

MRI-COURS 7-8: Servlet et JSP

yoann.maurel@irisa.fr



Certains slides sont basés sur ceux de Yves Bekkers

Objectifs de la séquence

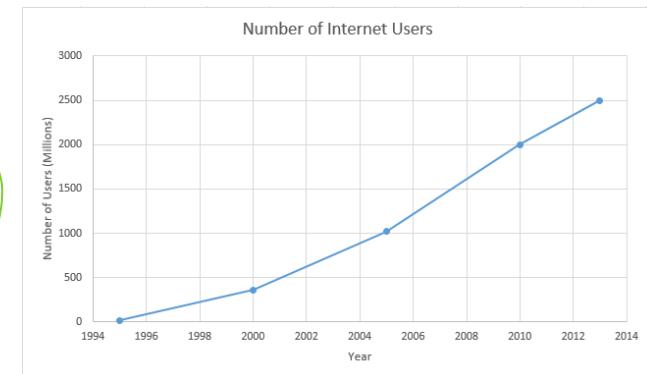
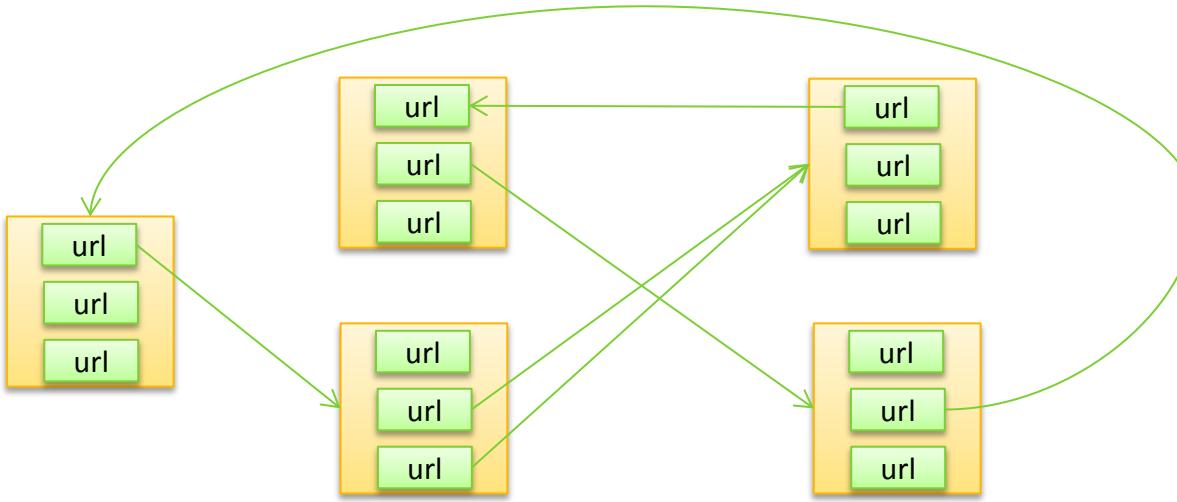
- ▶ Présentation de la programmation Web en JAVA :
 - ▶ HTML
 - ▶ Les servlets
 - ▶ JSP
 - ▶ Moteur de servlet (Tomcat, Jetty)
- ▶ Les services Web et la mise à jour dynamique de pages Web :
 - ▶ SOAP et REST
 - ▶ Ajax
 - ▶ Websockets ?



Introduction au Web

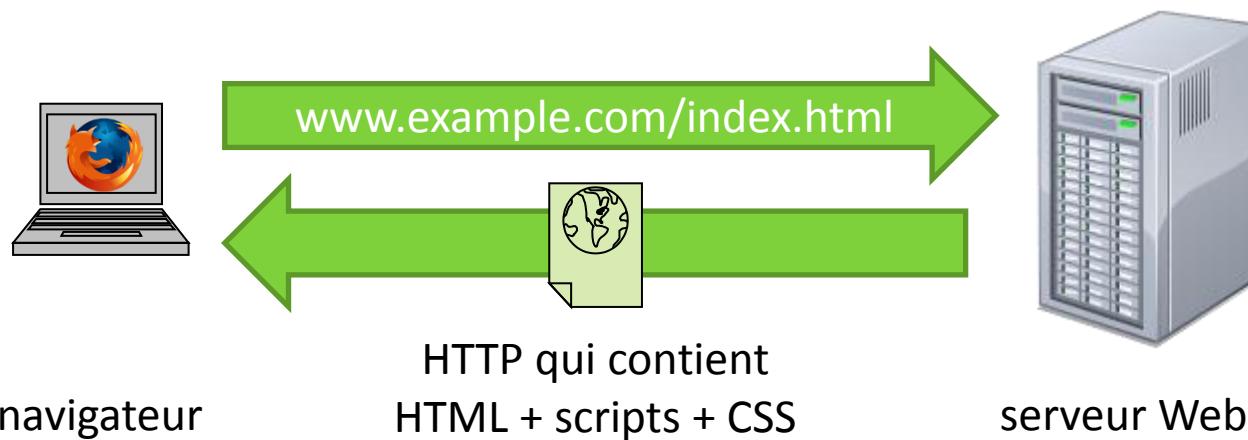
Web

- ▶ Construire un ensemble de documents liées par des liens hypertextes.
 - ▶ initiative CERN, 1989 - 1990, Tim Berners-Lee
 - ▶ essor en 1994 avec Mozaic, Yahoo, etc ...
 - ▶ accompagne le développement d'internet
 - ▶ idée du Web 2.0 en 2003/2004



Web

- ▶ Des adresses standards pour désigner les ressources (URL)
- ▶ Un serveur HTTP permettant de distribuer ces ressources
- ▶ Un langage balisé le HTML permettant de faire des liens entre les documents
- ▶ Des évolutions
 - ▶ pages dynamiques : les CGI, les scripts clients, les scripts serveurs ...
 - ▶ des frameworks, architectures, ... pour la construction d'appli Web
 - ▶ séparation forme/contenu : CSS, préprocesseurs CSS, XML/XSLT

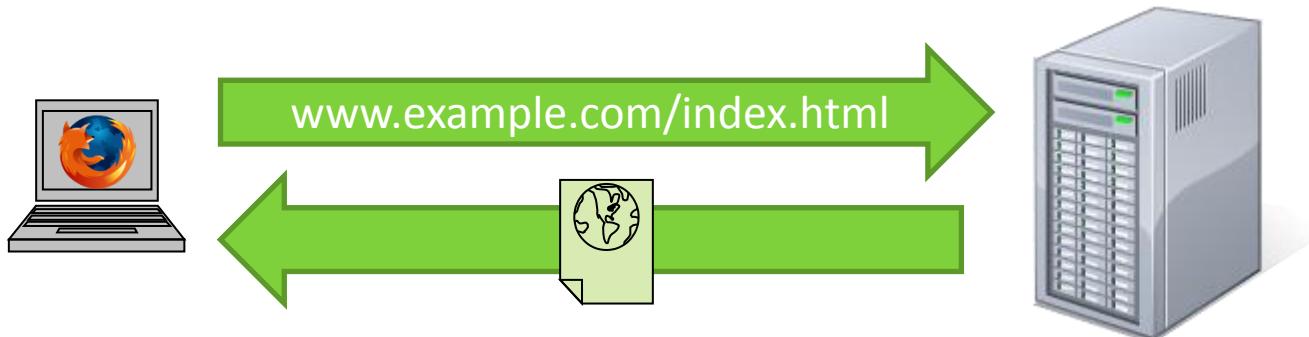




Le Serveur HTTP

Architecture Client/serveur

- ▶ HTTP : le protocole standard du Web
 - ▶ Client Serveur au dessus de TCP
 - ▶ principalement utilisé entre un navigateur web et un serveur Web
 - ▶ mais peut être utilisé de façon autonome (ex. web services)
- ▶ Chaque ressource est désigné par une URI :
 - ▶ nom-de-domaine/chemin-vers-la-ressource/nom-de-la-ressource
 - ▶ www.example.com/a/b/c/index.html



Client/serveur

- ▶ 2 modes de connexion
 - ▶ Connexion non persistante (HTTP < 1.1)
 - ▶ Une connexion TCP pour chaque transaction
 - ▶ Connexion persistante (HTTP >= 1.1)
 - ▶ Une connexion pour toutes les transactions
- ▶ Protocole stateless (sans-mémoire) :
 - ▶ le serveur ne garde pas l'état du client
- ▶ Deux types de messages :
 - ▶ **Requête** :
 - ▶ Exemple : GET /nom_repertoire/page.html http/1.1
 - ▶ **Réponse avec un code d'état** :
 - ▶ Exemples : 200->OK, 404->Not found



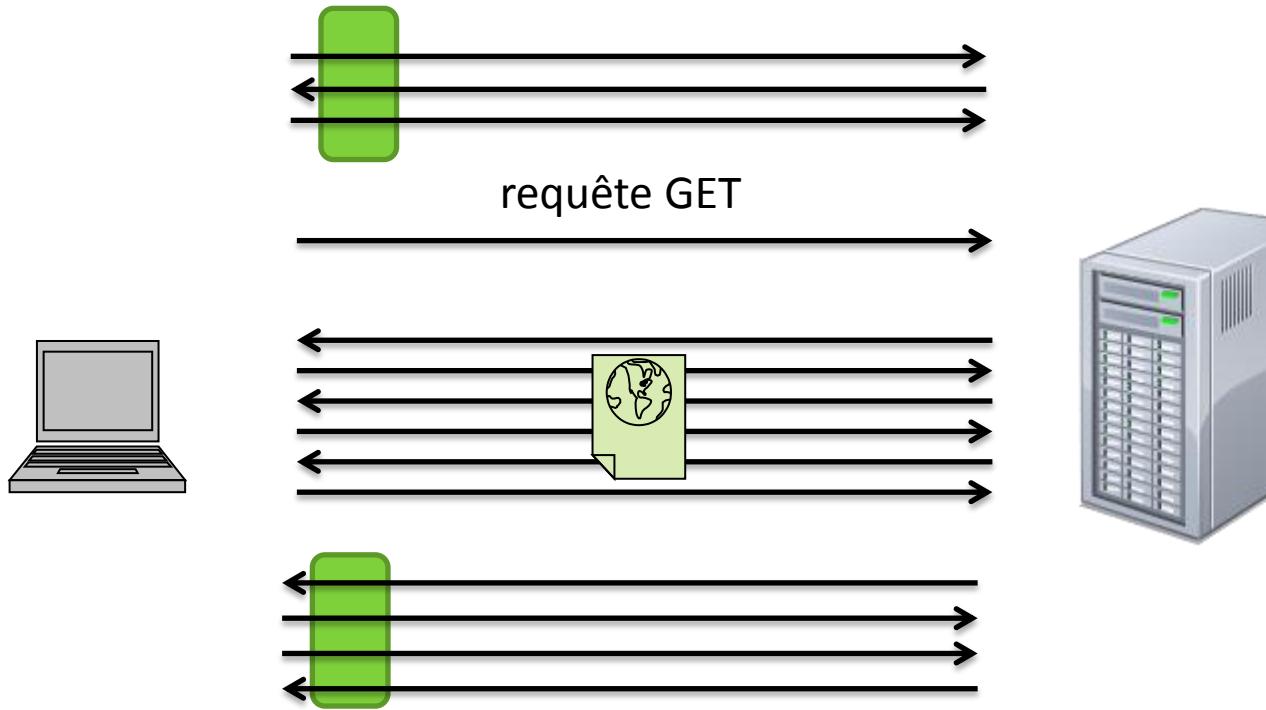
Principales commandes de requêtes

- ▶ Dans HTTP/1.1
 - ▶ GET <URI> : demande l'accès à la ressource.
 - ▶ ex : retourne la page Web complète à l'adresse demandée.
 - ▶ HEAD <URI> : avoir l'en-tête de la ressource
 - ▶ informations de bases sur la ressource
 - ▶ PUT <URI> : envoie une ressource à l'URI et remplace le contenu si nécessaire
 - ▶ POST<URI> : comme PUT mais ajoute les données à l'existant
 - ▶ utilisée notamment pour envoyer des informations de formulaires
 - ▶ DELETE<URI> : demande l'effacement d'une ressource
- ▶ Les commandes sont soumises à autorisation
- ▶ Les commandes sont complétées par des options



Exemple : Obtenir une page Web

ouverture connexion TCP (SYN, ACK, ACK)



fermeture connexion TCP (FIN, ACK) (FIN, ACK)



Format des requêtes et réponses

GET / HTTP/1.1

Host: www.google.fr
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:24.0) Gecko/20100101 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive



HTTP/1.1 302 Found

Location: <https://www.google.fr/>
Cache-Control: private
//...

