()
TINYINT	byte	getByte()
SMALLINT	short	getShort()
BIGINT	long	getLong()
BIT	Boolean	getBoolean()
REAL	float	getFloat()
FLOAT DOUBLE	double	getDouble()
NUMERIC DECIMAL	java.lang.BigDecimal	getBigDecimal()
DATE	java.sql.Date	getDate()
TIME	Java.sql.Time	getTime()
TIMESTAMP	Java.sql.Timestamp	getTimeStamp()

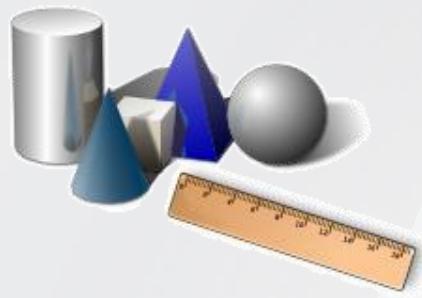
# Exemples

```
public ArrayList<Etudiant> findAll() {
 ArrayList<Etudiant> etudiants = new ArrayList<Etudiant>();
 Etudiant etudiant;
 ResultSet result;
 try {
 result = this.connect.createStatement().executeQuery("SELECT * FROM etudiant");
 while (result.next()) {
 etudiant = new Etudiant();
 etudiant.setId(result.getLong("id"));
 etudiant.setNom(result.getString("nom"));
 etudiant.setPrenom(result.getString("prenom"));
 etudiants.add(etudiant);
 }
 } catch (SQLException e) {
 e.printStackTrace();
 }
 return etudiants;
}
```

```
public boolean create(Etudiant obj) {
 try {
 PreparedStatement prepare = this.connect.prepareStatement(
 "INSERT INTO etudiant (id_classe,nom,prenom) VALUES (?, ?, ?)");
 prepare.setLong(1, obj.getIdClasse());
 prepare.setString(2, obj.getNom());
 prepare.setString(3, obj.getPrenom());
 prepare.executeUpdate();
 } catch (SQLException e) {
 e.printStackTrace();
 }
 return true;
}
```

# Fermer les ressources

- Toutes les ressources JDBC doivent être fermées dès qu'elles ne sont plus utilisées
- Le plus souvent la fermeture doit se faire dans un bloc finally pour qu'elle ait lieu quel que soit le déroulement des opérations (avec ou sans erreurs)
- Les ressources sont automatiquement fermées par le ramasse-miettes mais il faut les fermer explicitement (on ne sait quand/si il va être lancé)
- Les ressources à fermer
- Connection : leur fermeture est indispensable car c'est la ressource la plus coûteuse ; si on utilise un pool de connexions, la fermeture rend la connexion au pool
- Statement, et les sous-interfaces PreparedStatement et CallableStatement
- ResultSet : il est automatiquement fermé lorsque le statement qui l'a engendré est fermé ou réexécuté



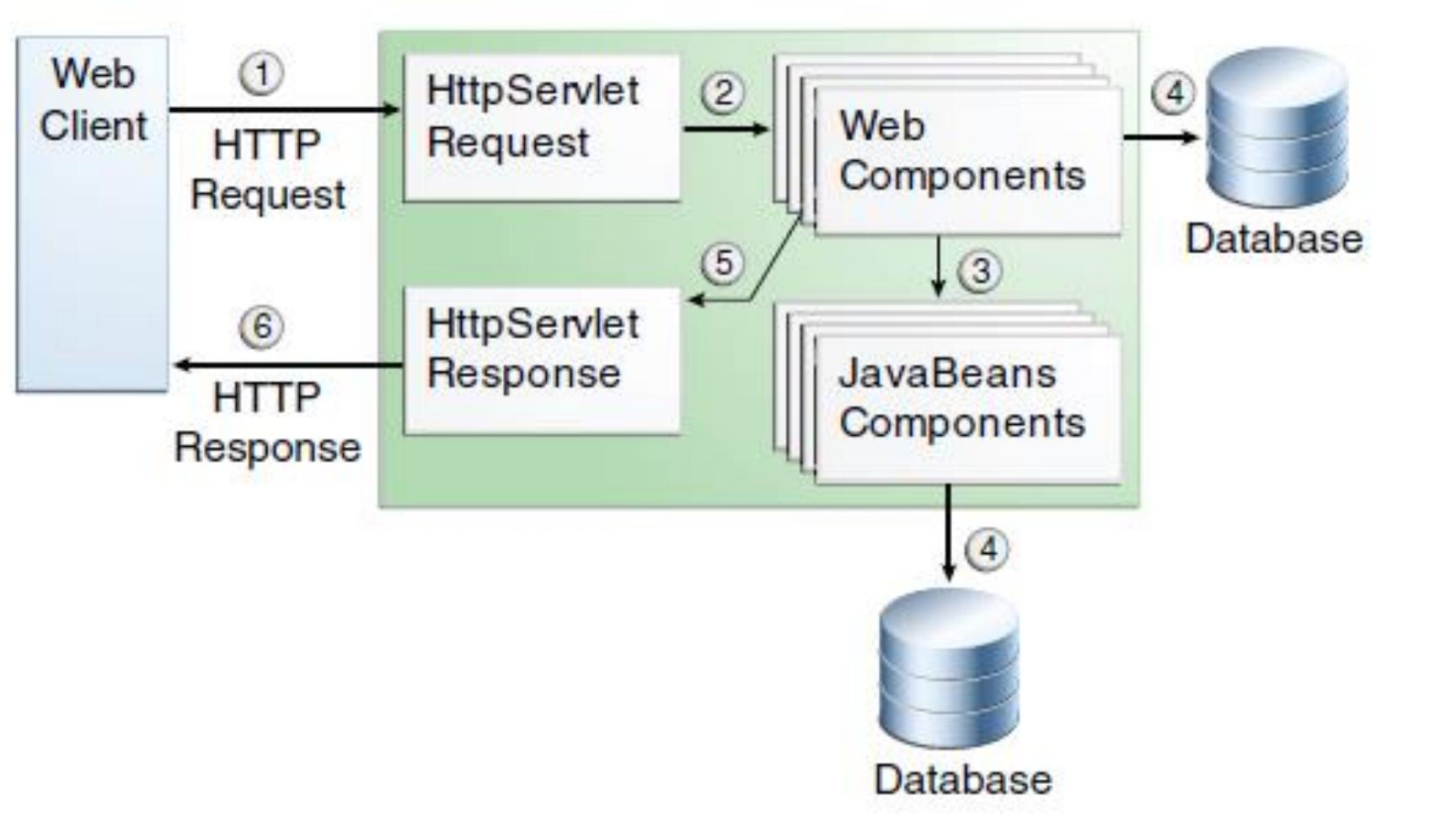
## Workshop 6 :

- Reprenez le workshop 5 en ajoutant la prise en charges de la base de données
  - Ajoutez le driver mysql
  - Ajoutez les tables pour accueillir a compte épargne et un compte courant
  - En java mettez en place la méthode qui recherche les comptes a partir de leur id

# Sommaire

- Chapitre 1 : INTRODUCTION
  - Chapitre 2 : PLATE-FORME JAVA EE
  - Chapitre 3 : SERVLET / JSP
  - Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE
  - **Chapitre 5 : JSF**
  - Chapitre 6 : PRIMEFACES
- 
- 

# Architecture typique d'une application web



# Utilité de JSF

- Créer des pages Web dynamiques
- Par exemple, une page Web qui est construite à partir de données enregistrées dans une base de données
- Une grande partie de la programmation liée à la validation des données saisies par l'utilisateur et de leur passage au code de l'application est automatisée ou grandement facilitée
- Ajax sans programmation

# JSF

- Framework MVC qui reprend le modèle des interfaces utilisateurs locales (comme Swing),
- Le modèle est représenté par des classes Java, les backing beans, sur le serveur,
- Les vues sont représentées par des composants Java sur le serveur, transformées en pages HTML sur le client,
- Le contrôleur est une servlet qui intercepte les requêtes HTTP liées aux applications JSF et qui organise les traitements JSF

# Services rendus par JSF (1/2)

- **Architecture MVC** pour séparer l'interface utilisateur, la couche de persistance et les processus métier, utilisant la notion d'événement,
- Conversion des données (tout est texte dans l'interface utilisateur),
- Validation des données (par exemple, des champs de formulaires),
- Automatisation de l'affichage des messages d'erreur en cas de problèmes de conversion ou de validation

# Services rendus par JSF (2/2)

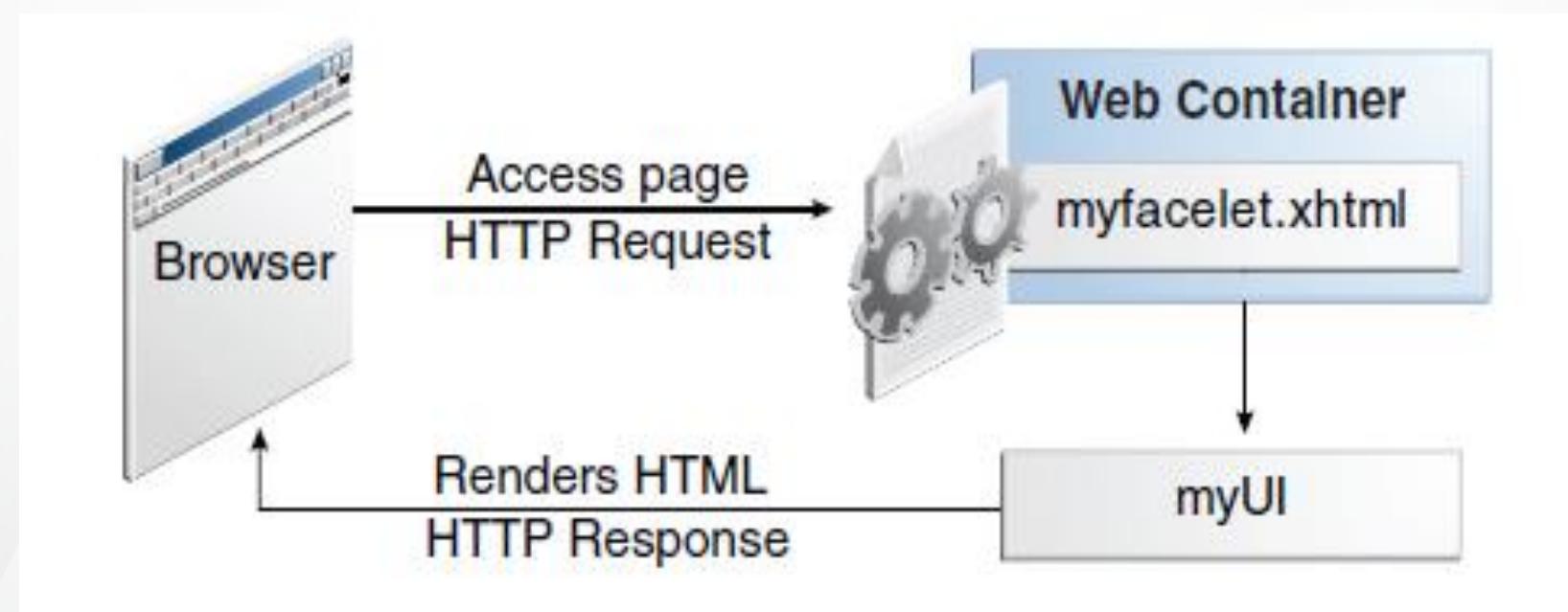
- Internationalisation,
- Support d'Ajax sans programmation javascript (communication en arrière-plan et mise à jour partielle de l'interface utilisateur),
- Fournit des composants standards simples pour l'interface utilisateur,
- Possible d'ajouter ses propres composants,
- Adaptable à d'autres langages de balise que HTML (WML par exemple pour les téléphones portables).

# Standards

- JSF 2 est intégré dans Java EE 6,
- JSF 2 s'appuie sur les standards suivants :
  - Servlet 2.5 et +
  - Java SE 5.0 et +
  - Java EE 5.0 et +
  - JSTL 1.1 <- A EVITER !
    - ➔ JSF 2 peut (c'est conseillé) utiliser CDI (*Contexts and Dependency Injection*)
- La version 2.1 date de novembre 2010. ( correction de bug et des améliorations)
- La version 2.2 date de avril 2013 (support HTML5, les vues sans état ...)
- La version 2.3 (la plus récente au moment de l'écriture de cette page) déprécie certaines annotations JSF au profit d'annotations CDI. CDI + optimisation

# Page JSF

- Les pages JSF contiennent des balises qui décrivent les composants qui représenteront la page sur le serveur



# Une page JSF

- Code XHTML,
- Contient des parties en EL : #{...},
- Utilise souvent une (ou plusieurs) bibliothèque de composants,
- Sera traduite en XHTML « pur » pour être envoyée au client Web,

# JSF, servlets et beans

- Dans l'architecture des applications qui utilisent JSF, on trouve aussi des servlets et des beans,
- Les servlets et les beans s'occupent principalement des traitements métier et des accès aux données,
- Les pages JSF sont utilisées pour l'interface avec l'utilisateur.

# Répartition des tâches

- Les pages JSF ne contiennent pas de traitements (pas de code Java ou autre code comme dans les pages JSP, « ancien langage » de JSF pour créer les pages affichées à l'utilisateur,
- Les traitements liés directement à l'interface utilisateur sont écrits dans les backing beans,
- Ces backing beans font appels à des EJB ou des classes Java ordinaire pour effectuer les traitements qui ne sont pas liés directement à l'interface utilisateur

# Répartition des tâches (2)

- Les EJB sont chargés de traitements métier et des accès aux bases de données,
- Les accès aux bases de données utilisent JPA et donc les entités.