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="https://www.google.fr/">here</A>.
</BODY></HTML>
```



Structure de la réponse

- ▶ Un code d'état, par exemple :
 - ▶ 200 : la page est dispo
 - ▶ 302 : la ressource a été déplacée
 - ▶ 403 : accès interdit
 - ▶ 404 : ressource non trouvée
- ▶ Les informations supplémentaires :
 - ▶ ex : date de validité, cookies, encodage des caractères, nouvelle adresse
- ▶ La réponse à la requête (si GET)
 - ▶ la page HTML, une image, un feuille de style CSS, un script, un fichier, ...



Les Architecture n-tiers

Les différentes activités d'une application

- ▶ ***De façon logique***, une application se divise en 3 grandes activités :
 - ▶ la **présentation** des informations : Ensemble des traitements permettant l'affichage (éventuellement la conversion) des informations pour les présenter à l'utilisateur
 - ▶ IHM utilisateur (souvent interface graphique)
 - ▶ le **traitement** des informations (la logique métier) : Ensemble des traitements qui rendent le service à l'utilisateur
 - ▶ ex : la génération d'une facture
 - ▶ la **persistance** des informations : Enregistrement des données de façon durable
 - ▶ un fichier (binaire, textuel, XML, ...)
 - ▶ une base de donnée plus ou moins complexes :
 - simple, avec ou sans redondance, distribuée,

Présentation

Logique métier

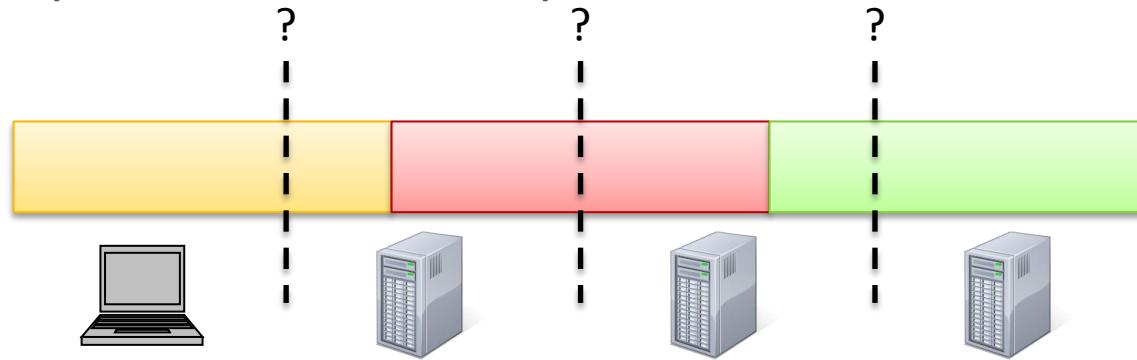
Persistante

Répartition client/serveur

- ▶ A l'implémentation, ce découpage logique donne lieu à plusieurs implémentations/répartitions possibles. Les activités doivent être réparties :
 - ▶ entre le client et le(s) serveur(s)
 - ▶ à l'intérieur du client et du(des) serveur(s)
- ▶ Cette division permet donc de ***séparer les préoccupations***
- ▶ Chaque partie coopèrent pour former l'application. Ainsi :
 - ▶ il est plus facile d'implémenter et faire évoluer le code
 - ▶ il est possible d'avoir plusieurs présentations pour une même logique
 - ▶ il est possible de varier et d'adapter la capacité de chaque parties aux besoins.

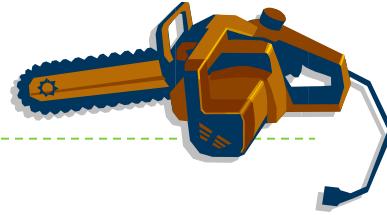
Notion de tiers

- ▶ Chaque partie/couche réalisant une activité déterminée et cohérente est appelée ***tiers***.
- ▶ Dans une architecture distribuée :
 - ▶ les tiers sont souvent exécutés sur des machines différentes
 - ▶ les tiers peuvent être découplés

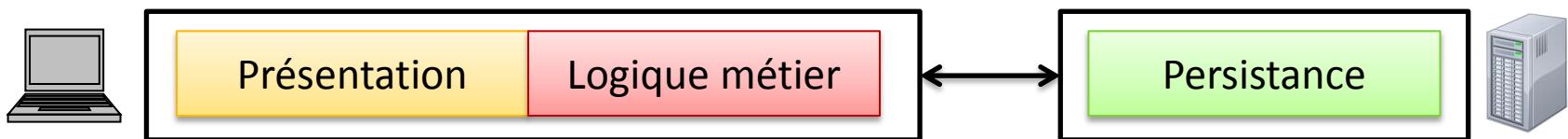


- ▶ Il existe de nombreuses variantes d'organisations des tiers:
 - ▶ 2-tiers, 3-tiers, n-tiers

Architecture 2-tiers

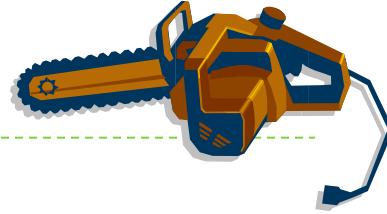


- ▶ On sépare en deux gros morceaux entre le client et le serveur. En général, le client s'occupe de la présentation et le serveur de la persistance.
- ▶ Il a plusieurs façons de répartir la logique applicative :
- ▶ Coté client :

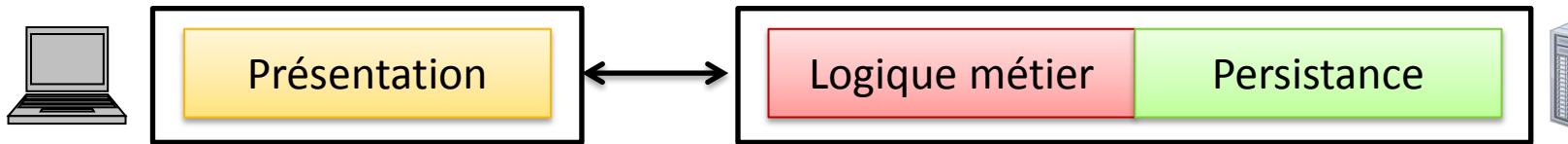


- ▶ client = présentation + applicatif
- ▶ le serveur se charge uniquement de la persistance des données
- ▶ traitement de textes avec stockage distant
- ▶ accès à une base de données distantes

Architecture 2-tiers

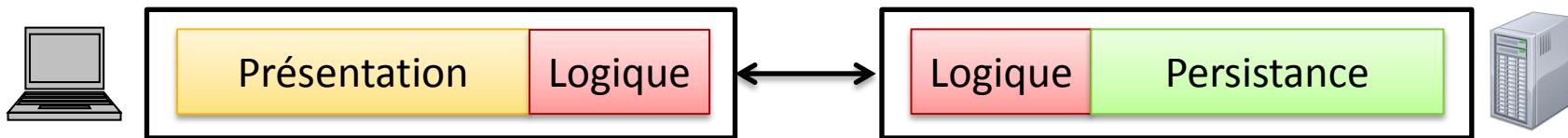


► Coté serveur :



- ▶ Le client est stupide, il ne fait qu'afficher l'interface
 - ▶ peu coûteux, facile à faire évoluer
- ▶ le serveur réalise l'essentiel des opérations

► Mixte :



- ▶ Le client réalise une partie des opérations liées aux traitements
 - ▶ améliore la latence, la réactivité, la sécurité, l'usage du réseau
- ▶ En fonction de la répartition, on parle de clients lourds, riches ou légers

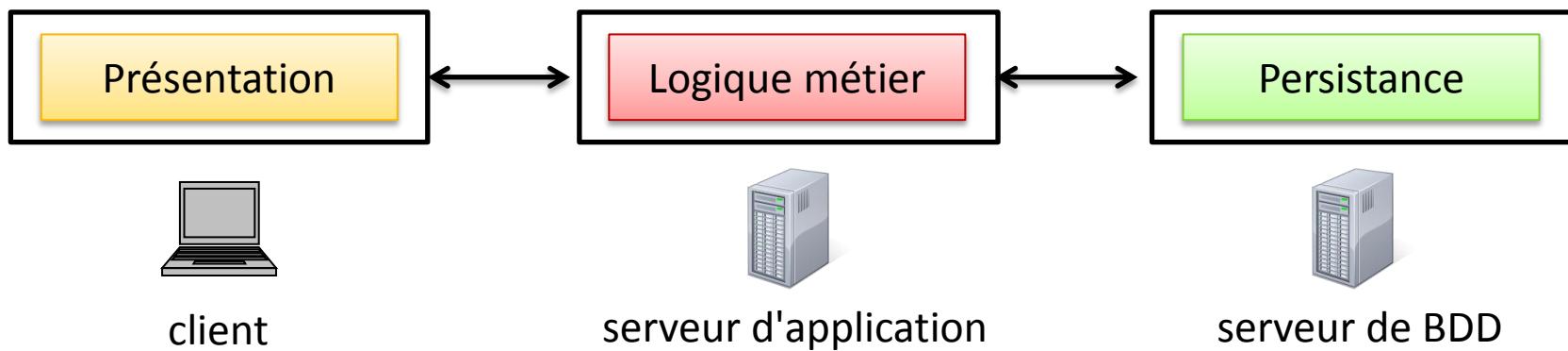
Qui exécute quoi dans le monde du Web?

- ▶ Donc le client et le serveur peuvent exécuter des trucs ...
 - ▶ modifier la présentation
 - ▶ faire évoluer le contenu dynamiquement
 - ▶ ou faire des traitements
- ▶ Du côté serveur plusieurs possibilités :
 - ▶ scripts générant du HTML : les CGI, **les servlets**, ...
 - ▶ les scripts embarqués dans le HTML : **les JSP**, le PHP, ...
- ▶ Du côté client (navigateur):
 - ▶ du code compilé: flash, applet JAVA avec une VM pour les faire tourner
 - ▶ du script avec parfois un JIT : **JavaScript**, dart, ...
- ▶ Si client lourd, alors VM classique ou code classique qui appelle des services Web par exemple

Architecture 3-tiers

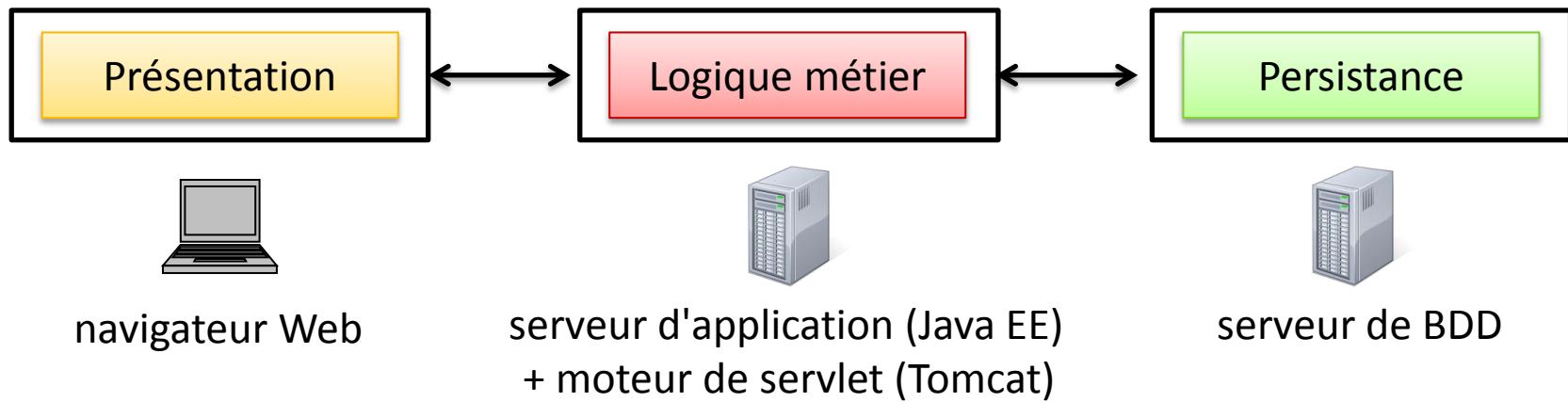


- ▶ Pour plus de flexibilité, on sépare la logique de la persistance
 - ▶ la **présentation** par le client
 - ▶ le **traitement** par un serveur d'application
 - ▶ la **persistance** par un serveur de base de données



Exemple pour le Web

- ▶ Architecture très courante du Web :



- ▶ On va voir plus loin qu'on peut encore découpler la logique applicative du moteur de servlet

Architecture n-tiers

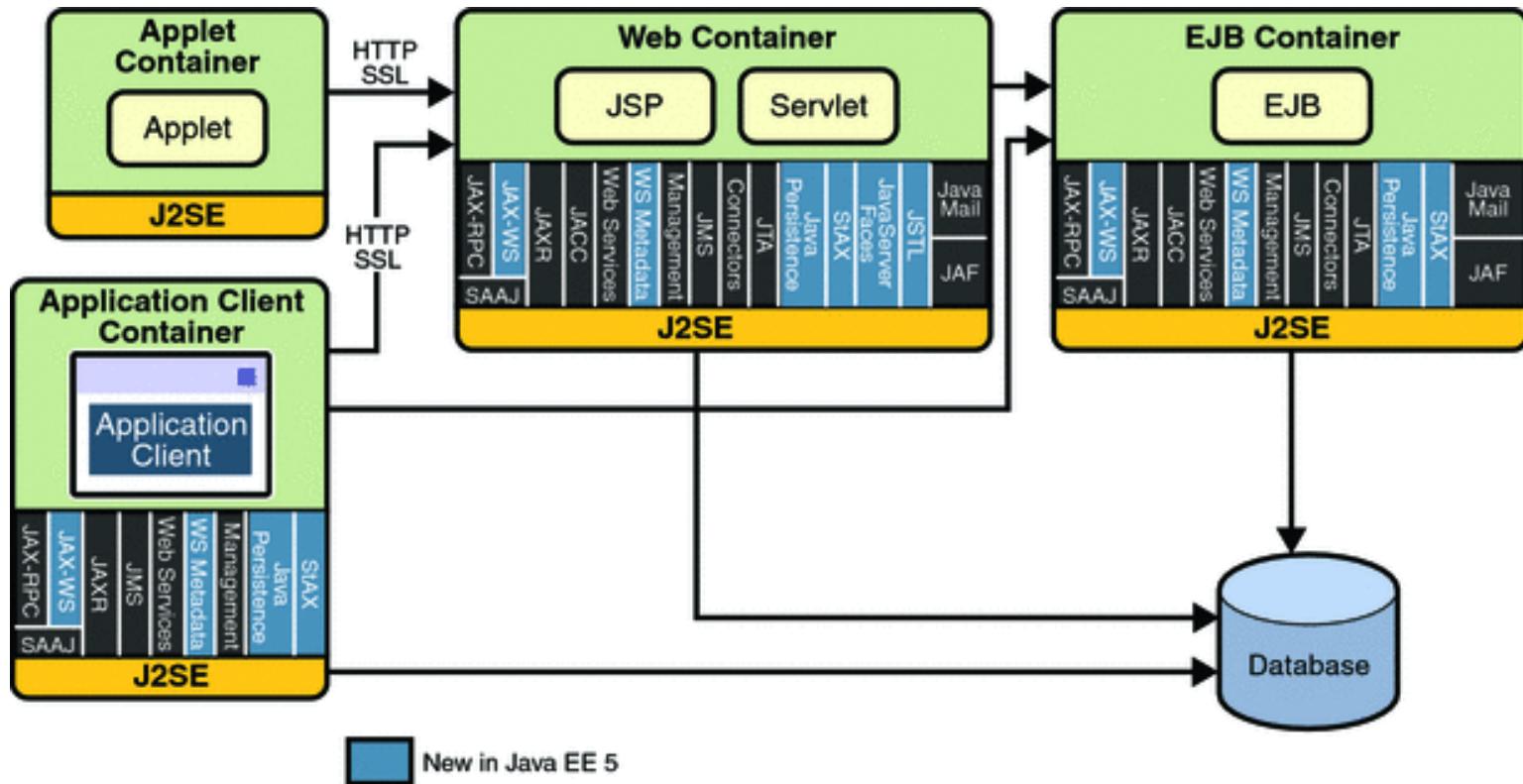
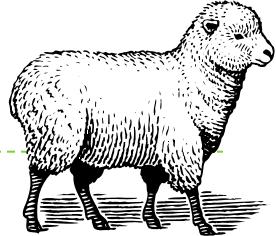


- ▶ On rajoute encore des couches ...
- ▶ Exemple, la couche application peut être subdivisé encore:
 - ▶ composition verticale (service orthogonaux) :
 - ▶ services de transaction, sécurité, ...
 - ▶ composition horizontale (services métiers) :
 - ▶ si l'application s'appuie sur différents traitements
 - prise des commandes, facturations, ...
- ▶ Avantage : découplage fort et bonne séparation des préoccupations => développement, maintenance et évolution plus facile
- ▶ Inconvénients : hétérogénéité des technos à gérer, maintenance/administration répartie



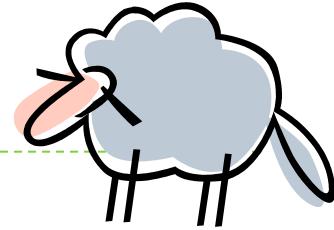
- ▶ Avant J2EE, maintenant Java-EE, version 6
- ▶ Framework proposé par Oracle pour permettre la mise-en-œuvre de ce type d'architecture
- ▶ Mécanismes fournis :
 - ▶ **présentation** : composants web (les servlets et JSP)
 - ▶ **métier** : composants logiciels (EJB)
 - ▶ **persistence** : gestion de données (JDBC, JPA)
 - ▶ communication : JAVA-RMI, IIOP, JMS, ...
 - ▶ annuaire : JNDI
 - ▶ transactions : JTA
- ▶ Serveurs d'applications libres populaires : GlassFish, JBoss, Jonas ...
- ▶ Alternative hors JAVA : framework PHP, Ruby on rails, .NET, ...

Dessine moi un Java EE :

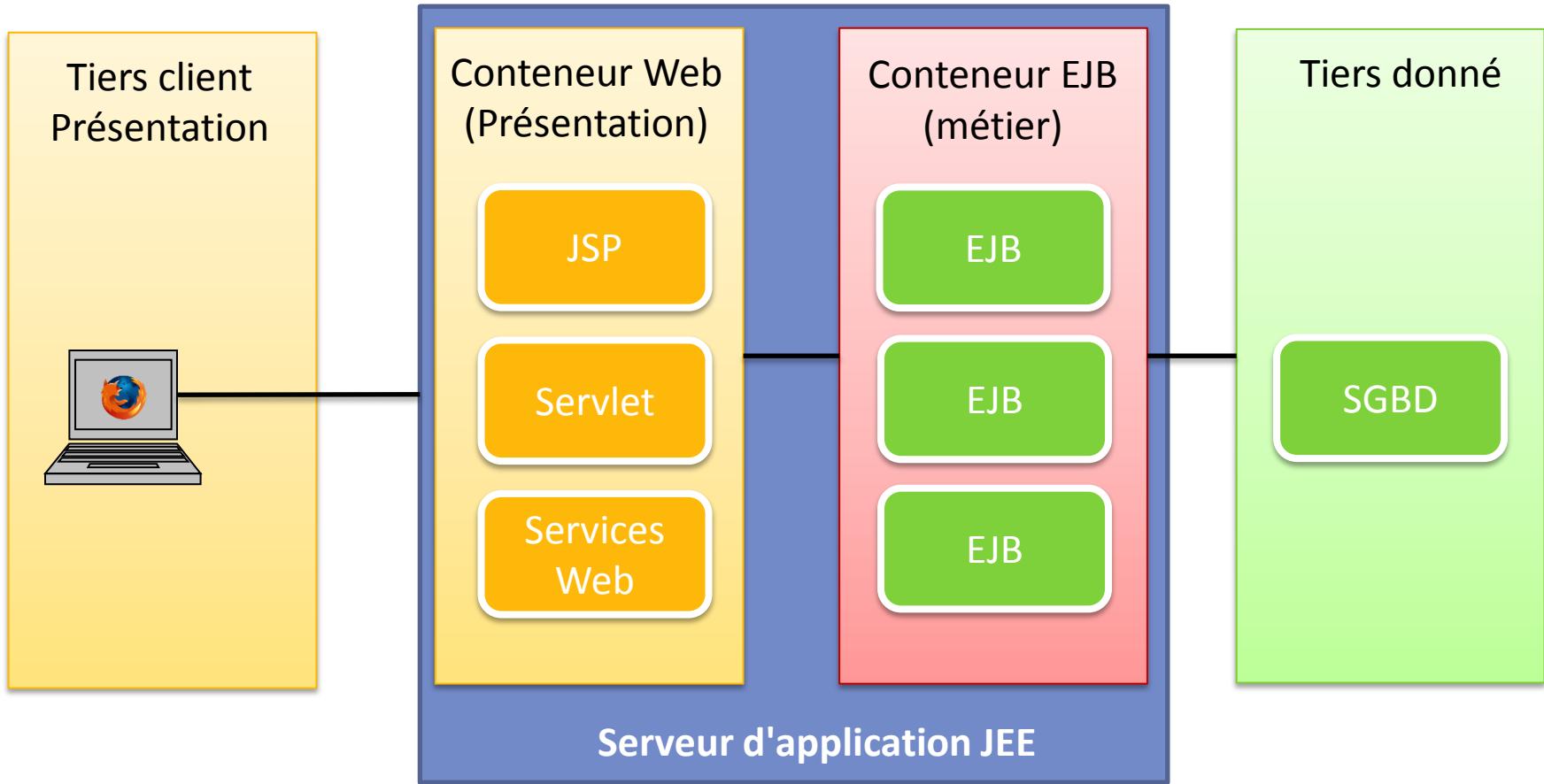


<http://docs.oracle.com/javaee/5/tutorial/doc/bnacj.html>

Avec moins de gros mots ...



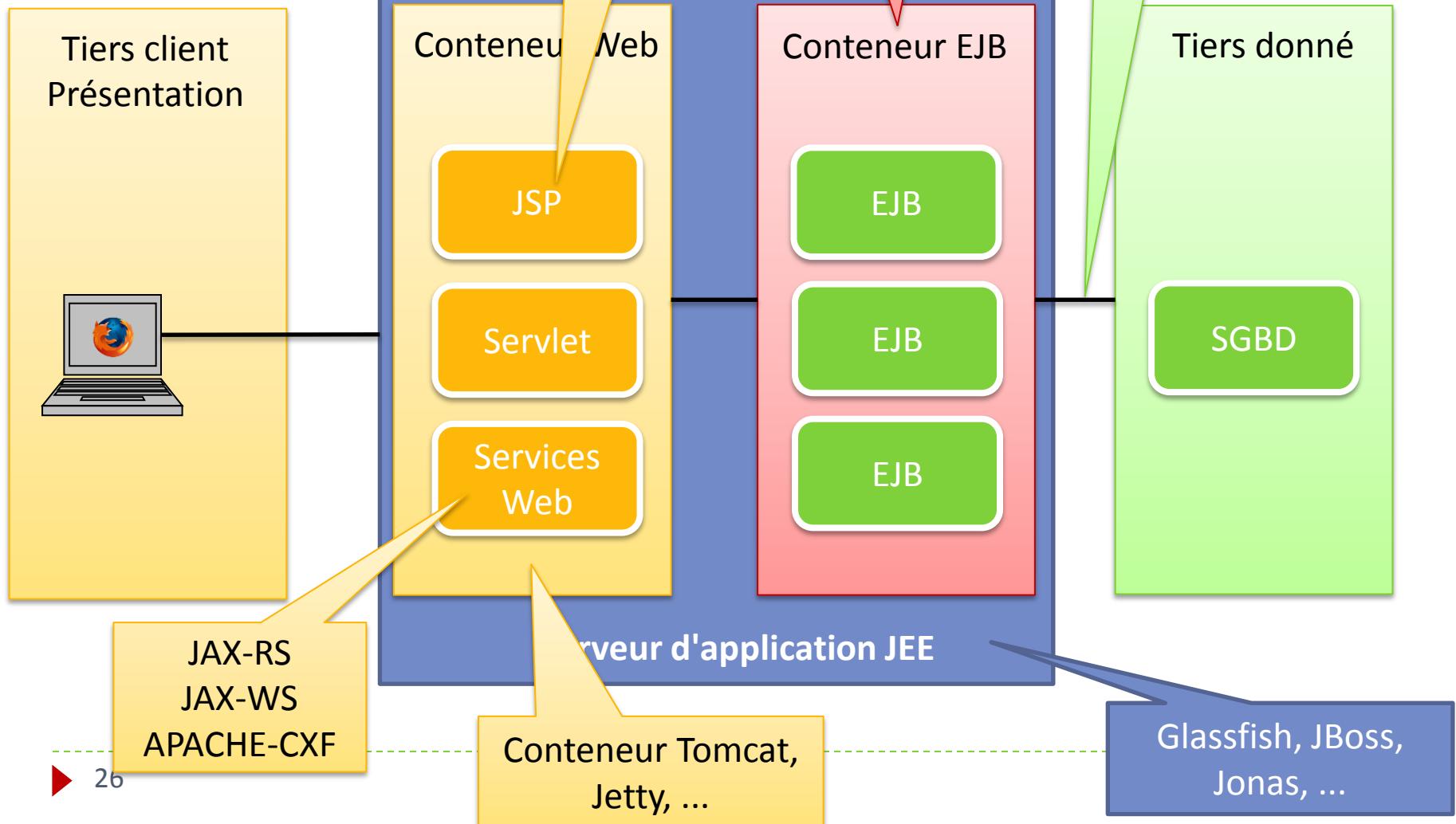
► 4 Tiers



Avec les gros mots ...

Java Persistant Entities
Message Driven Beans

► 4 Tiers



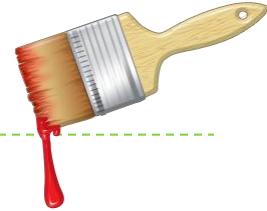
Partie présentation des applications Java EE

- ▶ Servlet
 - ▶ Filter
 - ▶ JSP (Java Server Page)
 - ▶ JSTL (Java Server Tag Library)
 - ▶ JSF (Java Server Face)
-
- ▶ fichiers statiques (images, css, scripts)
 - ▶ et les fichiers bibliothèques (JSON, ,...)



Les pages statiques (HTML et CSS)

HTML, CSS, JS, le principe



- ▶ **HTML** : le fond, structure du document
- ▶ **CSS** : la forme, comment on doit afficher les éléments
- ▶ **JavaScript** : le dynamisme et le contrôles par le client
- ▶ JS n'est pas JAVA (mais tout le monde commence à le savoir ...)



Le HTML



- ▶ Standard du W3C
- ▶ Différentes versions HTML 1-5, XHTML, ...