# Beans

- 2 types principaux de beans :
  - EJB (essentiellement des beans sessions) liés aux processus métier,
  - « Backing beans » associés aux pages JSF et donc à l'interface utilisateur qui font la liaison entre les composants JSF de l'interface, les valeurs affichées ou saisies dans les pages JSF et le code Java qui effectue les traitements métier.

# Backing bean

- Souvent, mais pas obligatoirement, un backing bean par page JSF,
- Conserve, par exemple, les valeurs saisies par l'utilisateur (attribut value), ou une expression qui indique si un composant ou un groupe de composant doit être affiché ; peut aussi conserver un lien vers un composant de l'interface utilisateur (attribut binding), ce qui permet de manipuler ce composant ou les attributs de ce composant par programmation.

# Backing Bean

## ➤ Version simple : les modèles

- Les modèles utilisés dans les vues: définis par les getters, ex: `getAllComptes()...` définit une propriété `allComptes` et sera accessible en lecture dans une page JSF par `#{nomBean.allComptes}`
- Si il y a un setter, accessible aussi en modification (ex: un formulaire)
- Les setters sont appelés dès la soumission du formulaire, avant les méthodes de traitement que vous aurez écrites
- Les vues peuvent consulter les modèles plusieurs fois pour s'afficher -> ne pas faire de choses complexes dans les getters !

# Backing Bean

- Version simple: les méthodes de contrôle:
  - Ce sont les autres méthodes dans le bean,
  - Appelées par des actionLink ou actionBarButton via l'attribut **action=#{nomBean.save()}** par ex.
  - Si ces méthodes renvoient null ou void -> on re-affiche la même page,
  - Sinon si elles renvoient une String -> renvoient une autre page dont le nom est la valeur renournée

# Backing bean

- Version simple: on délègue le travail aux EJB
  - Les méthodes de contrôle délèguent le travail aux EJB.

# EJB et JSF

- Un backing bean fournit du code Java pour l'interface graphique,
- Lorsqu'un processus métier doit être déclenché, le backing bean cède la main à un EJB ou à un service JAVA,
- Un EJB/Service Java est totalement indépendant de l'interface graphique ; il exécute les processus métier ou s'occupe de la persistance des données,
- Les entités JPA 2 sont utilisées par les EJB/Service pour la persistance des données.

# Container pour les JSF

- Pour qu'il sache traiter les pages JSF, le serveur Web doit disposer d'un container pour ces pages,
- On peut utiliser pour cela un serveur d'applications du type de Glassfish ou Tomcat.

# Composants JSF sur le serveur

- JSF utilise des composants côté serveur pour construire la page Web,
- Par exemple, un composant java UIInputText du serveur sera représenté par une balise <INPUT> dans la page XHTML,
- Une page Web sera représentée par une vue, UIViewRoot, hiérarchie de composants JSF qui reflète la hiérarchie de balises HTML.

# Cycle de vie JSF 2

- Pour bien comprendre JSF il est indispensable de bien comprendre tout le processus qui se déroule entre le remplissage d'un formulaire par l'utilisateur et la réponse du serveur sous la forme de l'affichage d'une nouvelle page.

# Le servlet « Faces »

- Toutes les requêtes vers des pages « JSF » sont interceptées par un servlet défini dans le fichier `web.xml` de l'application Web
- ```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

URL des pages JSF

- Les pages JSF sont traitées par le servlet parce que le fichier web.xml contient une configuration telle que celle-ci :

```
<servlet-mapping>
    < servlet-name>Faces Servlet</servlet-name>
    < url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

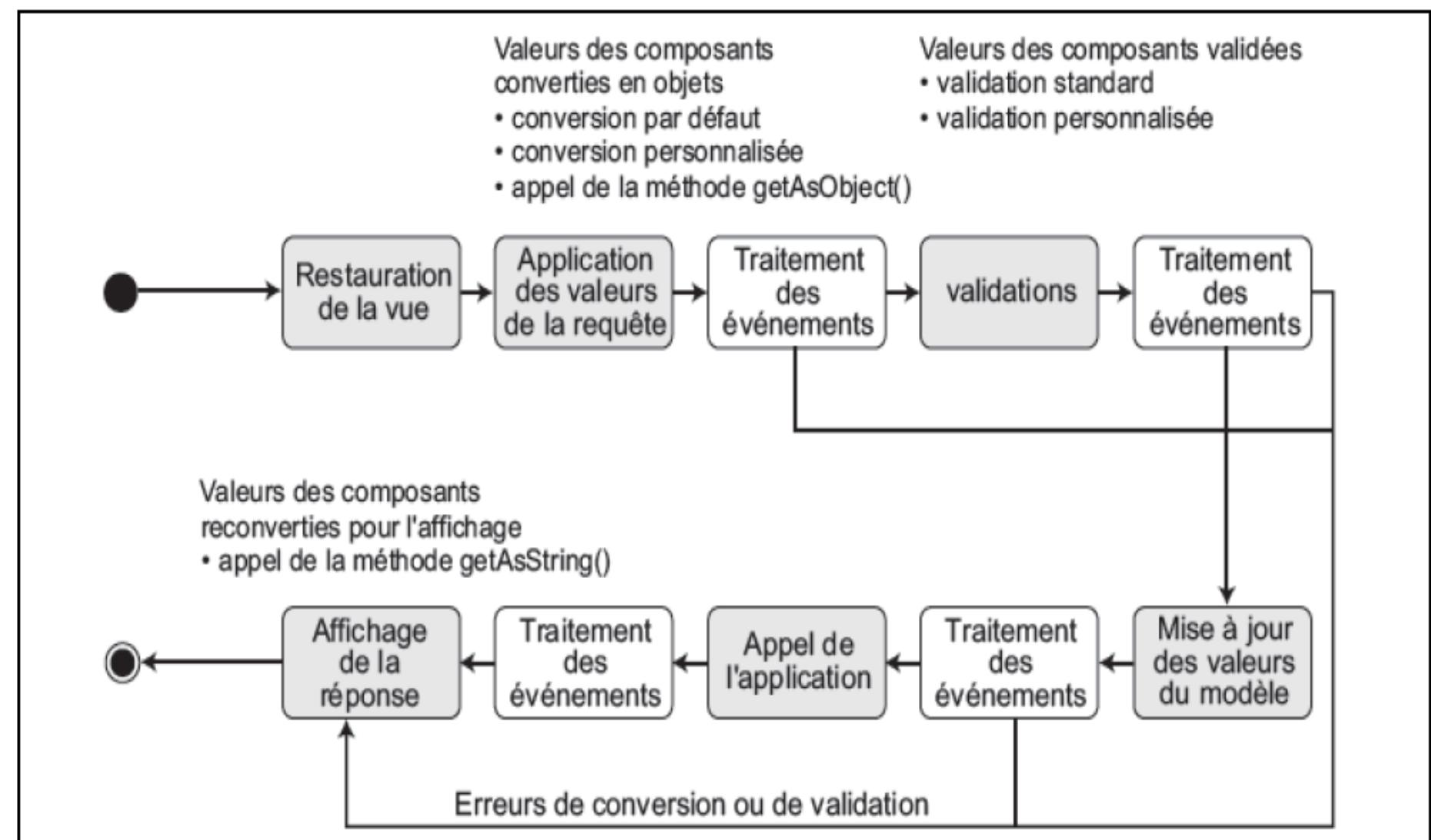
Le pattern est souvent aussi de la forme
*** .faces** (les pages dont l'URL se termine par .faces sont considérées comme des pages JSF) ou ***.jsf**

Codage - décodage

- Les pages HTML renvoyées par une application JSF sont représentées par un arbre de composants Java,
- L'encodage est la génération d'une page HTML à partir de l'arbre des composants,
- Le décodage est l'utilisation des valeurs renvoyées par un POST HTML pour donner des valeurs aux variables d'instance des composants Java, et le lancement des actions associées aux « UICommand » JSF (boutons ou liens).

Cycle de vie

- Le codage/décodage fait partie du cycle de vie des pages JSF,
- Le cycle de vie est composé de 6 phases,
- Ces phases sont gérées par la servlet « Faces » qui est activé lorsque qu'une requête demande une page JSF (correspond au pattern indiqué dans le fichier web.xml ; le plus souvent /faces/* ou *.faces)



Demande page HTML

- Etudions tout d'abord le cas simple d'une requête d'un client qui demande l'affichage d'une page JSF.

La vue

- Cette requête HTTP correspond à une page JSF (par exemple *.faces) et est donc interceptée par le servlet Faces,
- La page HTML correspondant à la page JSF doit être affichée à la suite de cette requête HTTP,
- La page HTML qui sera affichée est représentée sur le serveur par une « vue »,
- Cette vue va être construite sur le serveur et transformée sur le serveur en une page HTML qui sera envoyée au client.

Contenu de la vue

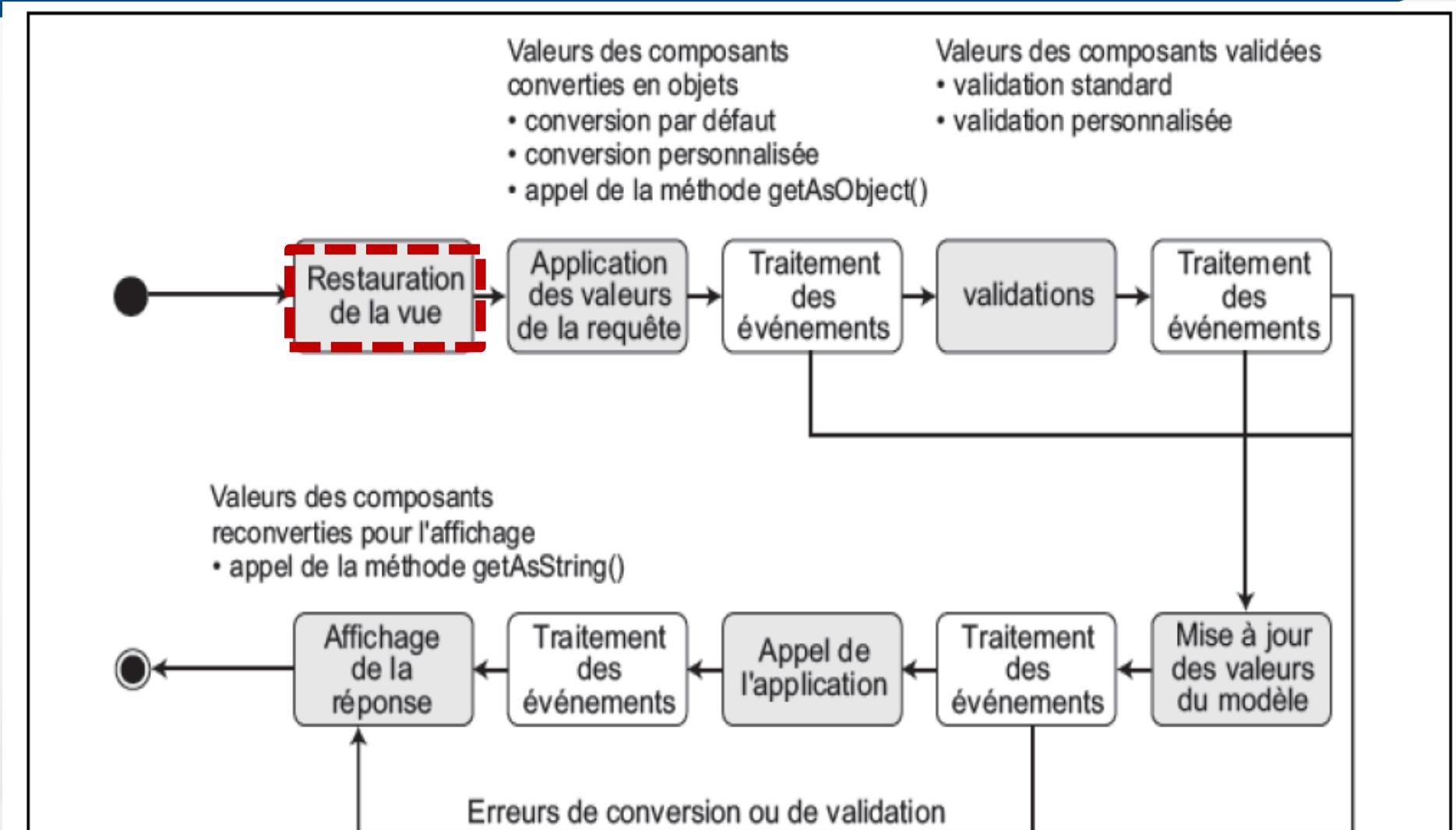
- Cette vue est **un arbre dont les éléments sont des composants JSF** (composants Java) qui sont sur le serveur (des instances de classes qui héritent de UIComponent),
- Sa racine est de la classe UIViewRoot.

Construction vue et page HTML

- Une vue formée des composants JSF est donc construite sur le serveur (ou restaurée si elle avait déjà été affichée) : phase « **Restore View** »,
- Puisqu'il n'y a pas de données ou d'événements à traiter, la vue est immédiatement rendue : le code HTML est construit à partir des composants de la vue et envoyé au client : phase « **Render Response** ».

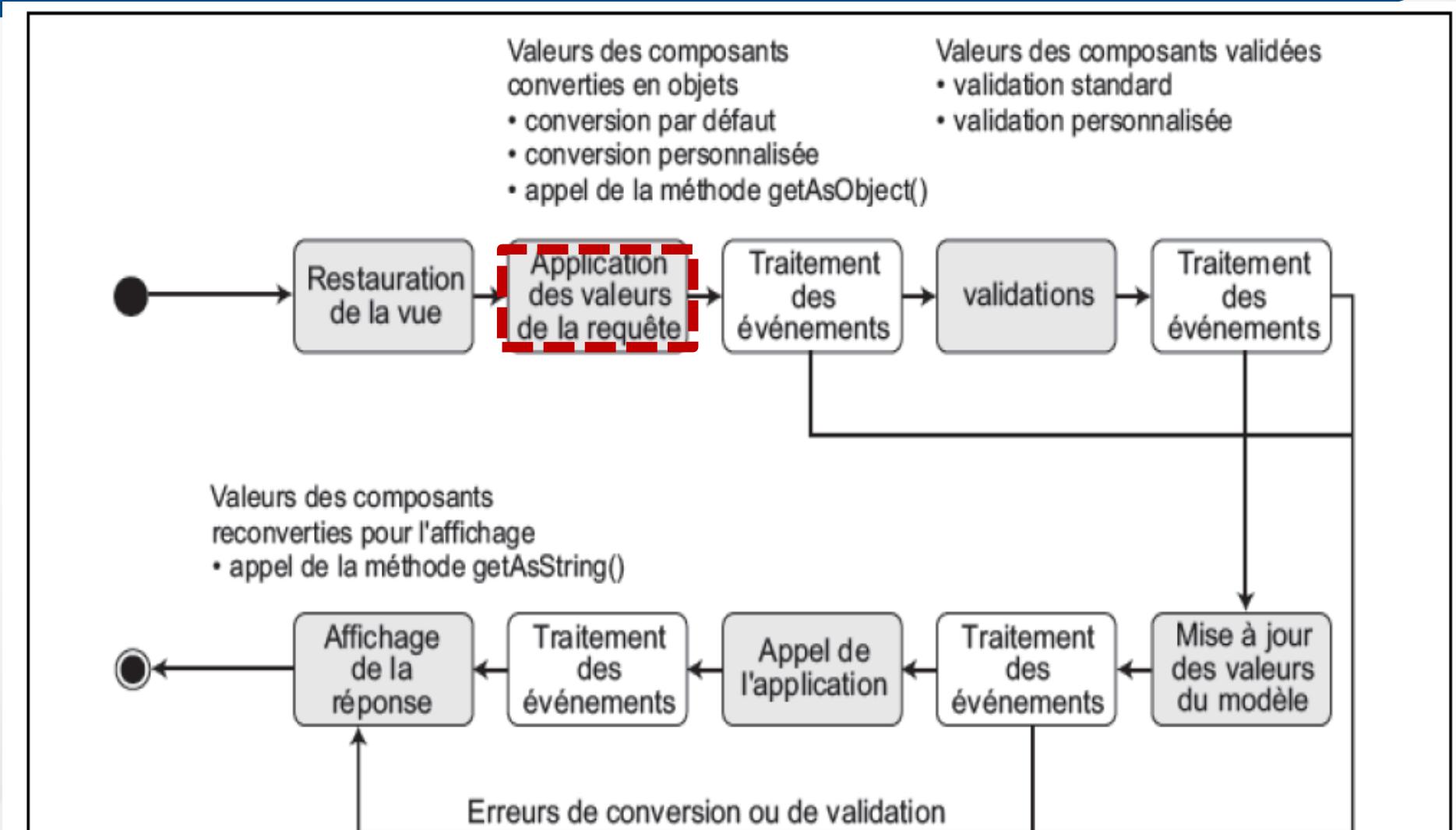
Traitement d'un formulaire

- Nous allons cette fois-ci partir d'une requête HTTP générée à partir d'une page HTML qui contient un formulaire (page construite à partir d'une page JSF),
- L'utilisateur a saisi des valeurs dans ce formulaire,
- Ces valeurs sont passées comme des paramètres de la requête HTTP ; par exemple, `http://machine/page.xhtml?nom=bibi&prenom=bob` si la requête est une requête GET, ou dans l'entité d'un POST.



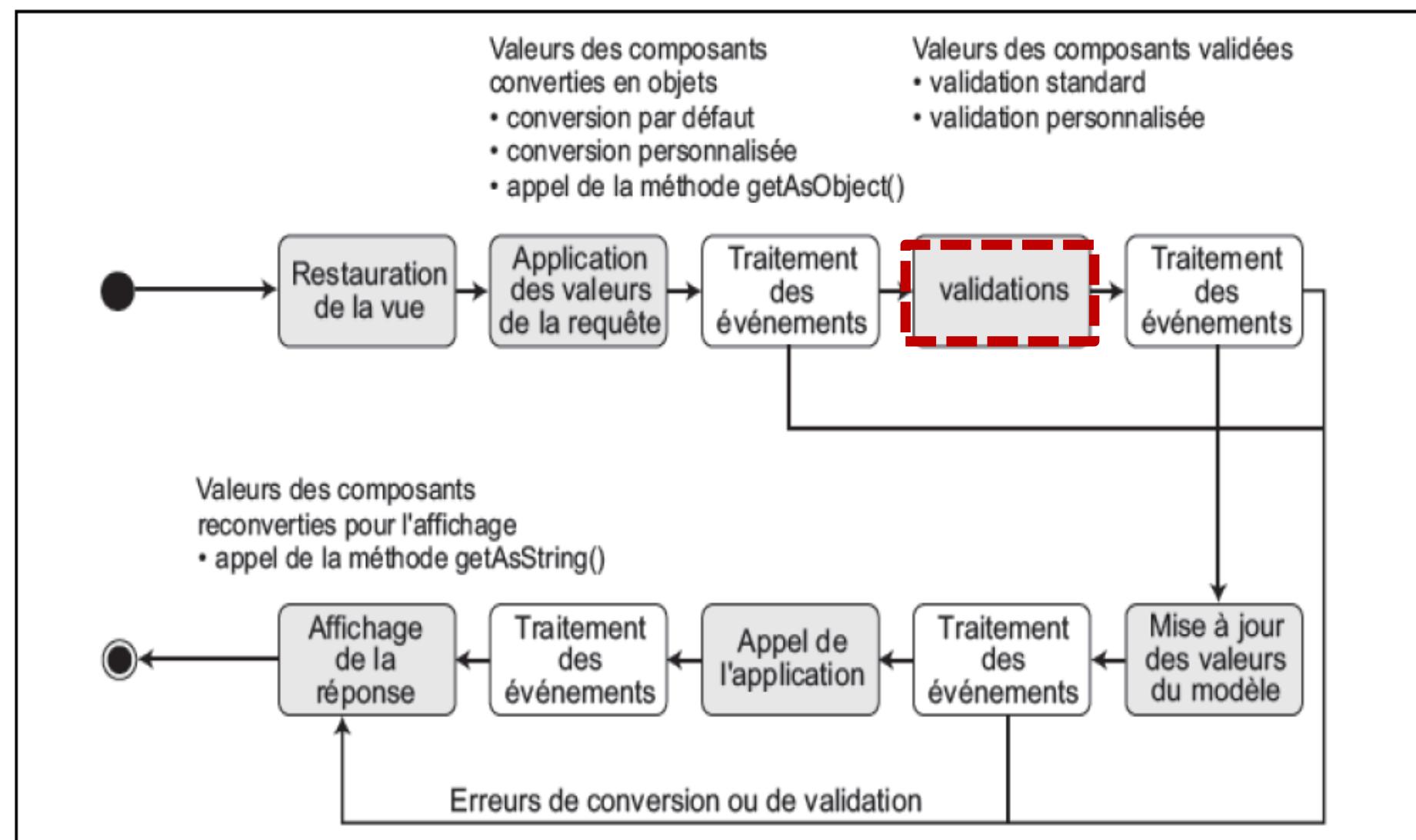
Phase de restauration de la vue

- La vue qui correspond à la page qui contient le formulaire est restaurée (phase « Restore View »),
- Tous les composants reçoivent la valeur qu'ils avaient avant les nouvelles saisies de l'utilisateur.



Phase d'application des paramètres

- Tous les composants Java de l'arbre des composants reçoivent les valeurs qui les concernent dans les paramètres de la requête HTTP : phase « Apply Request Values »,
- Par exemple, si le composant texte d'un formulaire contient un nom, le composant Java associé conserve ce nom dans une variable,
- En fait, chaque composant de la vue récupère ses propres paramètres dans la requête HTTP.

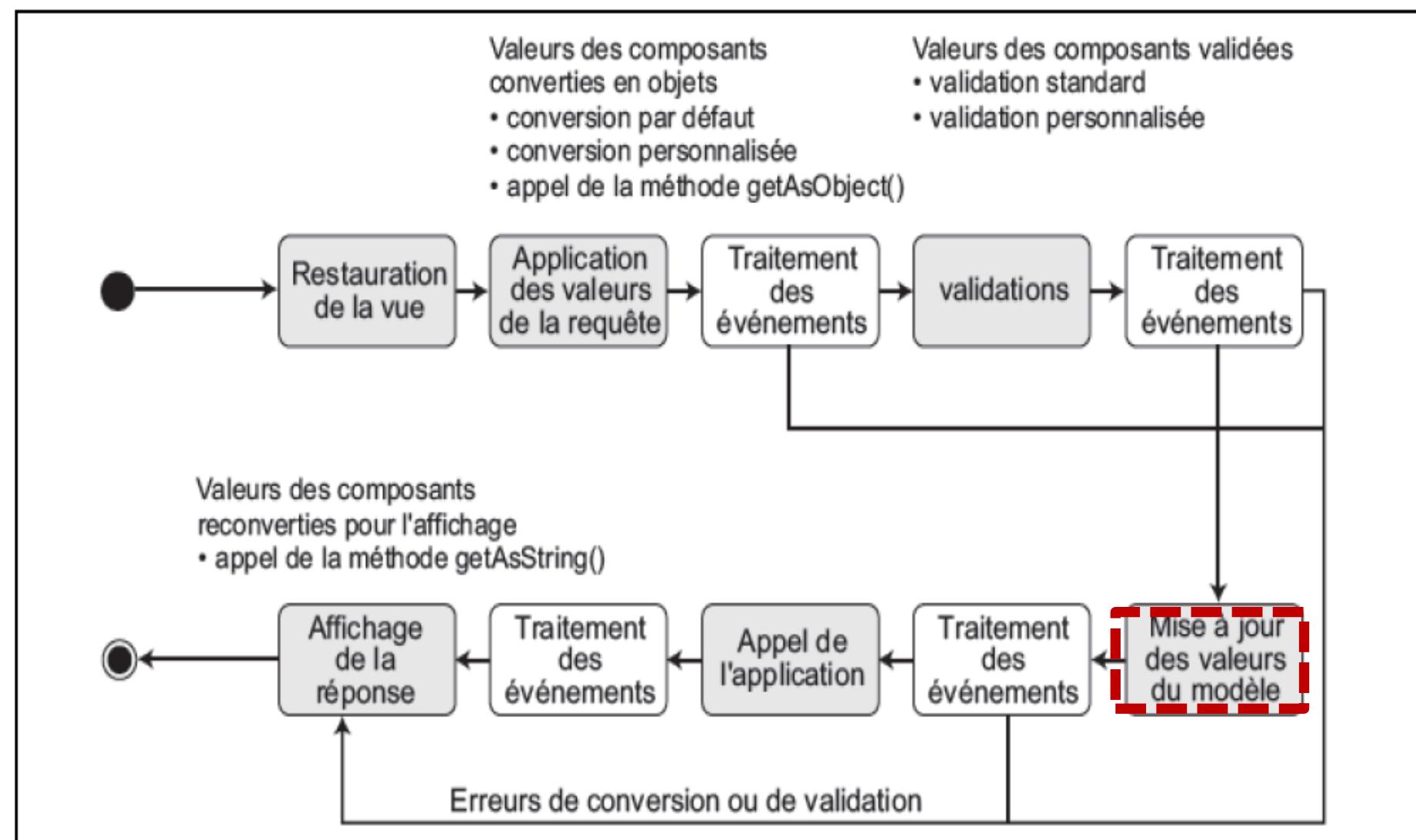


Phase de validation

- Toutes les validations des données traitées à l'étape précédentes sont exécutées,
- Les données sont converties dans le bon type Java,
- Si une validation échoue, la main est donnée à la phase de rendu de la réponse.