```
<!DOCTYPE html>
```

indique la version de HTML utilisée, ici
HTML 5

```
<html>
```

Ouverture de la page HTML

```
<head>  
  <!-- Ici l'en-tête de page -->  
</head>
```

En-tête

```
<body>  
  <!-- Ici le corps de page -->  
</body>
```

Corps (contenu)

```
</html>
```





- ▶ Donne le titre de la page et descriptions:
 - ▶ balise title, balises meta
- ▶ Et de fournir directement la description des styles ou du javascript
 - ▶ balises style et script

```
<head>
  <title>Cours MRI</title>
  <meta name="author" lang="fr" content="YM">
  <meta name="keywords" content="MRI">
  <meta name="robots" content=".....">

  <style>
    body { background: navy; color: yellow; }
  </style>
  <script type="text/javascript">
    //<![CDATA[...code javascript...//]]>
  </script>
</head>
```

Titre de la page

Description

Information pour les robots

Code CSS dans une balise style

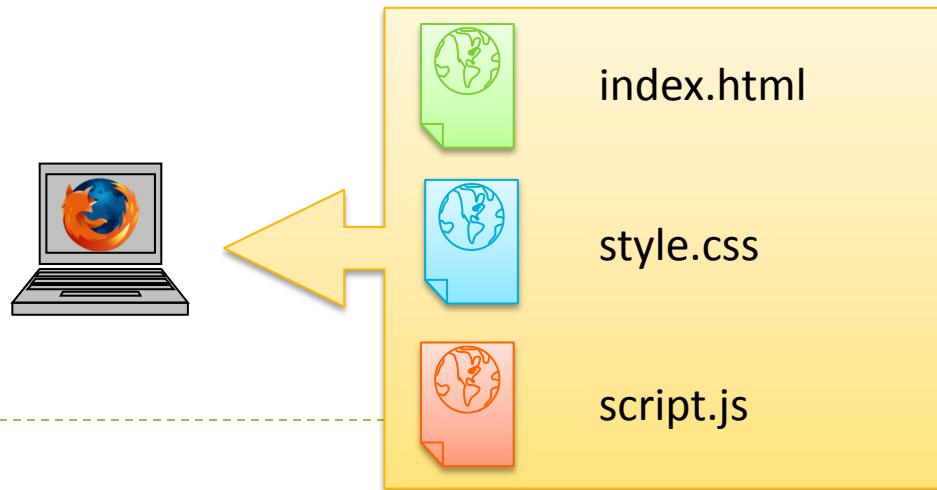
Code JS dans une balise script

On ne mélange pas forme et contenu

- ▶ On met un lien vers les fichiers JS et CSS

- ▶ plus facile à maintenir et réutiliser
- ▶ sépare bien les préoccupations

```
<head>
...
<link rel="stylesheet" type="text/css" href="style.css">
<link rel="javascript" type="text/javascript" href="script.js">
```



Corps



- ▶ Balises de titres : h1, h2, ... h6
 - ▶ Balises de contenu blocs : div
 - ▶ Balises de contenu : p, span
 - ▶ Balises de mise en relief : strong, em
 - ▶ Balises de liens : a
 - ▶ Pour les images : img
 - ▶ Pour faire des listes : ul (ou ol) puis li
 - ▶ Balises de formulaire : form, input, textarea
 - ▶ Balises pour les tableaux : table, th, tr, td,
-
- ▶ **Remarques :**
 - ▶ On évite d'utiliser les balises b (bold), i (italic), font, ... qui lient trop solidement la forme et le fond
 - ▶ On donne la forme des balises en utilisant une feuille de style :
 - ▶ ou plus spécifiquement en utilisant les attributs class et id

Exemple de pages ...



```
<!DOCTYPE html>
<html>
<head>
<title>Cours MRI</title>
<meta name="author" lang="fr" content="YM">
<meta name="keywords" content="MRI">
<meta name="robots" content=".....">
<link rel="stylesheet" type="text/css" href="style.css">
<link rel="javascript" type="text/javascript" href="script.js">
</head>
<body>
<h1> Ma super Page </h1>
<h2>Mon super sous-titre ! </h2>
<p> Et je donne des liens : </p>
<ul>
<li><a href="https://jaxb.java.net/tutorial/">tutoriel JAXB</a>
<li><a href="http://w3c.org">j'aime le W3C</a>
</ul>
</body>
</html>
```

Résultat : pas top

A screenshot of a web browser window titled "Cours MRI". The address bar shows "localhost:8080/mri.servlet.example/index.html". The page content is as follows:

Ma super Page

Mon super sous-titre !

Et je donne des liens :

- [tutoriel JAXB](#)
- [j'aime le W3C](#)

Tevr. 16, 2014 10:30:09 PM org.apache.coyote.AbstractProtocol start

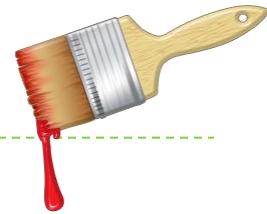
Les attributs utiles



- ▶ id : permet d'identifier un élément de façon unique
- ▶ class : permet d'identifier une classe d'éléments
- ▶ Permet de mettre un style ou de retrouver les balises avec JS

```
<body>
  <h1 class="ugly">Ma super Page </h1>
  <h2 class="ugly">Mon super sous-titre ! </h2>
  <p> Et je donne des liens : </p>
  <ul>
    <li id="jaxb"><a href="https://jaxb.java.net/tutorial/">tutoriel JAXB</a>
    <li><a href="http://w3c.org">j'aime le W3C</a>
  </ul>
```

Exemple de feuille de style (css)



- ▶ Fichier CSS, définit la forme
- ▶ Fonctionne par héritage du style du dessus

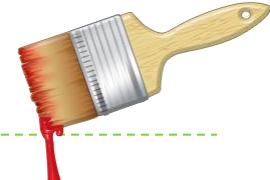
```
@CHARSET "ISO-8859-1";  
  