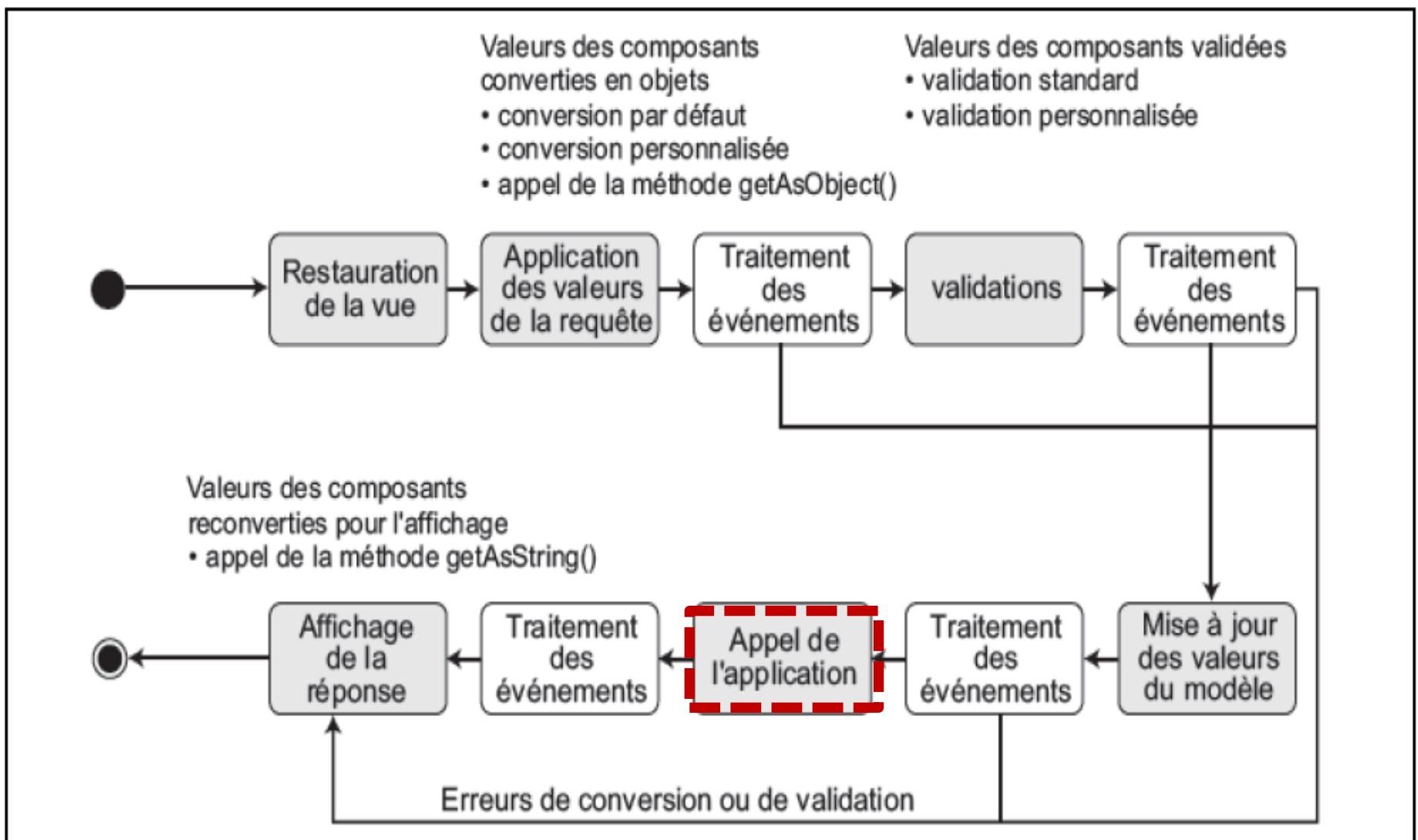
Phase de validation

- Comment c'est fait : chaque composant valide et convertit les données qu'il contient,
- Si un composant détecte une valeur non valable, il met sa propriété « valid » à false et il met un message d'erreur dans la file d'attente des messages (ils seront affichés lors de la phase de rendu « render response ») et les phases suivantes sont sautées.



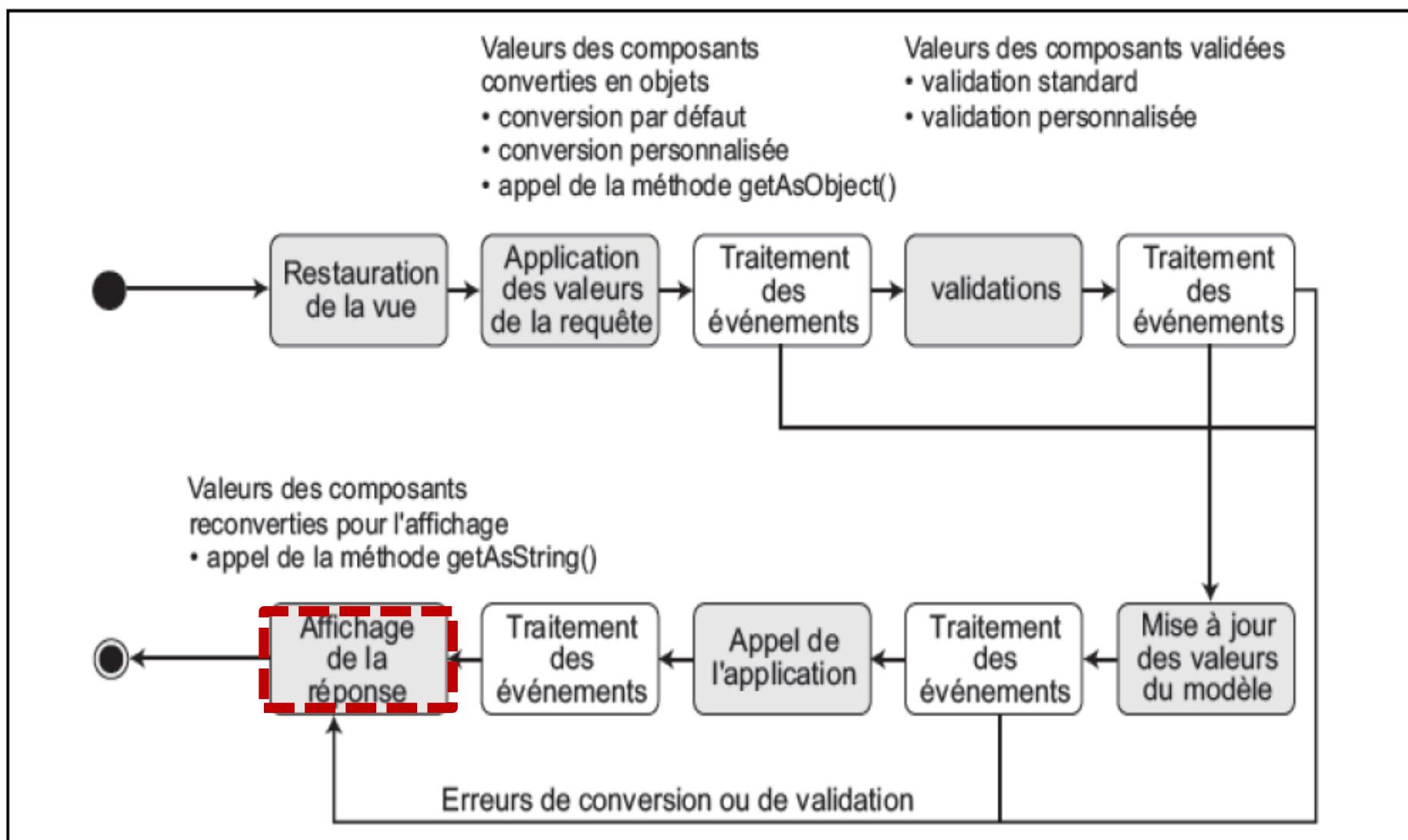
Phase de mise à jour du modèle

- Si les données ont été validées, elles sont mises dans les variables d'instance des Java beans associés aux composants de l'arbre des composants,



Phase d'invocation de l'application

- Les actions associées aux boutons ou aux liens sont exécutées,
- Le plus souvent le lancement des processus métier se fait par ces actions,
- La valeur de retour de ces actions va déterminer la prochaine page à afficher (navigation),



Phase de rendu de la réponse

- La page déterminée par la navigation est encodée en HTML et envoyée vers le client HTTP.

Sauter des phases

- Il est quelquefois indispensable de sauter des phases du cycle de vie,
- Par exemple, si l'utilisateur clique sur un bouton d'annulation, on ne veut pas que la validation des champs de saisie soit effectuée, ni que les valeurs actuelles soient mises dans le modèle,
- Autre exemple : si on génère un fichier PDF à renvoyer à l'utilisateur et qu'on l'envoie à l'utilisateur directement sur le flot de sortie de la réponse HTTP, on ne veut pas que la phase de rendu habituelle soit exécutée.

immediate=true sur un UICommand

- Cet attribut peut être ajouté à un bouton ou à un lien (`<h:commandButton>`, `<h:commandLink>`, `<h:button>` et `<h:link>`) pour faire passer directement (immédiatement) de la phase « Apply request values » à la phase « Invoke Application » (en sautant donc les phases de validation et de mise à jour du modèle),
- Exemple : un bouton d'annulation d'un formulaire.

immediate=true sur un EditableValueHolder

- Cet attribut peut être ajouté à un champ de saisie, une liste déroulante ou des boîte à cocher pour déclencher immédiatement la validation et la conversion de la valeur qu'il contient, avant la validation et la conversion des autres composants de la page,
- Utile pour effectuer des modifications sur l'interface utilisateur sans valider toutes les valeurs du formulaire.

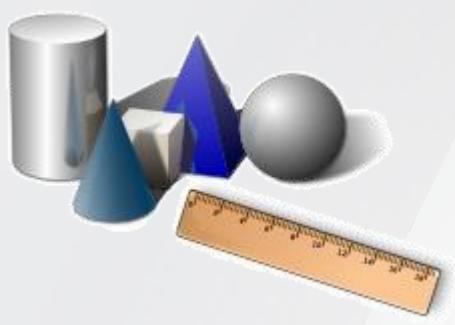
Exemple (1/2)

- Formulaire qui contient champ de saisie du code postal et un champ de saisie de la ville,
- Lorsque le code postal est saisi, un ValueChangeListener met automatiquement à jour la ville :

```
<h:inputText valueChangeListener="#{...}" ...  
onChange = "this.form.submit()" />
```
- La soumission du formulaire déclenchée par la modification du code postal ne doit pas lancer la validation de tous les composants du formulaire qui ne sont peut-être pas encore saisis.

Exemple (2/2)

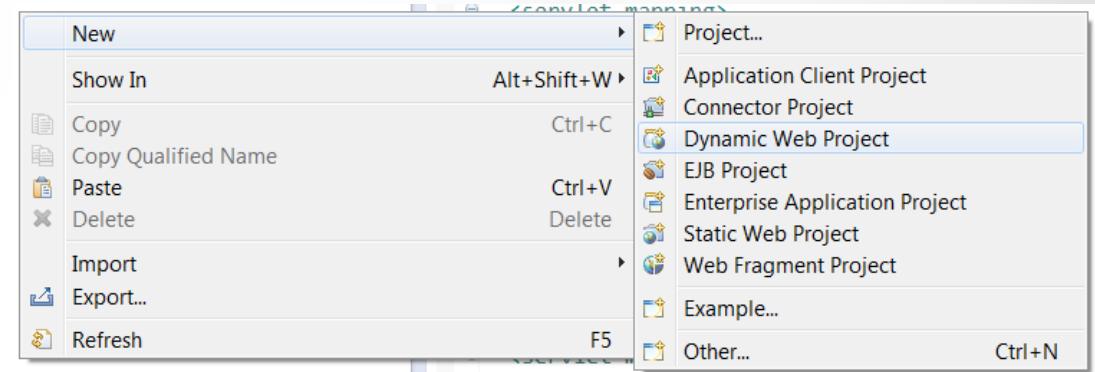
- La solution est de mettre un « immediate=true » sur le champ code postal.



Atelier 7 : HELLO WORLD JSF2

➤ 1 – Création d'un projet Web Dynamique

- 2- Dans la fenêtre suivante tapez : tapez :
 - « helloJSF » comme nom de projet
 - 2,5 comme version Dynamic web module
 - Dans configuration , tapez sur Modify puis cochez « JAVA SERVER FACES 2,2)
 - Cliquez sur finish



Project name:

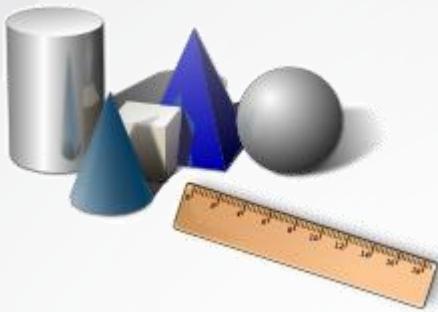
Project location
 Use default location
Location: D:\formation\eclipse\wkspace_archiJava

Target runtime

Dynamic web module version

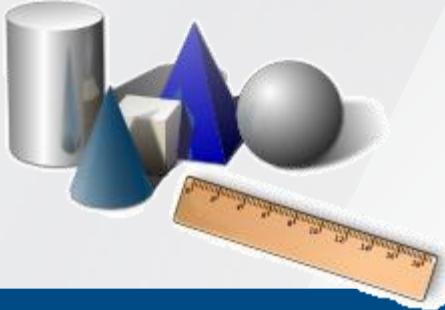
Configuration

Hint: Get started quickly by selecting one of the pre-defined project configurations.



Atelier 7 : HELLO WORLD JSF2

- Convertissez le projet en Maven :
 - Clique droite – configure – convert to maven Project
 - Tapez les paramètres suivantes :
 - <groupId>helloJSF</groupId>
 - <artifactId>helloJSF</artifactId>
 - <version>0.0.1-SNAPSHOT</version>
 - <packaging>war</packaging>
- Puis sur finish
- ➔ le projet est maintenant mavenisé

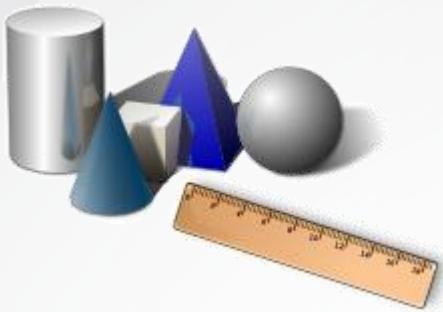


Atelier 7 :

HELLO WORLD

JSF2

```
<dependencies>
    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>jsf-api</artifactId>
        <version>2.2.4</version>
    </dependency>
    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>mojarra-jsf-impl</artifactId>
        <version>2.0.0-b04</version>
    </dependency>
    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>mojarra-jsf-api</artifactId>
        <version>2.0.0-b04</version>
    </dependency>
    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>jsf-impl</artifactId>
        <version>2.2.4</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>3.0-alpha-1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
    </dependency>
</dependencies>
```



Atelier 7 : HELLO WORLD JSF2

➤ Création du managed bean

- Choisissez ajout nouvelle classe : new – class
- Donner comme nom : HelloMbean
- Ecrivez le code

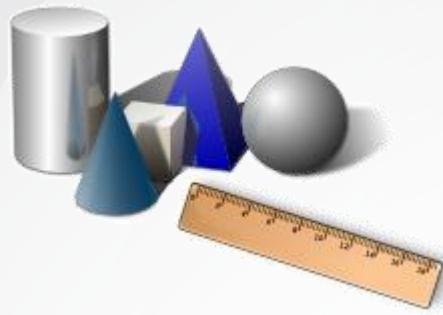
Suivant :

```
package com.wha.MBean;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class HelloMbean {
    private static final long serialVersionUID = 1L;
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```



Atelier 7 : HELLO WORLD JSF2

➤ Création des 2 vues :

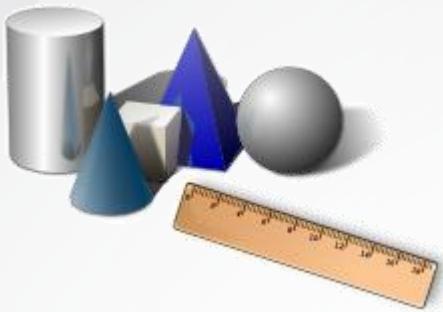
hello.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">

<h:head>
<title>First JSF Example</title>
</h:head>
<h:body>
<h3>JSF 2.2 Hello World Example</h3>
<h:form>
What's your name?
<h:inputText
value="#{helloMbean.name}"></h:inputText>
<h:commandButton value="Welcome Me"
action="welcome"></h:commandButton>
</h:form>
</h:body>
</html>
```

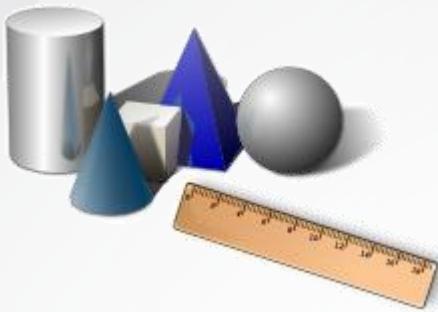
welcome.xhtml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:h="http://java.sun.com/jsf/html">
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1" />
<title>Welcome</title>
</head>
<body>
<h3>Everything went right!</h3>
<h4>Welcome #{HelloMbean.name}</h4>
</body>
</html>
```



➤ Atelier / Ajout de la déclaration des JSF, e HELLO WORLD JSF2

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>helloJSF</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-
      class>javax.faces.webapp.FacesServlet</servlet-
      class>
      <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>*.xhtml</url-pattern>
    </servlet-mapping>
  </web-app>
```



Atelier 7 : HELLO WORLD JSF2

- Testez l'application
- Clique droite sur hello.xhtml puis run on server

http://localhost:8080/helloJSF/faces/hello.xhtml

JSF 2.2 Hello World Example

What's your name?

Questions ?



Navigation entre pages JSF : Handler de navigation

- La navigation entre les pages indique quelle page est affichée quand l'utilisateur clique sur un bouton pour soumettre un formulaire ou sur un lien,
- Cette navigation est déléguée au handler de navigation,
- La navigation peut être définie par des règles dans le fichier de configuration faces-config.xml **ou par des valeurs écrites dans le code Java**

Navigation statique et dynamique

- La navigation peut être statique : définie « en dur » au moment de l'écriture de l'application,
- La navigation peut aussi être dynamique : définie au moment de l'exécution par l'état de l'application (en fait par la valeur renvoyée par une méthode)

Dans les pages JSF

➤ Dans les pages JSF on indique la valeur ou la méthode associée à un bouton qui déterminera la navigation.

➤ Avec une valeur :

```
<h:commandButton value="Texte du bouton"  
    action= "pageSuivante"/>
```

➤ Avec une méthode (qui retourne un nom de page) :

```
<h:commandButton value="Texte du bouton"  
    action="#{nomBean.nomMethode}"/>
```

Dans le fichier de configuration faces-config.xml

```
<navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>succes</from-outcome>
        <to-view-id>/succes.xhtml</to-view-id>
    <navigation-case>
        <navigation-case>
            <from-outcome>echec</from-outcome>
            <to-view-id>/echec.xhtml</to-view-id>
        <navigation-case>
    </navigation-rule>
```

Compléments

- S'il n'y a pas d'élément `<from-view-id>`, la règle de navigation s'applique à toutes les pages
- On peut mettre un joker à la fin de la valeur de `<from-view-id>` :
`<from-view-id>/truc/*</from-view-id>`
- On peut ajouter `<redirect/>` à la fin de la règle de navigation pour que l'URL de la page soit modifié (redirection à la place de *forwarding*) ; intéressant si l'utilisateur peut avoir à enregistrer un signet (*bookmark*)

Compléments

- Plusieurs boutons peuvent renvoyer la même valeur,
- Si on veut distinguer la navigation suivant le bouton, on peut ajouter un élément **<from-action>**,

Navigation par défaut

- Par défaut, JSF travaille avec des requêtes POST
- Depuis JSF 2.0 il est plus simple de travailler avec des requêtes GET, ce qui facilite l'utilisation de l'historique et marque-pages (*bookmarks*) des navigateurs et évite des doubles validations de formulaire non voulues par l'utilisateur.

Utiliser GET

- 2 composants de JSF 2.0 permettent de générer des requêtes GET :
`<h:button>` et `<h:link>`

Le problème avec POST

- Avec une requête POST envoyée pour soumettre un formulaire
 - le refresh de la page affichée (ou un retour en arrière) après un POST soumet à nouveau le formulaire ; le navigateur prévient l'utilisateur en affichant une fenêtre popup inquiétante et difficile à comprendre pour l'utilisateur,
 - l'adresse de la page affichée après le POST est la même que celle du formulaire (donc pas possible de faire réafficher cette page en utilisant l'historique du navigateur)

La raison du problème

- C'est le servlet qui intercepte les requêtes attachées aux pages JSF qui redirige vers la page désignée par le modèle de navigation JSF,
- Le navigateur n'a pas connaissance de cette direction et pense être toujours dans la page qui contient le formulaire qui a été soumis.

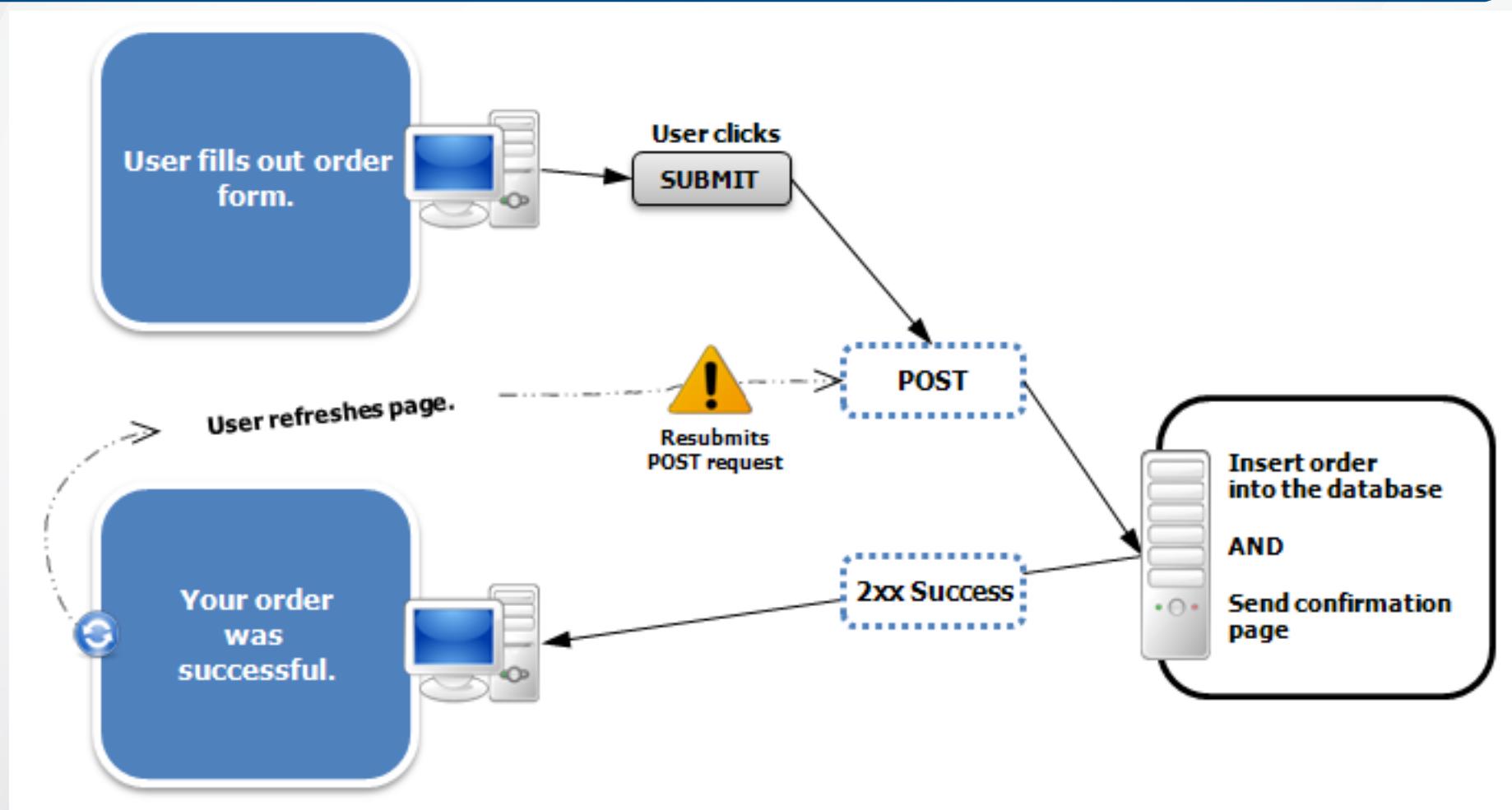
Les conséquences du problème

- Le navigateur est en retard d'une page pour afficher l'URL de la page en cours,
- Il ne garde donc pas la bonne adresse URL si l'utilisateur veut garder un marque-page,
- Le navigateur pense être toujours dans la page qui contient le formulaire après un retour en arrière ou un refresh et il essaie de le soumettre à nouveau (il demande malgré tout une confirmation lors de la soumission multiple d'un formulaire).

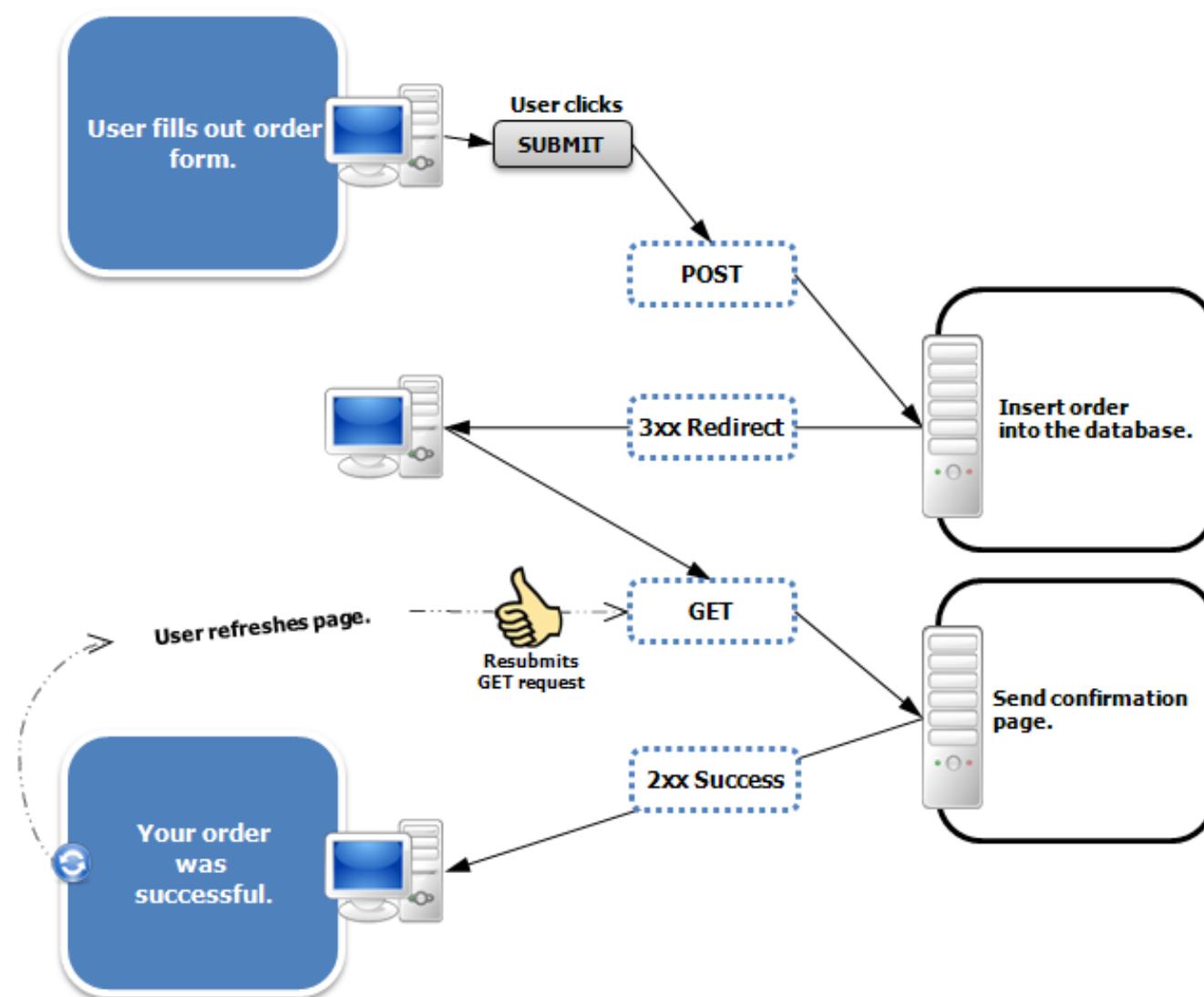
La solution : POST, REDIRECT, GET (PRG)

- Le modèle POST–REDIRECT–GET préconise de
 - Ne jamais montrer une page en réponse à un POST,
 - Charger les pages uniquement avec des GET,
 - Utiliser **la redirection** pour passer de POST à GET.