body{  
color : red;  
background : yellow;  
}  
  
#jaxb {  
background-color : pink;  
}  
  
.ugly{  
color : brown;  
}
```

redéfini le style de la balise body

redéfini le style de la balise d'id jaxb (on préfixe par #)

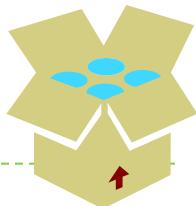
redéfini le style de la balise de la classe ugly (on préfixe par .)

Résultat : pire

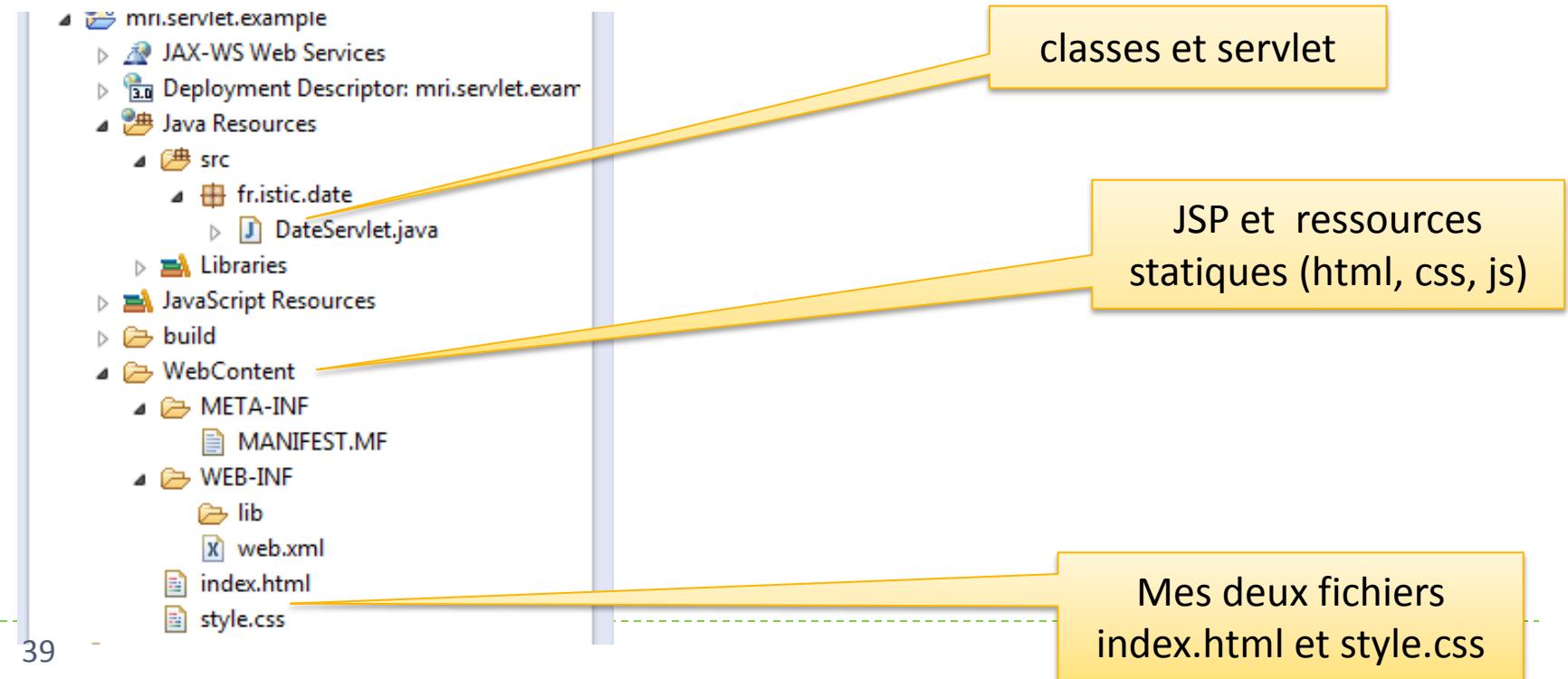


- ▶ C'est pour ça qu'on s'adresse à des graphistes et ergonomes pour ce qui est de la forme

Packaging des applications Web



- ▶ Les applications Web sont packagés dans un jar particulier **le war**
- ▶ Il est paramétré par le fichier `web.xml` (voir plus loin)
- ▶ Dans Eclipse :



Saines lectures

- ▶ Spécification HTML 5 :
 - ▶ <http://www.w3.org/TR/html5/introduction.html>
- ▶ Sur les css :
 - ▶ <http://www.w3.org/Style/CSS/>
 - ▶ <http://www.csszengarden.com/>
- ▶ Sur les préprocesseurs CSS
 - ▶ <http://sass-lang.com/>
 - ▶ <http://lesscss.org/>
- ▶ Généralistes :
 - ▶ <http://www.pompage.net/>
 - ▶ <http://tympanus.net/codrops/>



Les pages dynamiques (Servlets)

Pourquoi des pages dynamiques ?

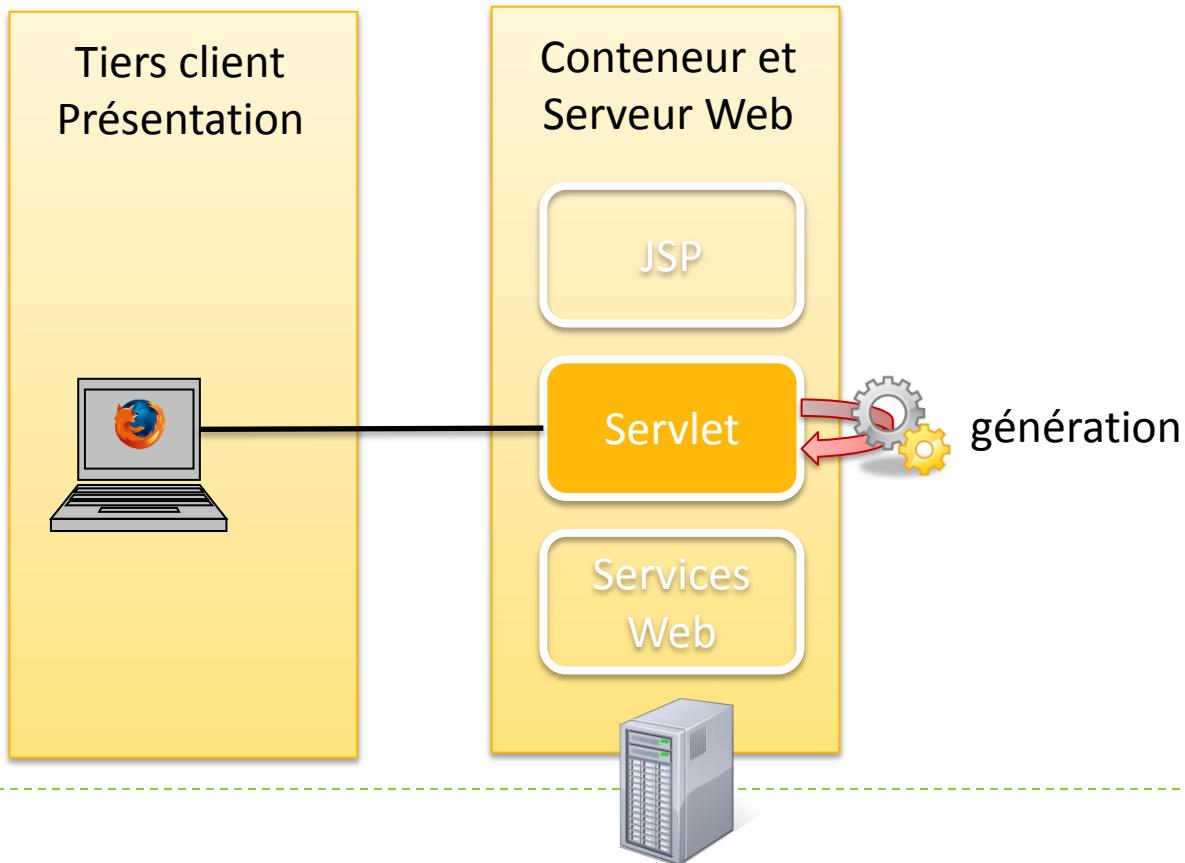
- ▶ Protocole HTTP (toujours)
- ▶ Les pages sont dynamiques pour
 - ▶ présenter des informations changeantes
 - ▶ météo, catalogue de produit avec leur prix, ...
 - ▶ adapter les pages à la demande du client
 - ▶ outils de recherche, contrôle d'accès, ...
- ▶ Comment ?
 1. identification du client ou du contexte lors de la demande d'une ressource (ex : page HTML)
 2. génération de la page en fonction de ce contexte
 3. persistance (ex base de donnée) côté serveur et cookies côté client pour une prochaine consultation

JavaScript vs (CGI vs Servlet)

- ▶ **JavaScript :**
 - ▶ langage historiquement exécuté côté client (devient faux avec node.js)
 - ▶ le script doit être envoyé au client par le serveur ou être présent chez le client
 - ▶ utilise des requêtes vers le serveur pour avoir des données (Ajax, Websockets ...)
- ▶ **Applet Java :**
 - ▶ code Java exécuté par le navigateur du client (de plus en plus rare)
- ▶ **CGI :**
 - ▶ exécuté côté serveur
 - ▶ programmes compilés (C, ...) spécifiques à la plateforme donc moins portables
 - ▶ un processus dédié par CGI
- ▶ **Servlet**
 - ▶ exécuté côté serveur dans un conteneur Web
 - ▶ un objet java qui rend des services (ex : génération de la page Web)
 - ▶ le conteneur Web et la JVM gère l'affectation des ressources (Threads)

La servlet

- ▶ Une classe qui génère le code HTML (ou n'importe quel ressource CSS, JS, ...)
- ▶ Exécutée côté serveur par le conteneur Web



La servlet

- ▶ Les servlets sont des sous-classes de l'interface Servlet du package javax.servlet
- ▶ On peut directement étendre GenericServlet mais il faut alors gérer le protocole
- ▶ Plusieurs types en fonction de la nature du retour :
 - ▶ FileServlet, CGIServlet, ...
 - ▶ HttpServlet, SIPServlet, ...
- ▶ HttpServlet implémente le protocole HTTP et est donc fortement recommandé pour notre usage.

Exemple de servlet minimaliste

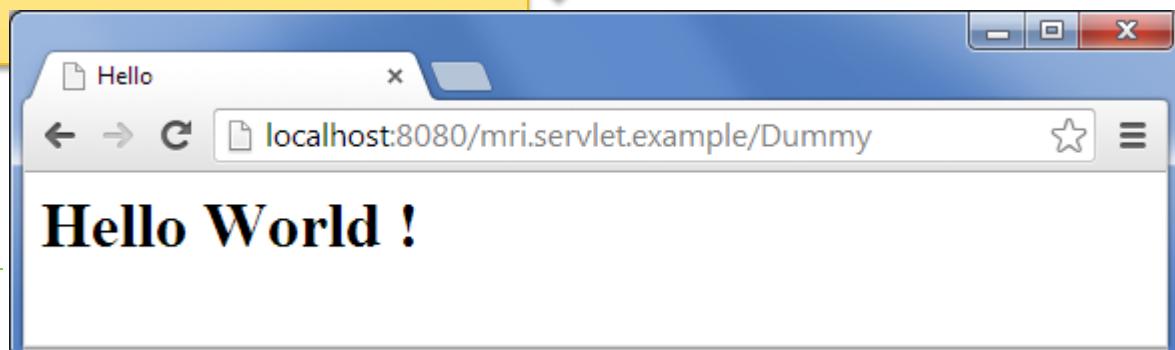
```
package fr.istic.dumb;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/Dummy")
public class DummyServlet extends HttpServlet {

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



- ▶ Pas de méthode main et **on ne doit pas utiliser le constructeur** pour initialiser la servlet.
- ▶ La servlet est créée par le conteneur **et son cycle de vie géré par lui.**
- ▶ Deux méthodes importantes :
 - ▶ **init(ServletConfig)** est appelée par à la création de la servlet et permet d'initialiser la servlet (équivalent constructeur)
 - ▶ **destroy()** est appelé à la destruction
 - ▶ Ces deux méthodes sont très utiles pour récupérer et stocker l'état de la servlet

Servlet, exécution

Servlet



génération

- ▶ Le client envoie GET, POST, DELETE, HEAD en HTTP
- ▶ La servlet possède une méthode **service**. Cette méthode redirige vers les bonnes méthodes doXXX :
 - ▶ doGet quand on reçoit un GET
 - ▶ doPost quand on reçoit un post
 - ▶ doHead
 - ▶ Note : on déconseille de réécrire service() mais d'écrire le code dans les méthodes doXXX()
- ▶ Autres méthodes :
 - ▶ getLastModified permet de dire quand a eu lieu le dernier changement dans la page
 - ▶ getServletName, getServletInfo, getServletContext pour obtenir les informations sur la servlet

Cycle de vie de la servlet

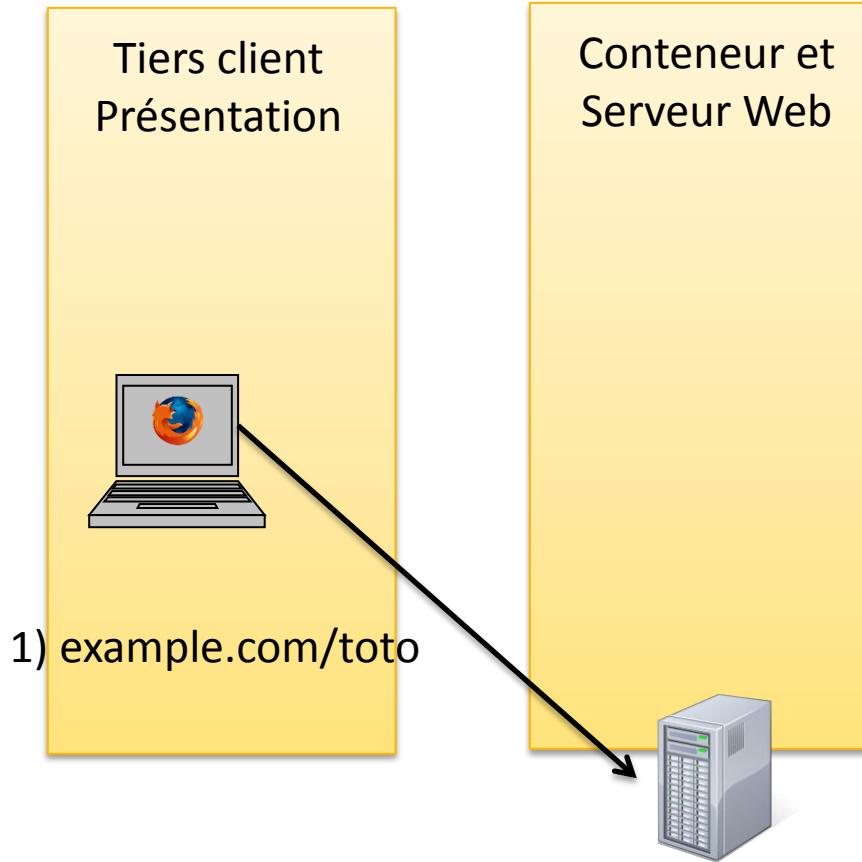
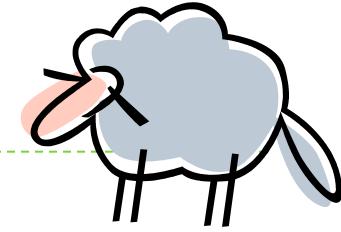
Servlet



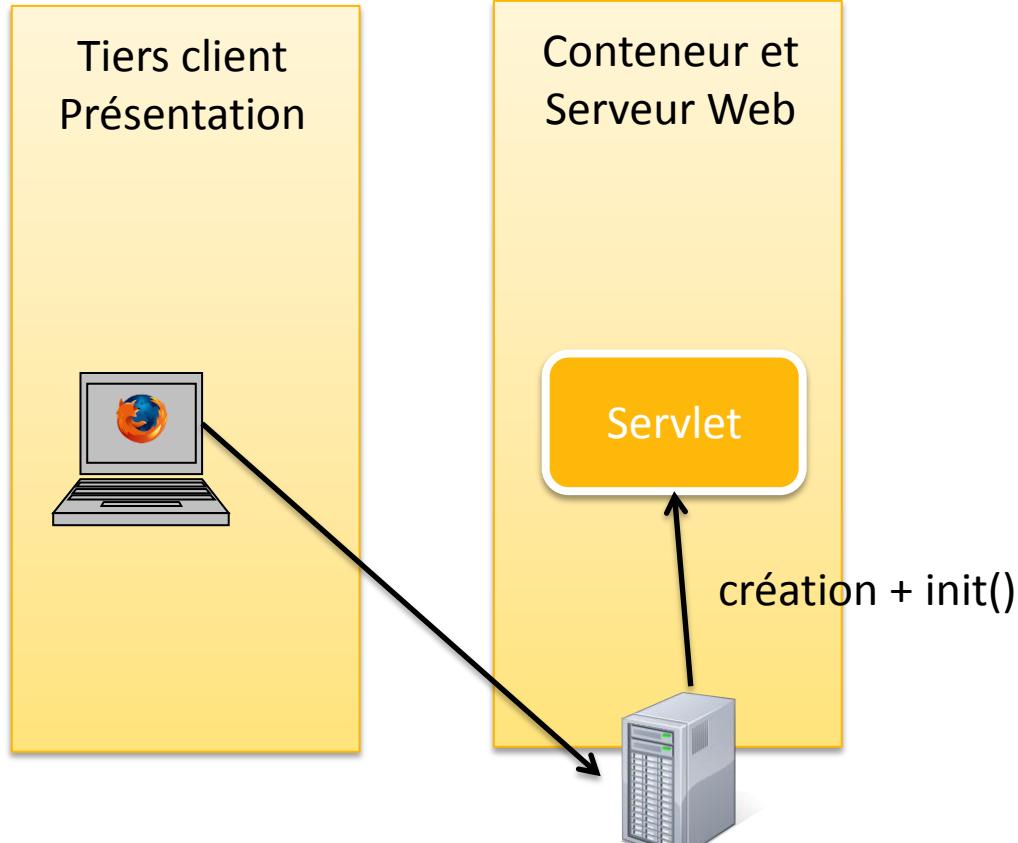
génération

1. Un client fait une première requête vers le serveur
2. Ensuite soit le conteneur Web :
 - ▶ trouve la Servlet et passe à la suite
 - ▶ crée la Servlet (elle n'existe pas) et appelle la méthode init
3. La méthode service est appelée et redirige vers la bonne méthode doXXX. Pour doGet et doPost :
 1. analyse des paramètres de la requête
 2. génération de la page
 3. retour au client
4. Le conteneur décide quand détruire la servlet :
 - ▶ par exemple si elle n'est plus utilisée depuis un moment