Sans PRG (POST-REDIRECT-GET)



Avec PRG



Problème de PRG

- PRG peut poser un problème lorsque la page vers laquelle l'utilisateur est redirigée (le GET), doit afficher des données manipulées par le formulaire,
- C'est le cas, par exemple, lorsque cette page est une page qui confirme l'enregistrement dans une base de données des informations saisies par l'utilisateur dans le formulaire,
- En effet, les informations conservées dans la portée de la **requête** du POST ne sont plus disponibles.

Solutions

- Une des solutions est de ranger les informations dans la session plutôt que dans la requête,
- Cependant cette solution peut conduire à une session trop encombrée,
- Une autre solution est de passer les informations d'une requête à l'autre,
- JSF 2.0 offre 2 nouvelles possibilités qui facilitent la tâche du développeur :
 - la mémoire flash,
 - les paramètres de vue,
 - La portée conversation de CDI.

Paramètres de vue

- Les paramètres d'une vue sont définis par des balises `<f:viewParam>` incluses dans une balise `<f:metadata>` (à placer au début, avant les `<h:head>` et `<h:body>`) :

```
<f:metadata>
    <f:viewParam name="n1"
                  value="#{bean1.p1}" />
    <f:viewParam name="n2"
                  value="#{bean2.p2}" />
<f:metadata>
```

<f:viewParam>

- L'attribut name désigne le nom d'un paramètre HTTP de requête GET,
- L'attribut value désigne (par une expression du langage EL) le nom d'une propriété d'un bean dans laquelle la valeur du paramètre est rangée,
- Important : il est possible d'indiquer une conversion ou une validation à faire sur les paramètres, comme sur les valeurs des composants saisis par l'utilisateur.

<f:viewParam> (suite)

- Un URL vers une page qui contient des balises <f:viewParam> contiendra tous les paramètres indiqués par les <f:viewParam> s'il contient « **includeViewParams=true** »
- Exemple :

```
<h:commandButton value=...
    action="page2?faces-redirect=true
&amp;includeViewParams=true"
```

 - Dans le navigateur on verra l'URL avec les paramètres HTTP.

Fonctionnement de includeViewParams

1. La page de départ a un URL qui a le paramètre **includeViewParams**,
2. Elle va chercher les **<f:viewParam>** de la page de destination. Pour chacun, elle ajoute un paramètre à la requête GET en allant chercher la valeur qui est indiquée par l'attribut value du **<f:viewParam>**,
3. A l'arrivée dans la page cible, la valeur du paramètre est mise dans la propriété du bean indiquée par l'attribut value

Fonctionnement de includeViewParams

- Cela revient à faire passer une valeur d'une page à l'autre,
- Si la portée du bean est la requête et qu'il y a eu redirection, cela revient plus précisément à faire passer la valeur d'une propriété d'un bean dans un autre bean (de la même classe).

Donner une valeur à un paramètre

- Il y a plusieurs façons de donner une valeur à un paramètre de requête GET ; les voici dans l'ordre de priorité inverse (la dernière façon l'emporte)
 - Dans la valeur du outcome

```
<h:link outcome="page?p=4&p2='bibi' " ...>
<h:link outcome="page?p=#{bean.prop + 2} " ...>
```
 - Avec les paramètres de vue

```
<h:link outcome="page" includeViewParams="true" ...>
```
 - Avec un <f:param>

```
<h:link outcome="page" includeViewParams="true" ...>
  <f:param name=p value=.../> </h:link>
```

Exemple ; page2.xhtml

```
<!DOCTYPE ...>

<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns=...>

<f:metadata>
    <f:viewParam name="param1"
                  value="#{bean.prop1}"/>
    <f:viewParam name="param2" value="#{...}"/>
</f:metadata>

<h:head>...</h:head>

<h:body>
    Richard Grin
    Valeur : #{bean.prop1}
</h:body>
```

Fonctionnement

- Si **page2** est appelé par la requête GET suivante,
page2.xhtml?param1=v1¶m2=v2
la méthode **setProp1** du bean est appelée avec l'argument v1 (idem pour param2 et v2),
- Si un paramètre n'apparaît pas dans le GET, la valeur du paramètre de requête est **null** et le *setter* n'est pas appelé (la propriété du bean n'est donc pas mise à null).

Bookmarker des URL de page

- Outre le fait qu'un refresh ne provoque plus de soumission du formulaire, le modèle PRG permet aussi de permettre de conserver un URL utile dans un marque-page ou dans l'historique,
- En effet, l'URL contient les paramètres qui permettront de réafficher les mêmes données à un autre moment,
- Sans PRG, les données utiles sont conservées dans l'entité de la requête et pas dans l'URL.

preRenderView Listener (1)

- Une situation courante : il est possible de ne mettre en paramètre qu'un identificateur des données qui ont été traitées,
- Par exemple, que la clé primaire des données enregistrées dans la base de données,
- Avec les procédés précédents on peut faire passer cet identificateur dans la page qui va afficher les données.

preRenderView Listener (2)

- Mais ça ne suffit pas ; il faut utiliser une autre nouveauté de JSF 2.0, les listeners « preRenderView » qui permet **d'exécuter une action avant l'affichage de la vue**,
- On peut ainsi aller rechercher (par exemple dans une base de données) les informations identifiées par l'identificateur **depuis la vue**,
- Pour le GET, le code du *listener* joue en quelque sorte le rôle de l'action pour un POST.

Exemple de preRenderView

```
<f:metadata>  
    <f:viewParam name="id"  
        value="#{bean.IdClient}"/>  
    <f:event type="preRenderView"  
        listener="#{bean.chargeClient}"/>  
</f:metadata>  
...  
Nom : #{bean.client.nom}  
...
```

Managed Beans

- Classes pour représenter les données de formulaires

Plan : nous étudierons

- Les différences entre Beans basiques et “managed beans”
- Le contenu des beans dans dans JSF :
 1. Des gettes/setters qui correspondent aux <input.../> de formulaires,
 2. Des méthodes “action controller” par ex : verifyLoginPassword(...)
 3. Des variables pour stocker des résultats (qu’on appelle “propriétés”).
- Comment pré-remplir les champs des fomulaires
 - Notamment les champs <input../> et les menus/listes déroulantes

Bean basique et bean managé

- Rappel : un bean c'est une classe java qui suit certaines conventions
 - Constructeur vide,
 - Pas d'attributs publics, les attributs doivent avoir des getter et setters, on les appelle des propriétés.
- Une propriété n'est pas forcément un attribut
 - Si une classe possède une méthode getTitle() qui renvoie une String alors, on dit que la classe possède une propriété « title »,
 - Si book est une instance de cette classe, alors dans une page JSF `#{{book.title}}` correspondra à un appel à getTitle() sur l'objet book

Bean basique et bean managé

- Une propriété booléenne sera définie par la méthode `isValid()`, non pas `getValid()`,
- Ce sont bien les méthodes qui définissent une « propriété », pas un attribut. On peut avoir `isValid()` sans variable « valid ».

Bean basique et bean managé

- Règle pour transformer une méthode en propriété
 - Commencer par get, continuer par un nom capitalisé, ex `getFirstName()`,
 - Le nom de la propriété sera `firstName`
 - On y accèdera dans une page JSF par `#{{customer.firstName}}` où `customer` est l'instance du bean et `firstName` le nom de la propriété,
 - Cela revient à appeler la méthode `getFirstName()` sur l'objet `customer`.

Bean basique et bean managé

- Exception 1 : propriétés booléennes
 - getValid() ou isValid() (recommandé),
 - Nom de la propriété : valid
 - Accès par #{login.valid}
- Exception 2 : propriétés majuscules
 - Si deux majuscules suivent le get ou le set, la propriété est toute en majuscule,
 - Ex : getURL(),
 - Propriété : URL,
 - Accès par #{website.URL}

Exemples de propriétés

Nom de méthode	Nom de propriété	Utilisation dans une page JSF
getFirstName setFirstName	firstName	<code>#{customer.firstName}</code> <code><h:inputText value="#{customer.firstName}" /></code>
isValid setValid (booléen)	valid	<code>#{login.valid}</code> <code><h:selectBooleanCheckbox value="#{customer.executive}" /></code>
getValid setValid (booléen)	valid	<code>#{login.valid}</code> <code><h:selectBooleanCheckbox value="#{customer.executive}" /></code>
getURL setURL	URL	<code>#{address.ZIP}</code> <code><h:inputText value="#{address.ZIP}" /></code>

Pourquoi ne pas utiliser d'attributs publics

➤ 1) Règle d'or des beans

- Mauvais

```
public double vitesse;
```

- Bon

```
private double v;
```

```
public double getVitesse() {  
    return v;  
}  
public void setVitesse(double v) {  
    this.v = v;  
}
```

➤ Utilisez les wizards de vos IDEs pour générer du code pour des propriétés (netbeans : clic droit/insert code)

Pourquoi ne pas utiliser d'attributs publics

➤ 2) On peut mettre des contraintes

```
public void setVitesse(double v) {  
    if(v > 0) {  
        this.v = v;  
    } else {  
        envoyerMessageErreur(); // ou exception...  
    }  
}
```

➤ 3) On peut faire plus : ex notifier un changement de valeur

```
public void setVitesse(double v) {  
    if(v > 0) {  
        this.v = v;  
        updateCompteurSurTableauDeBord(v);  
    } else {  
        envoyerMessageErreur();  
    }  
}
```

Pourquoi ne pas utiliser d'attributs publics

- 4) on peut changer la représentation interne de la variable sans changer son interface d'utilisation
 - Par ex : stocker la vitesse int alors qu'on a des getters/setters qui fonctionnent en double...

Les trois composantes des beans managés

- Des propriétés (donc définies par des get/set ou is...)
 - Une paire pour chaque élément input de formulaire,
 - Les setters sont automatiquement appelés par JSF lorsque le formulaire sera soumis. Appelé avant les « méthodes de contrôle »;
- Des méthodes « action controller »
 - Une par bouton de soumission dans le formulaires (un formulaire peut avoir plusieurs boutons de soumission),
 - La méthode sera appelée lors du clic sur le bouton par JSF

Les trois composantes des beans managés

- Des propriétés pour les données résultat
 - Seront initialisées par les méthodes « action controller » après un traitement métier,
 - Il faut au moins une méthode get sur la propriété afin que les données puissent être affichées dans une page de résultat.

Caractéristiques des Beans Managés

- JSF « manage » le bean
 - Il l'instancie automatiquement,
 - Nécessité d'avoir un constructeur par défaut
 - Contrôle son cycle de vie
 - Ce cycle dépend du « scope » (request, session, application, etc.)
 - Appelle les méthodes setter
 - Par ex pour <h:inputText value="#{customer.firstName}"/>, lorsque le formulaire est soumis, le paramètre est passé à setFirstName(...)
 - Appelle les méthodes getter
 - #{customer.firstName} revient à appeler getFirstName()
- Déclaration par @ManagedBean avant la classe

Un exemple simple (1)

- Scénario
 - Entrer l'id d'un client et son password,
 - Afficher en résultat :
 - Une page affichant son nom, prénom et solde de son compte bancaire,
 - 3 pages différentes selon la valeur du solde,
 - Une page d'erreur si le formulaire a été mal rempli (données manquantes ou incorrectes)

Un exemple simple (2)

- De quoi a besoin de bean managé ?
 - Propriétés correspondant aux éléments du formulaire d'entrée : ex getCustomerId/setCustomerId, etc.
 - Méthodes « action controller » :
 - Pour récupérer un Customer à partir de l'Id.
 - De quoi stocker les résultats
 - Stocker le Customer résultat dans une variable d'instance, initialement vide, avec get/set associés

BanqueMBean partie 1 : propriétés pour les éléments de formulaire

```
@ManagedBean  
public class BanqueMBean {  
    private String customerId, password;  
  
    public String getCustomerId() {  
        return customerId;  
    }  
  
    public void setCustomerId(String customerId) {  
        this.customerId = customerId;  
    }  
  
    public String getPassword() {  
        return password;  
    }  
  
    public void setPassword(String password) {  
        this.password = password;  
    }  
}
```

Appelé par JSF lors de l'affichage du formulaire, comme la variable vaut null au départ le champs sera vide.