Cycle de vie en dessin

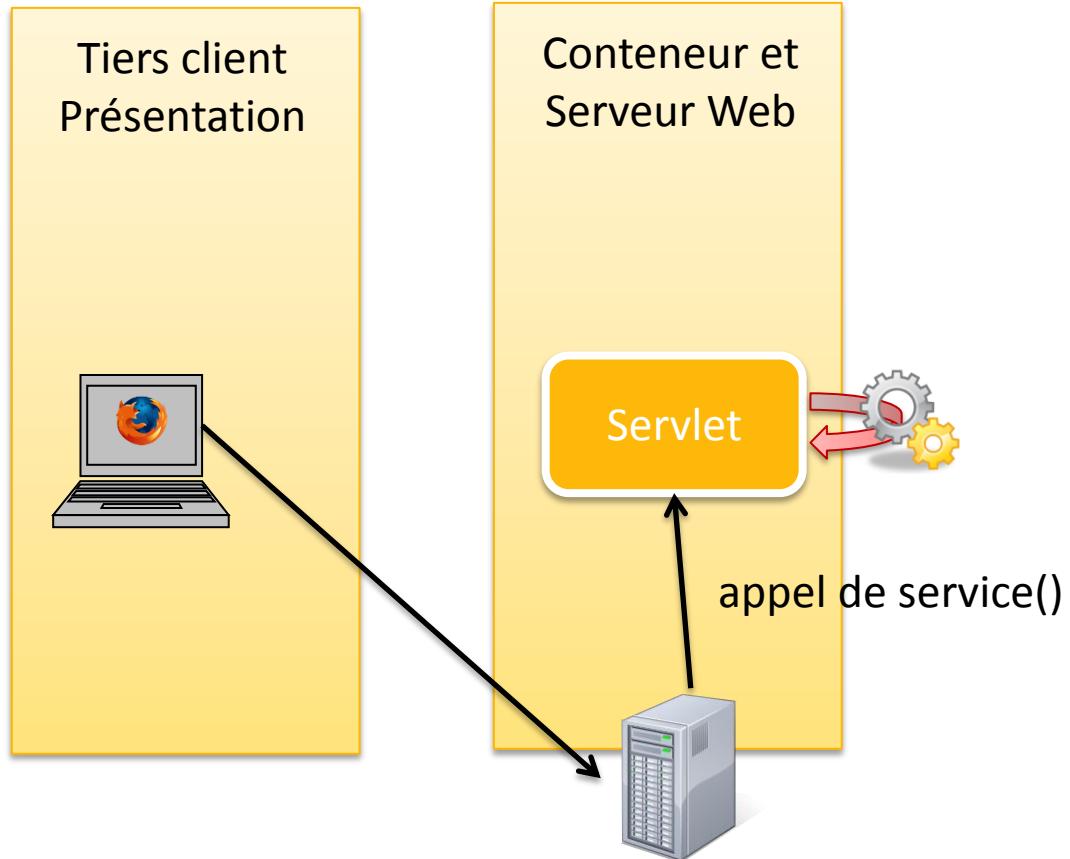


Cycle de vie en dessin

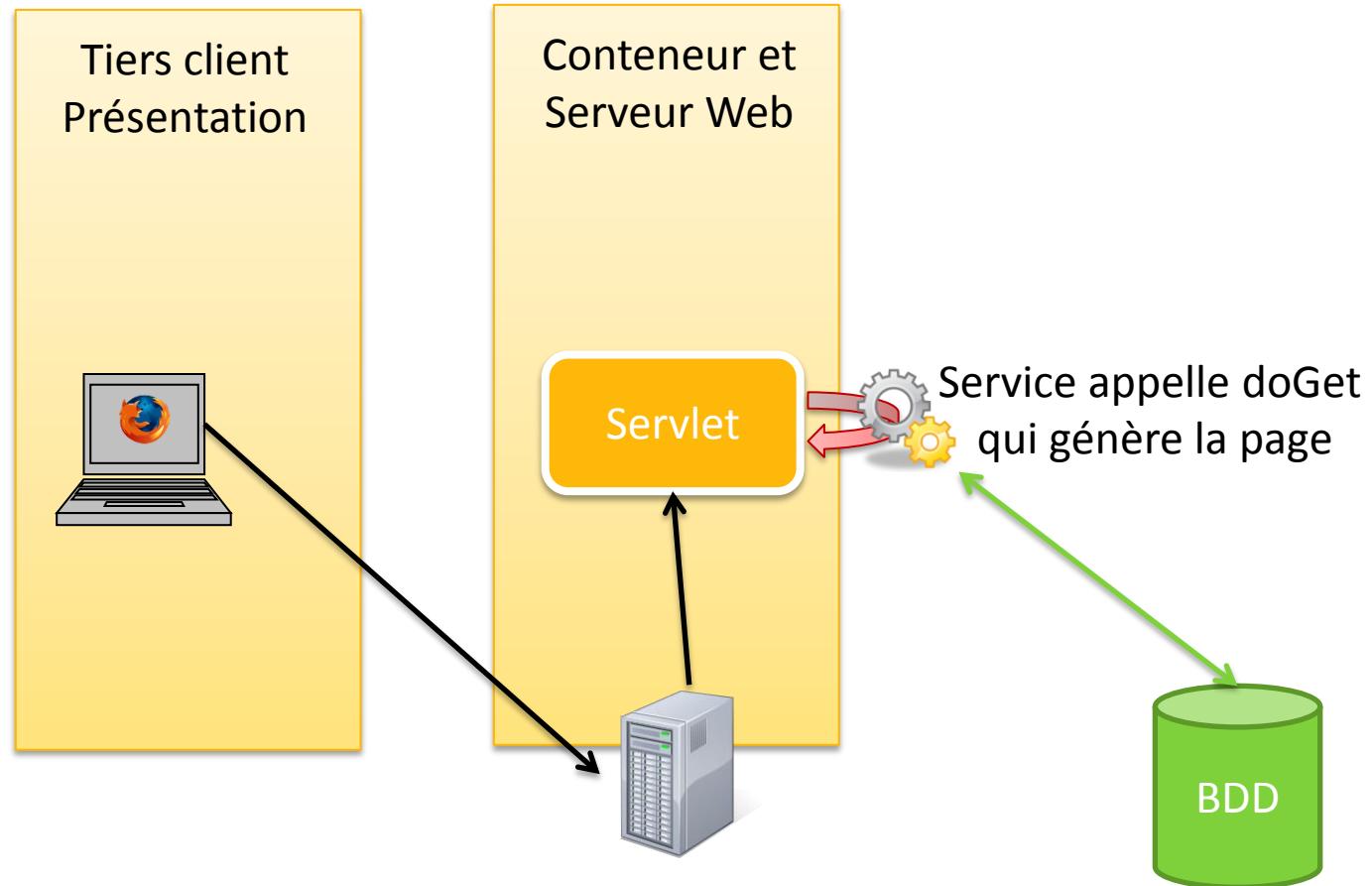


Qui est "toto" ?
Classe ServletToto, pas d'instances

Cycle de vie en dessin

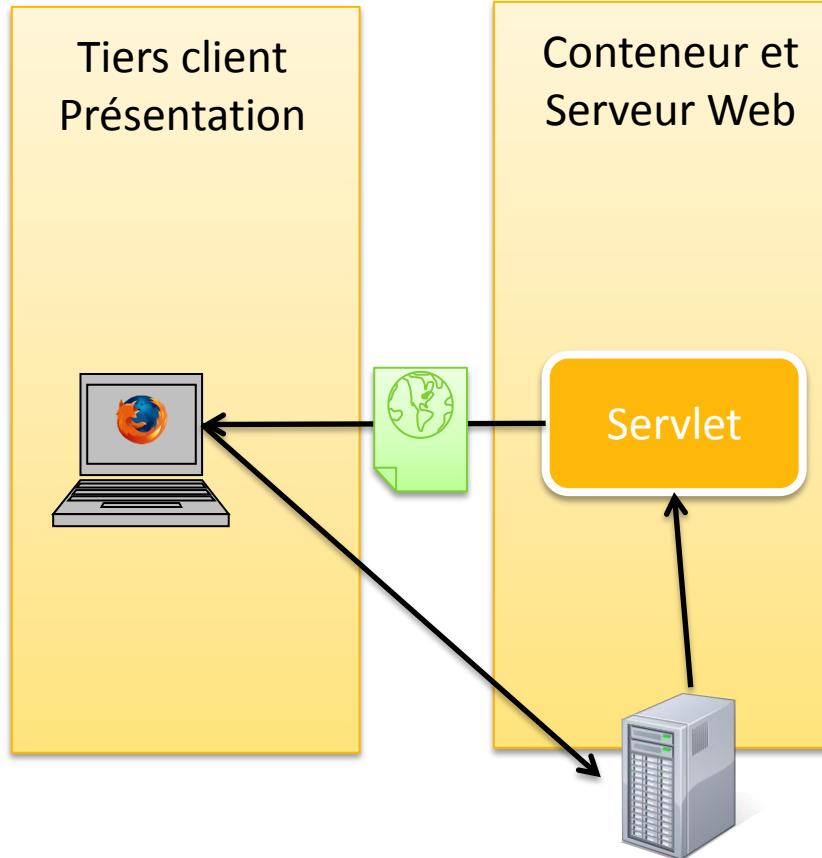


Cycle de vie en dessin

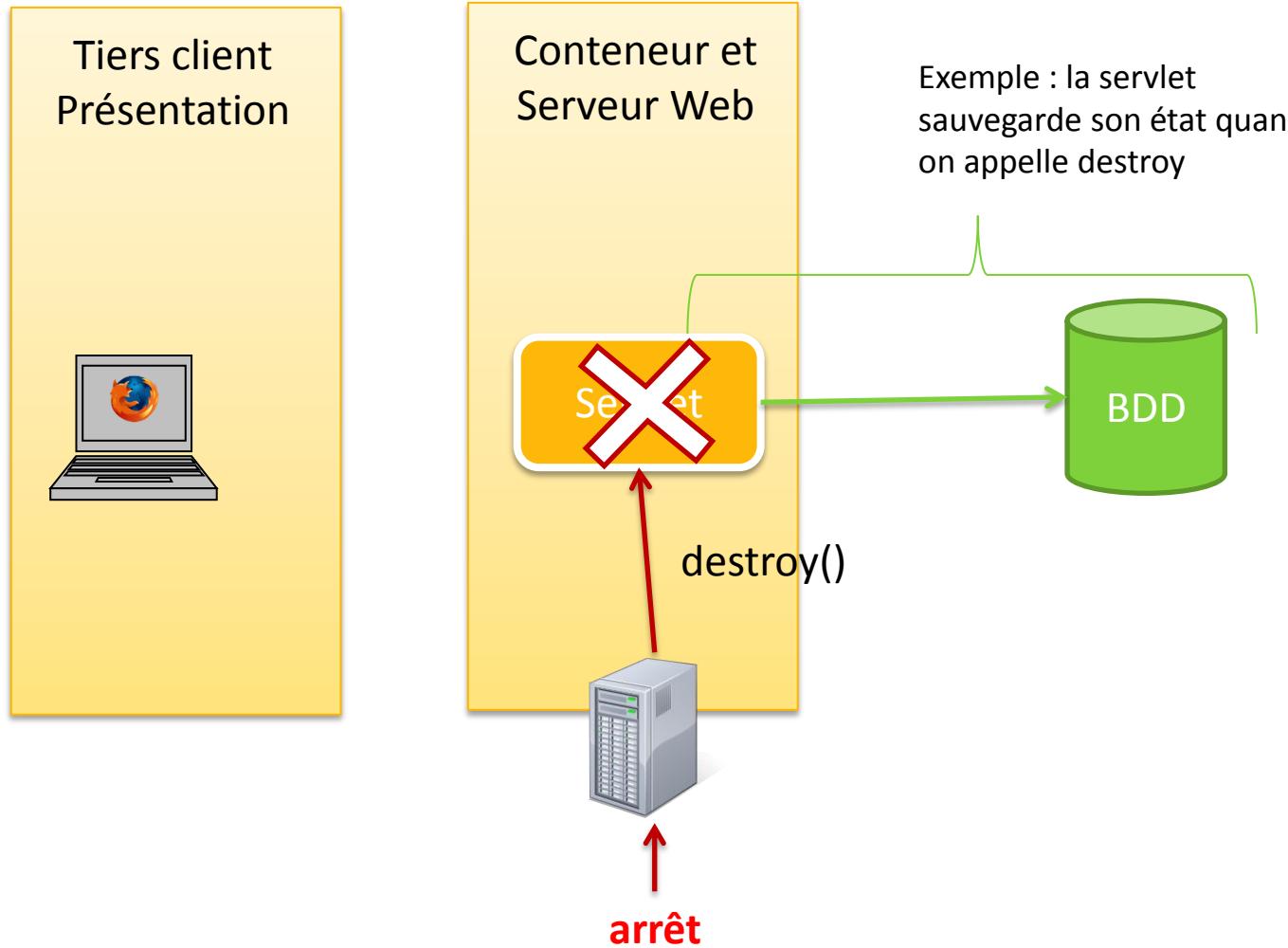


Exemple : L'usage d'une BDD est très fréquente pour générer la page en fonction d'un historique

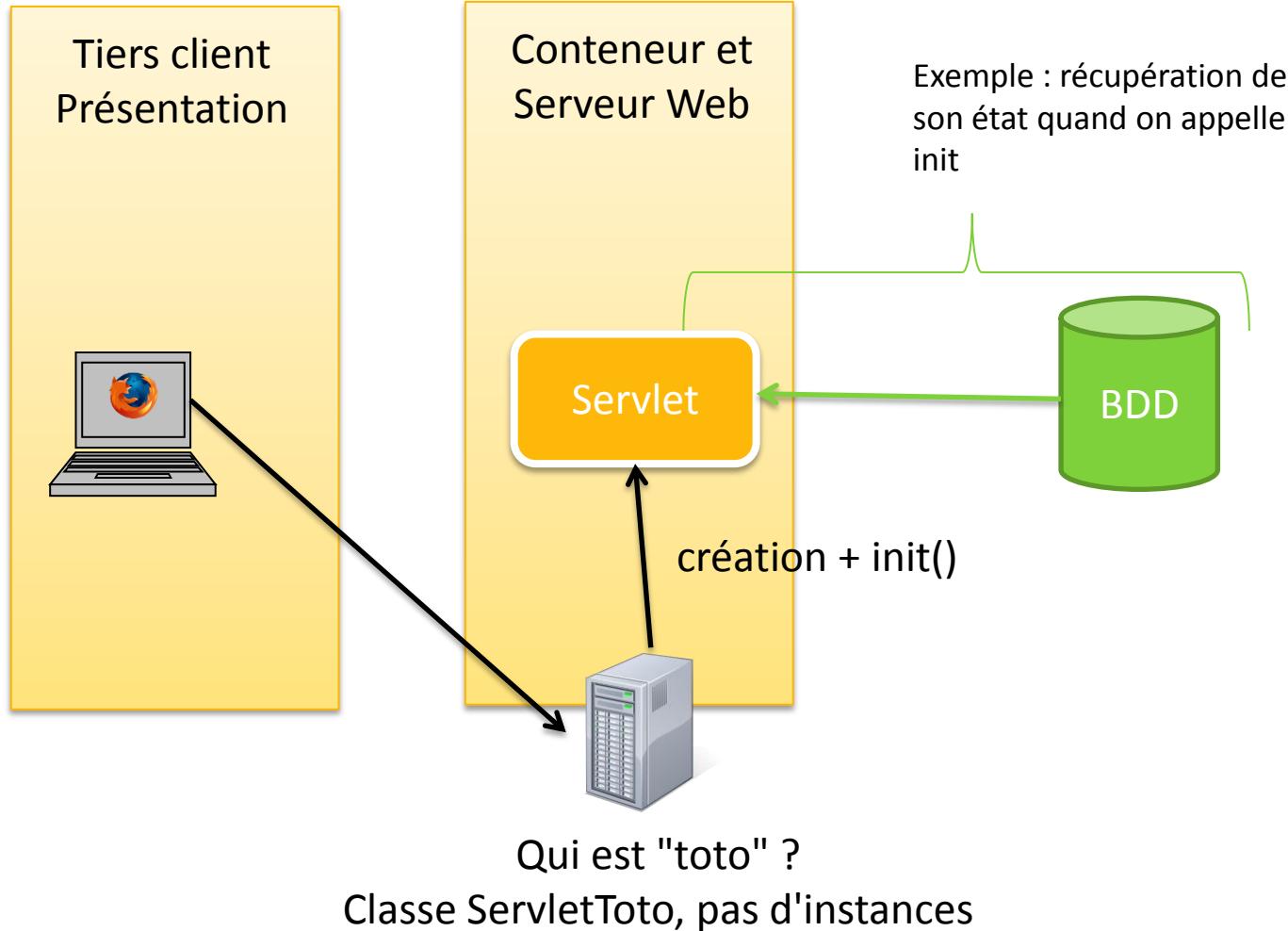
Cycle de vie en dessin



Plus tard (ex arrêt serveur)



Plus tard (après redémarrage serveur)





- ▶ Toutes les méthodes doXXX ont les même signatures
- ▶ doGet et doPost sont de loin les plus utilisés
 - ▶ font la même chose, on verra la différence plus loin
 - ▶ retournent une ressource (généralement html)
- ▶ doGet(**HttpServletRequest request**, **HttpServletResponse response**) throws ServletException, IOException
 - ▶ request : contient la description de la requête du client
 - ▶ response : utilisé pour envoyer la réponse au client

La requête **HttpServletRequest** permet

- ▶ de récupérer les valeurs des paramètres de formulaires :

```
String pname = request.getParameter("name"); // Get a parameter
```

- ▶ on peut avoir tous les paramètres :
 - ▶ `request.getParameterMap(); request.getParameterNames();`
- ▶ La valeur par défaut des paramètres est null si le paramètre n'a pas été passé dans la requête :
 - **attention au NullPointerException !**
- ▶ d'avoir un objet session qui permet d'identifier les clients
- ▶ d'obtenir les cookies fournis par le client
- ▶ d'obtenir les informations sur le header

Autres méthodes de **HttpServletRequest**

- ▶ Différentes méthodes pour avoir les infos

The screenshot shows a Firefox browser window with the URL `localhost:8080/mri.servlet.example/InfoServlet?nom=toto&age=14`. Below the browser, a table displays the results of calling various `request` methods:

<code>request.getMethod()</code>	GET
<code>request.getRequestURI()</code>	/mri.servlet.example/InfoServlet
<code>request.getQueryString()</code>	nom=toto&age=14
<code>request.getServletPath()</code>	/InfoServlet
<code>request.getContextPath()</code>	/mri.servlet.example

La réponse **HttpServletResponse**

- ▶ obtenir un writer ou un stream (selon ce qu'on veux écrire) pour retourner la réponse :

```
PrintWriter out = response.getWriter(); // Get the output
out.println("<html>");
out.println("<head><TITLE>Hello, World !</title></head>");
out.println("<body>");
out.println("Hello World !");
out.println("</body></html>");
out.flush();
```

- ▶ on doit déclarer le type de retour :

```
response.setContentType("text/html;charset=UTF-8");
```
- ▶ type MIME : image/gif, audio/mp3, application/pdf, ...
- ▶ on peut écrire des cookies ou créer une session

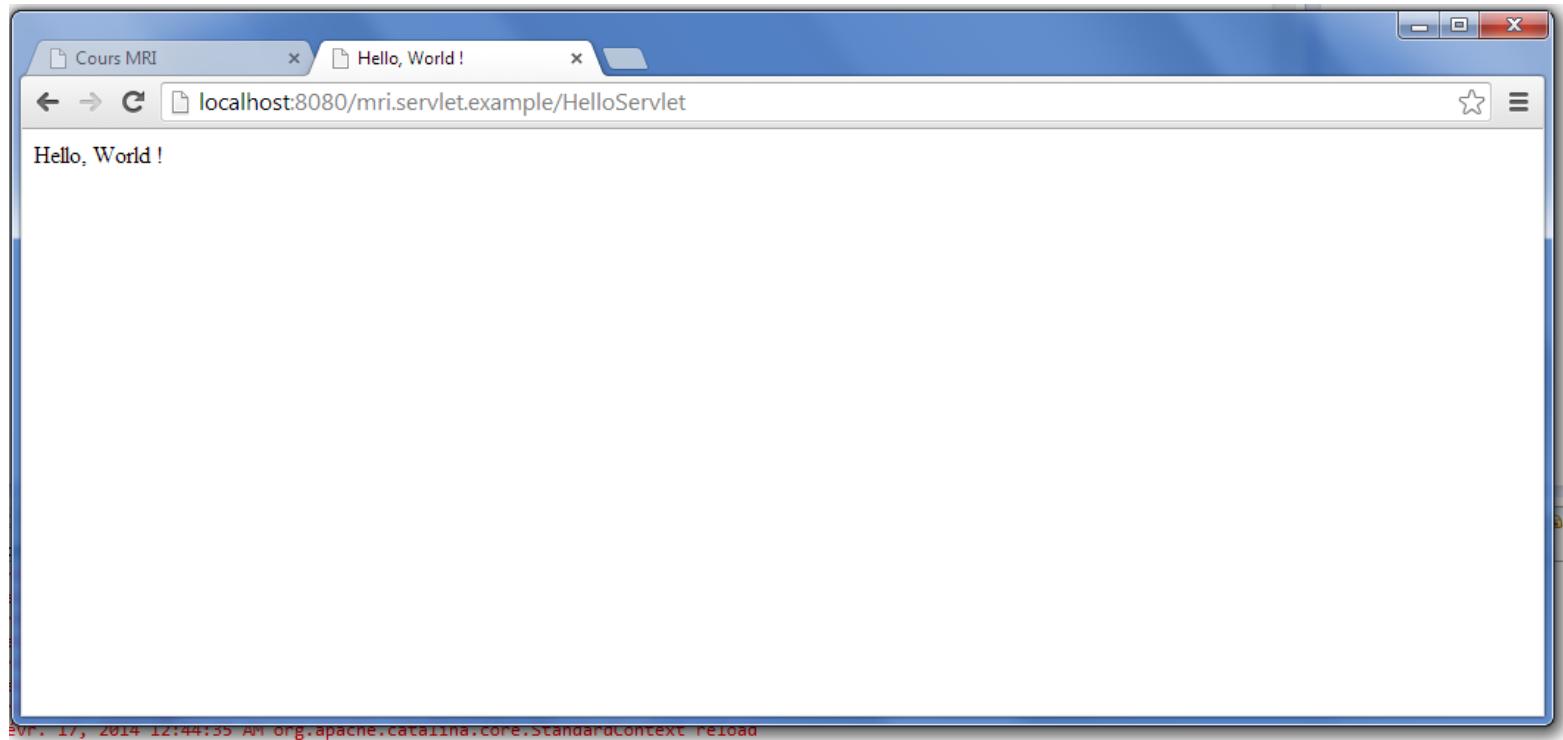
Exemple de servlet

```
@WebServlet("/HelloServlet")
public class BasicHello extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,
IOException {
        response.setContentType("text/html"); // Set the Content-Type header
        PrintWriter out = response.getWriter(); // Get the output

        String pname = request.getParameter("name"); // Get a parameter
        if (pname == null) {
            pname = "World !";
        }
        out.println("<html>");
        out.println("<head><TITLE>Hello, " + pname + "</title></head>");
        out.println("<body>");
        out.println("Hello, " + pname);
        out.println("</body></html>");
        out.flush();
    }
}
```

Résultat sans passer de paramètres



Contenu dynamique ...