Lors de la soumission, le bean, est instancié à nouveau (puisque il est RequestScoped par défaut), et la valeur dans le formulaire est Passée à cette méthode

get et setPassword sont identiques à get et setCustomerId, à part qu'on ne peut Pré-remplir un champ password (interdit par Les navigateurs), donc getPassword ne sera Pas appelé initialement

Les « scopes »

- Ils indiquent la durée de vie des managed beans,
- Valeurs possibles : request, session, application, view, conversation, aucun ou custom
- RequestScope = valeur par défaut.
- On les spécifie dans faces-config.xml ou sous forme d'annotations de code (recommandé)

Annotations pour les Scopes

- @RequestScoped
 - On crée une nouvelle instance du bean pour chaque requête.
 - Puisque les beans sont aussi utilisés pour initialiser des valeurs de formulaire, ceci signifie qu'ils sont donc généralement instanciés deux fois (une première fois à l'affichage du formulaire, une seconde lors de la soumission)
 - Ceci peut poser des problèmes...

Annotations pour les Scopes

➤ @SessionScoped

- On crée une instance du bean et elle durera le temps de la session. Le bean doit être Sérializable.
- Utile par exemple pour gérer le statut « connecté/non connecté » d'un formulaire login/password.
- On utilisera les attributs « render » des éléments de UI pour afficher telle ou telle partie des pages selon les valeurs des variables de session.
- Attention à ne pas « abuser » du SessionScoped, pièges possibles avec les variables cachées ou les éditions de données multiples (cf tp1)

Annotations pour les Scopes

➤ @ApplicationScoped

- Met le bean dans « l'application », l'instance sera partagée par tous les utilisateurs de toutes les sessions. Le bean est en général stateless (sans attributs d'état).
- Pour des méthodes utilitaires uniquement.

Annotations pour les Scopes

- @ViewScoped
 - La même instance est utilisée aussi souvent que le même utilisateur reste sur la même page, même s'il fait un refresh (reload) de la page !
 - Le bean doit être sérializable,
 - A été conçu spécialement pour les pages JSF faisant des appels Ajax (une requête ajax = une requête HTTP -> une instance si on est en RequestScoped !)
 - On utilise souvent des event handlers dans les pages JSF avec ce type de bean (tp1)

Annotations pour les Scopes

- @ConversationScoped
 - Utilise CDI, ne fait pas partie de JSF, **@Named obligatoire (pas @ManagedBean)**
 - Semblable aux session mais durée de vie gérée « par programme »,
 - Utile pour faire des wizards ou des formulaires remplis partiellement au travers de plusieurs pages;
 - On va confier à une variable injectée le démarrage et la fin de la durée de vie du bean,
- @CustomScoped(value=« #{uneMap} »)
 - Le bean est placé dans la HashMap et le développeur gère son cycle de vie.

Annotations pour les Scopes

- @NoneScoped
 - Le bean est instancié mais pas placé dans un Scope.
 - Utile pour des beans qui sont utilisés par d'autres beans qui sont eux dans un autre Scope.

Où placer ces annotations

- Après @Named ou @ManagedBean en général,
 - Named recommandé si on a le choix.
- Attention aux imports !!!!!
 - Si @ManagedBean, alors les scopes doivent venir du package javax.faces.bean A EVITER !
 - Si @Named les scopes doivent venir de javax.enterprise.context !
 - Cas particuliers : ConversationScoped que dans javax.enterprise.context donc utilisé avec @Named, et ViewScoped vient de javax.faces.bean donc utilisé avec @ManagedBean
- En gros JSF = package javax.faces.xxx et CDI = javax.enterprise.xxx

Exemple ApplicationScoped

```
package coreservlets;

import javax.faces.bean.*;

@ManagedBean
@ApplicationScoped
public class Navigator {
    public String choosePage() {
        String[] results =
            { "page1", "page2", "page3" };
        return(RandomUtils.randomElement(results));
    }
}
```

@SessionScoped

- Idée : gestionnaire login password. Si l'utilisateur se trompe de password on re-affiche quand même le champs login rempli, sinon on dirige vers la page d'accueil et on mémorise dans une variable le fait qu'on est authentifié.
- Backing Bean :
 - Serializable, on peut ajouter faces-redirect=true à la fin du return pour faire un redirect au lieu d'un forward vers la page de résultats.
 - Ceci permettra d'accéder directement à la page de résultats, aussi.

@SessionScoped, page JSF login/password

```
<h:outputText rendered="#{!loginMBean.connected}"  
             value="#{loginMBean.message}" />  
  
<h:form id="loginForm"  
         rendered="#{!loginMBean.connected}">  
  
    Username: <h:inputText id="loginForm"  
                  value="#{loginMBean.login}" />  
  
    Username: <h:inputText id="password"  
                  value="#{loginMBean.password}" />  
  
    <h:commandButton value="Login"  
                     action="#{loginMBean.checkLogin}" />  
  
</h:form>  
  
... suite transparents suivant !
```

```
<h:form id="deconnexionForm"
    rendered="#{loginMBean.connected}">
    <h:outputText
        value="#{loginMBean.message}" />
    <h:commandButton value="Deconnexion"
        action="#{loginMBean.deconnexion}" />
</h:form>
```

- Note : l'attribut rendered remplace les if/then/else de JSTL !

@SessionScoped, page JSF login/password

```
@Named(value = "loginMBean")
@SessionScoped
public class LoginMBean implements Serializable {

    private String login;
    private String password;
    private boolean connected = false;
    private String message = "Veuillez vous identifier :";

    public boolean isConnected() {

        return connected;
    }

    ...
}
```

SessionScoped, page JSF login/password

```
...
public void deconnexion() {
    connected = false;
    message = "Veuillez vous identifier :";
}
public void checkLogin() {
    connected = (login.equals("michel") &&
                password.equals("toto"));
    if (connected) {
        message = "Bienvenue, vous êtes connecté en tant que "
                  + login + " ! ";
    } else {
        message = "Mauvais login/password, veuillez recommencer";
    }
}
```

@ViewScoped

- Prévu pour les pages faisant des appels Ajax,
 - Pas de nouvelle instance tant qu'on est dans la même page JSF,
- Peu d'infos pertinentes sur le web...

@ConversationScoped

```
import javax.enterprise.context.Conversation;
import javax.enterprise.context.ConversationScoped;
@Named(value = "customerMBean")
@ConversationScoped
public class CustomerMBean implements Serializable {
    @Inject private Conversation conversation;
    public String showDetails(Customer customer) {
        this.customer = customer;      conversation.begin();
        return "CustomerDetails?id=" +      customer.getCustomerId() +
               "&faces-redirect=true";
    }
    public String update() {
        customer = customerManager.update(customer);
        conversation.end();
        return "CustomerList?faces-redirect=true";
    }
}
```

@ConversationScoped

- Danger : à chaque begin() doit correspondre un end()
 - Attention avec des pages qui sont « bookmarkables » (correction tp1 par ex, le formulaire est bookmarkable) et appelables de plusieurs manières,
 - Piège avec les évènements preRenderView : la méthode appelée doit faire un begin et il faut faire un end quand on sort du formulaire, mais comme la page peut être aussi invoquée depuis une autre page, il ne fait pas faire un begin « avant » le preRenderView...
 - Donner exemple...

Accéder aux objets Request et Response

- Pas d'accès automatique !
- Il faut « penser » différemment, il faut considérer les formulaires comme des objets.
- Mauvaise nouvelle : si vous avez quand même besoin d'accéder à la requête et à la réponse, le code est assez horrible...
 - Utile pour : manipuler la session, appeler invalidate() ou régler la durée.
 - Manipulation des cookies explicite, consulter le user-agent, regarder le host, etc.

Exemple de code

```
ExternalContext context =  
    FacesContext.getCurrentInstance().getExternalContext();  
HttpServletRequest request =  
    (HttpServletRequest)context.getRequest();  
HttpServletResponse response =  
    (HttpServletResponse)context.getResponse();
```

Exemple : choix d'un moteur de recherche



Exemple : choix d'un moteur de recherche

```
<!DOCTYPE ...>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
<h:head>...</h:head>
<h:body>
...
<h:form>
    Search String:
    <h:inputText value="#{searchController.searchString}" /><br/>
    Search Engine:
    <h:selectOneMenu value="#{searchController.searchEngine}">
        <f:selectItems value="#{searchController.searchEngineNames}" />
    </h:selectOneMenu><br/>
    <h:commandButton value="Search"
                      action="#{searchController.doSearch}" />
</h:form>
</h:body></html>
```

Exemple : choix d'un moteur de recherche

```
@ManagedBean
public class SearchController {
    private String searchString="", searchEngine;

    public String getSearchString() {
        return(searchString);
    }
    public void setSearchString(String searchString) {
        this.searchString = searchString.trim();
    }
    public String getSearchEngine(){
        return(searchEngine);
    }
    public void setSearchEngine(String searchEngine) {
        this.searchEngine = searchEngine;
    }
    public List<SelectItem> getSearchEngineNames() {
        return(SearchUtilities.searchEngineNames());
    }
}
```

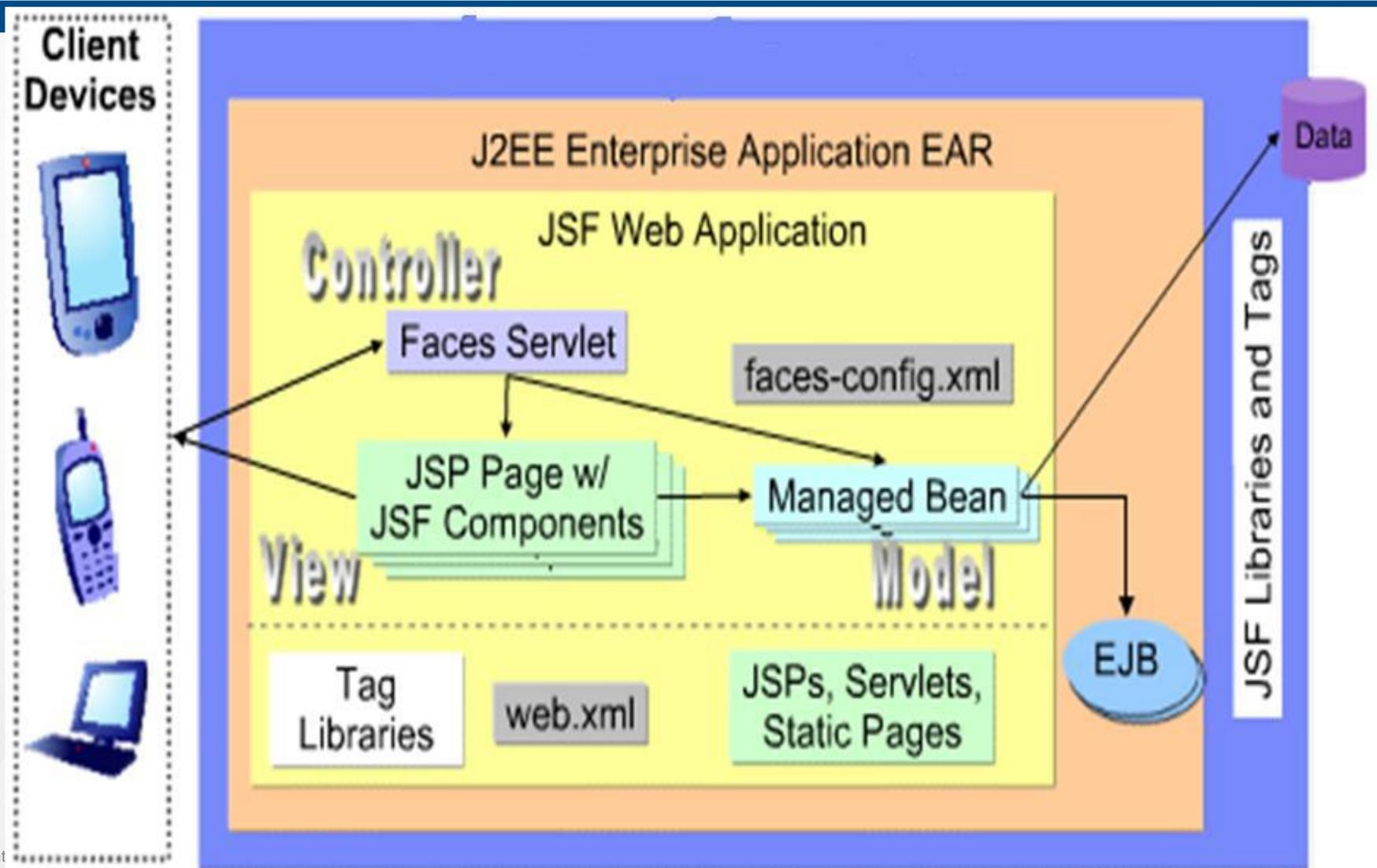
Exemple : choix d'un moteur de recherche

```
public String doSearch() throws IOException {
    if (searchString.isEmpty()){
        return("no-search-string");
    }
    searchString = URLEncoder.encode(searchString, "utf-8");
    String searchURL =
        SearchUtilities.makeURL(searchEngine, searchString);
    if (searchURL != null) {
        ExternalContext context =
            FacesContext.getCurrentInstance().getExternalContext();
        HttpServletResponse response =
            (HttpServletResponse)context.getResponse();
        response.sendRedirect(searchURL);
        return(null);
    } else {
        return("unknown-search-engine");
    }
}
```

Conclusion : Composants de l'architecture JSF

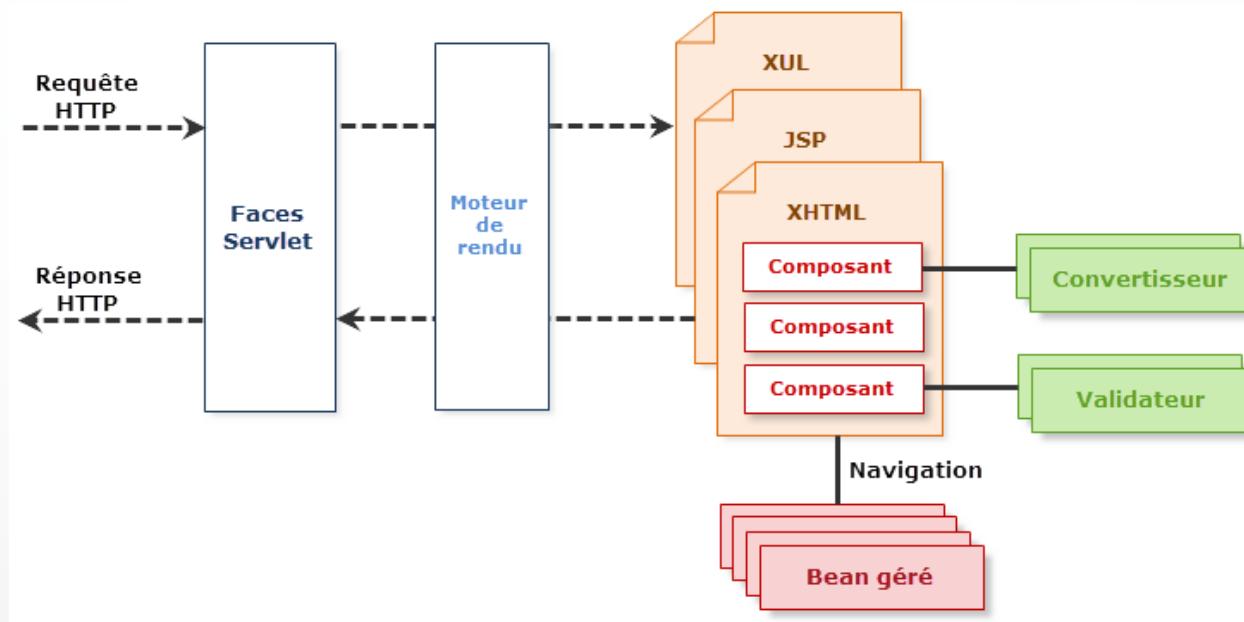
- **Le contrôleur(FacesServlet):**
 - Servlet principal de l'application qui sert de contrôleur. Déjà implémenté dans le Framework. Toutes les requêtes de l'utilisateur passent systématiquement par ce Servlet, qui les examine et appelle les différentes actions correspondantes. Ce contrôleur sera déclaré dans le Web.xml et configuré dans le fichier **faces-config.xml**
- **La vue:pagesweb(JSP,JSF,XHTML):**
 - D'autre type de vue existe, comme **WML** pour les dispositifs mobiles. La version **JSF2.0** utilise les **Facelets**. Les **Facelets** sont formées d'une arborescence de composants **UI**(également appelés widgets ou contrôles).
- **Le modèle(ManagedBean/BackingBean):**
 - Classes Java spécialisées qui synchronise les valeurs avec les composantsUI, accède au logique métier et gère la navigation entre les pages.
- **faces-config.xml**
 - Fichier de configuration de l'application définissant les règles de navigation et les différents ManagedBean utilisés.

Conclusion : Composants de l'architecture JSF



Conclusion : Composants de l'architecture JSF

- Outre les composants MVC, JSF est composé de:
 - Moteur de rendu (Rederrer): décode la requête de l'utilisateur pour initialiser les valeurs du composant et encode la réponse pour créer une représentation du composant que le client pourra comprendre et afficher.
 - Convertisseurs et validateurs: Le protocole HTTP est un protocole uniquement textuel, donc nous aurons besoin de valider les champs de saisie textuelle et de les convertir vers les autres types d'Objets.



Conclusion : Les étapes de développements avec JSF

1. Configurer le fichier **web.xml** afin de déclarer le Faces Servlet
2. Développer les objets du modèle qui détiennent les données (**ManagedBean ou les BackingBean**)
3. Déclarer les ManagedBean dans le fichier de configuration de l'application **faces-config.xml**
4. Créer des pages en utilisant les composants d'interface utilisateur UI et les tagLibde JSF
5. Définir les règles de navigation entre les pages dans **faces-config.xml**

LAB CRUD JSF



Sommaire

- Chapitre 1 : INTRODUCTION
- Chapitre 2 : PLATE-FORME JAVA EE
- Chapitre 3 : SERVLET / JSP
- Chapitre 4 : LES BASES DE DONNES AVEC JAVA EE
- Chapitre 5 : JSF
- **Chapitre 6 : PRIMEFACES**



INTRODUCTION :

- Le développement d'interface web avec JSF était limité à cause de peu de composants graphiques disponibles .
- A cause de ce manque de composants, de nombreux projets ont été lancé afin de créer des bibliothèques de composants JSF plus ou moins spécifiques .
- C'est quoi Prime Faces ?

Introduction

- Prime Faces est une bibliothèque open source de composants JSF .
- Il est basé côté serveur sur l'API standard de JSF 2.
- Coté client les scripts de Prime Faces sont basés sur la librairie la plus populaire de JavaScript jQuery .
- Prime Faces vise à garder le traitement propre, rapide et léger.

Historique

- Novembre 2008 - démarrage
- Janvier 2009 – Première version 0.8.0
- Plus que 20 versions jusqu'à présent
- Mars 2018 - 6.2 (Dernière version stable)
 - Support de JSF2.3 + nouveaux composants (>100)
 - Sécurité
 - Fonctionnalités

Les concurrents

- RichFaces
- ICEfaces
- RC Faces
- Open Faces
-



Commencer avec Primefaces

➤ DÉPENDANCES :

- Prime Faces exige seulement
 - un runtime Java 5+
 - JSF 2.0 et plus

➤ Il y a certaines bibliothèques en option pour certaines fonctions.

- Comme par exemple :
 - Itext



- Apache poi



Commencer avec Primefaces

➤ TÉLÉCHARGEMENT :

- PrimeFaces peut être téléchargé soit :
 - Manuellement :
 - primefaces-{version}.jar
 - <http://www.primefaces.org/downloads.html>
 - Via Maven :



```
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>6.2</version>
</dependency>
```

CONFIGURATION

- Prime Faces namespace est nécessaire pour ajouter les composants Prime Faces à votre page :

```
<!DOCTYPE html>
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.org/ui"
      xmlns:pt="http://xmlns.jcp.org/jsf/passthrough">

    <h:head>
    </h:head>

    <h:body>
        <p:inputText value="#{bean.value}" pt:placeholder="Watermark here"/>
    </h:body>

</html>
```

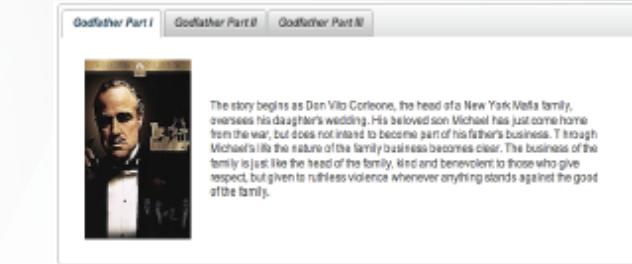
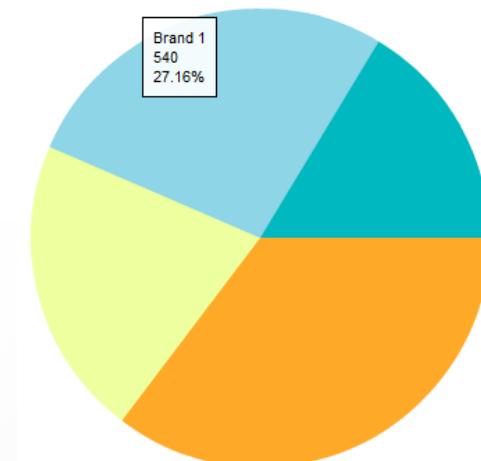
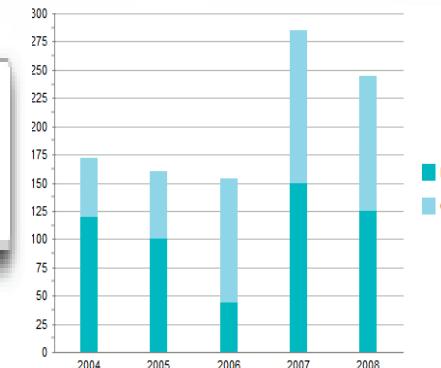
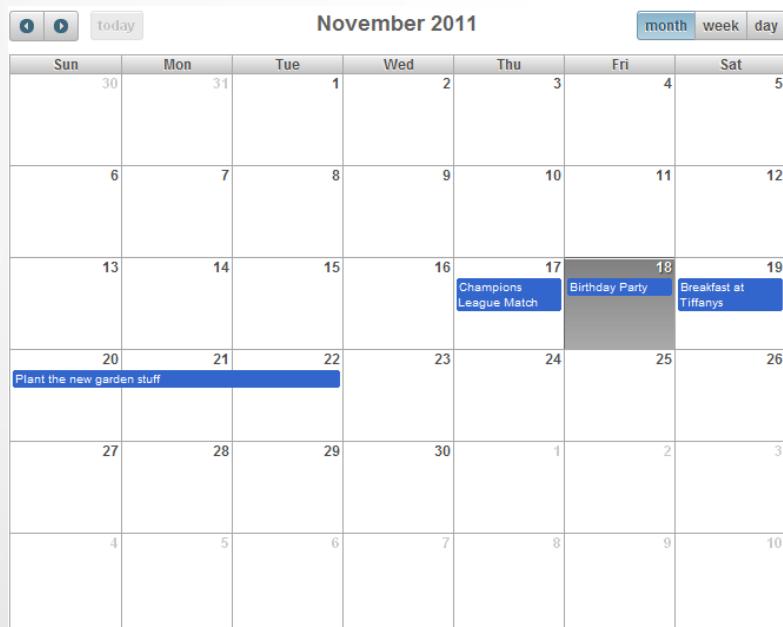
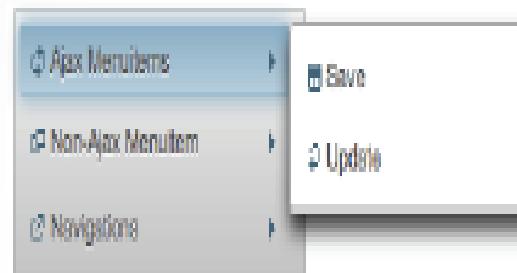
UI component

- Ensemble riche de composants : +100
- Personnalisables et faciles à utiliser
- Compatibles avec d'autre bibliothèques
- Composants légères et simples à intégrer
- Composants à base de jQuery, AJAX

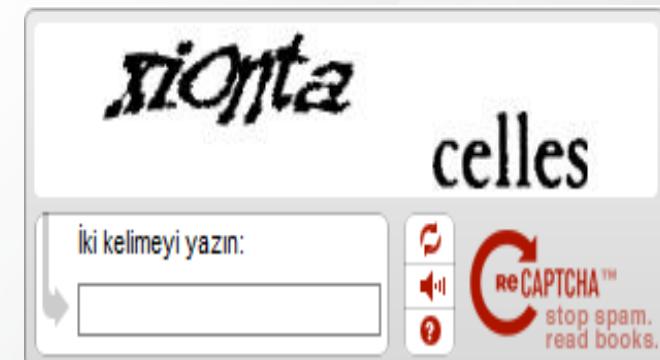
UI component

<https://www.primefaces.org/showcase/>

List of Cars			
Model	Year	Manufacturer	Color
1326edf	1982	Opel	Silver
5995e5s	1975	Audi	Orange
d51ca300	1960	Audi	Yellow
86c82e27	1982	Opel	Red
404e45cf	2002	Renault	Green
25e8dd34	1999	Audi	Green
6958249	1980	Renault	Blue
a79527b	2005	Pontiac	White
7d991773	1991	Volkswagen	White
9367rh4	1971	Renault	Blue



Model: 3590bd4a	Model: 72970a20	Model: 796c6367
Year: 1979	Year: 1997	Year: 2005
Color: Red	Color: White	Color: Green



Les thèmes (+30 THÈMES PRÉDÉFINIES)



Intégration avec la JEE

- Prime Faces peut être soutenu par plusieurs Framework d'application d'entreprise:
 - Spring Core
 - Spring Web Flow
 - EJBs



Conclusion

- Malgré le manque de documentation sur PRIMEFACES, PRIMEFACES se déplace vers la bonne direction à grâce a sa simple mise en place , la légèreté de ces composants et son évolution rapide .
- PRIMEFACES est le meilleur parmi les autres bibliothèques de composants JSF .
 - Une intégration facile dans les IDE
 - Une communauté active
 - Portage vers le Angular et le React
 - Le responsive Design
 - ...

Labs

- Ajouter le support de prime faces a votre dernier projet



Questions ?