Initialisation avec init.

```
private int appels;  
  
public void init() throws ServletException {  
    appels = 0;  
    log("création servlet");  
}
```

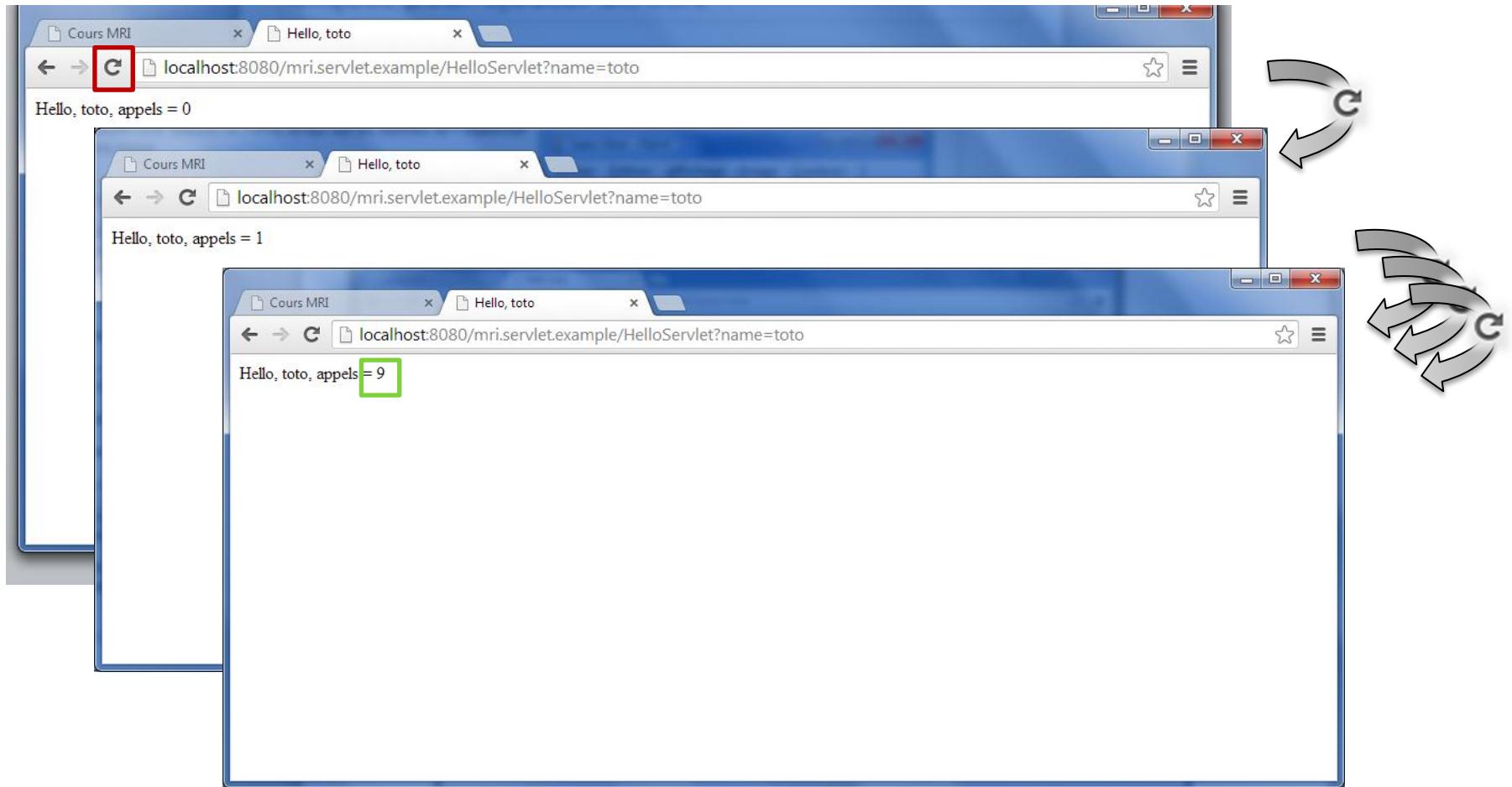
Remarque, **log** permet de logger des messages côté serveur

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    response.setContentType("text/html"); // Set the Content-Type header  
    PrintWriter out = response.getWriter(); // Get the output  
    String pname = request.getParameter("name"); // Get a parameter  
    if (pname == null) {  
        pname = "World !";  
    }  
    out.println("<html>");  
    out.println("<head><TITLE>Hello, " + pname + "</title></head>");  
    out.println("<body>");  
    out.println("Hello, " + pname + ", appels = " + appels);  
    out.println("</body></html>");  
    appels++;  
    out.flush();  
}
```

affiche les appels

incrémente les appels

Affichage après rechargement

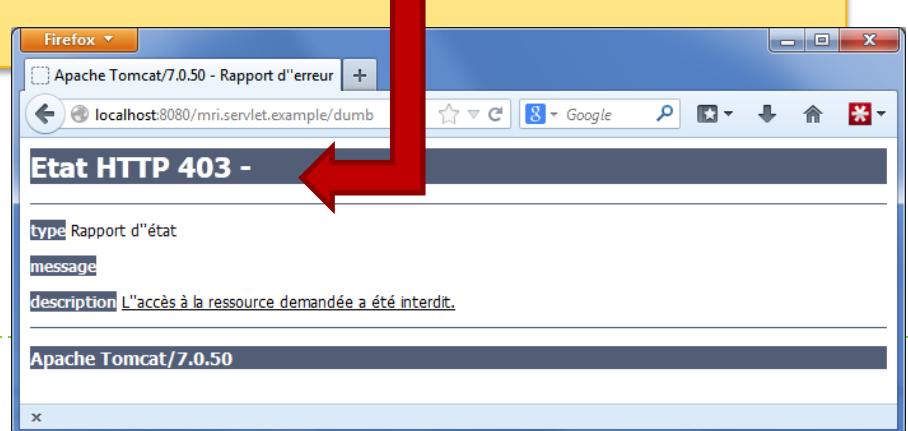


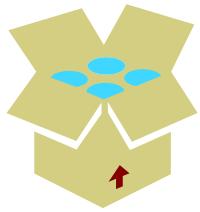
Code d'états

- ▶ On peut envoyer des codes d'états en cas d'exception par exemple

```
@WebServlet("/dumb")
public class UltraDummyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html"); // Set the Content-Type header
        response.sendError(HttpServletRequest.SC_FORBIDDEN);
    }
}
```

Constante pour le code 403

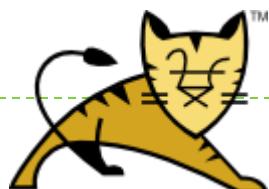




Développement sous Eclipse

Moteur de servlet/conteneur

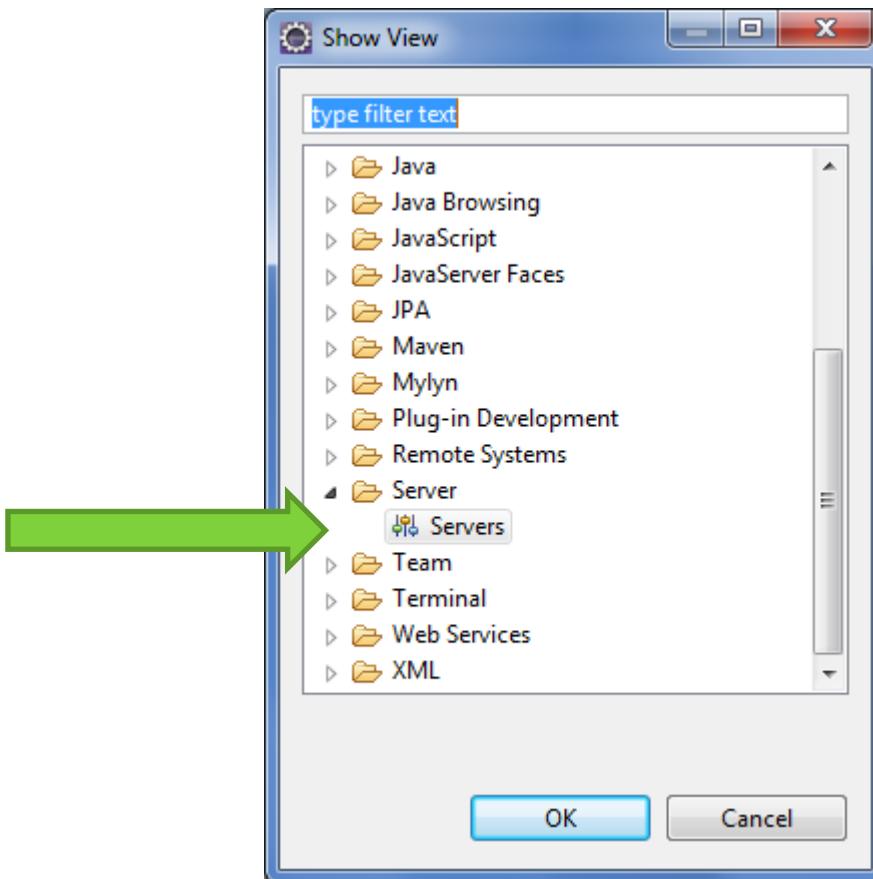
- ▶ Un grand nombre de moteur de servlet/conteneur web disponible soit indépendant, par exemple
 - ▶ Tomcat
 - ▶ Jetty
 - ▶ ...
- ▶ soit intégré au serveur d'application, par exemple dans
 - ▶ glassfish
 - ▶ JBoss
 - ▶
- ▶ Nous utiliserons **Tomcat** et **Eclipse JEE**



Intégration de Tomcat dans Eclipse JEE

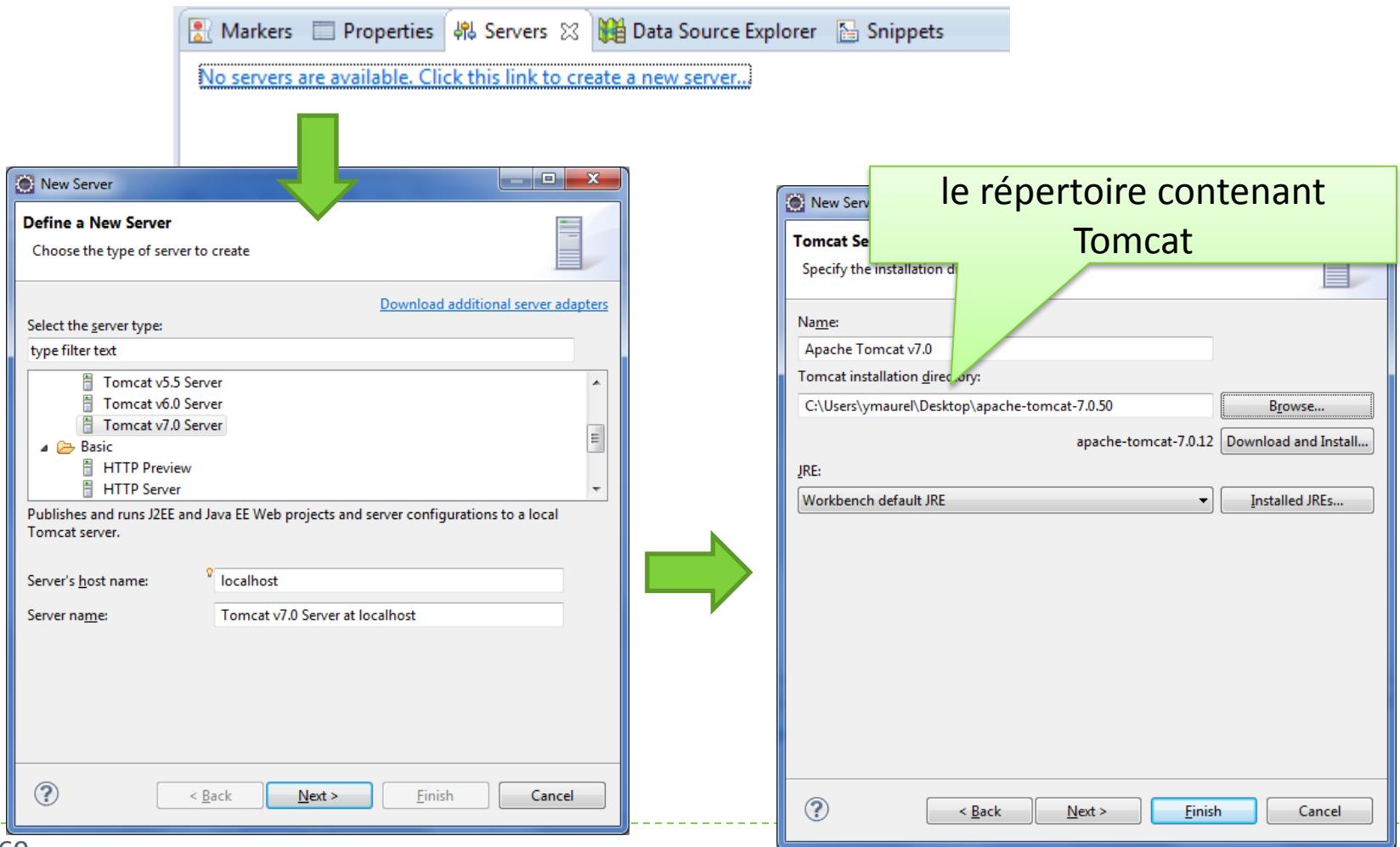
- Eclipse JEE peut gérer le cycle de vie d'un serveur Tomcat et gérer le déploiement

ajouter la vue server



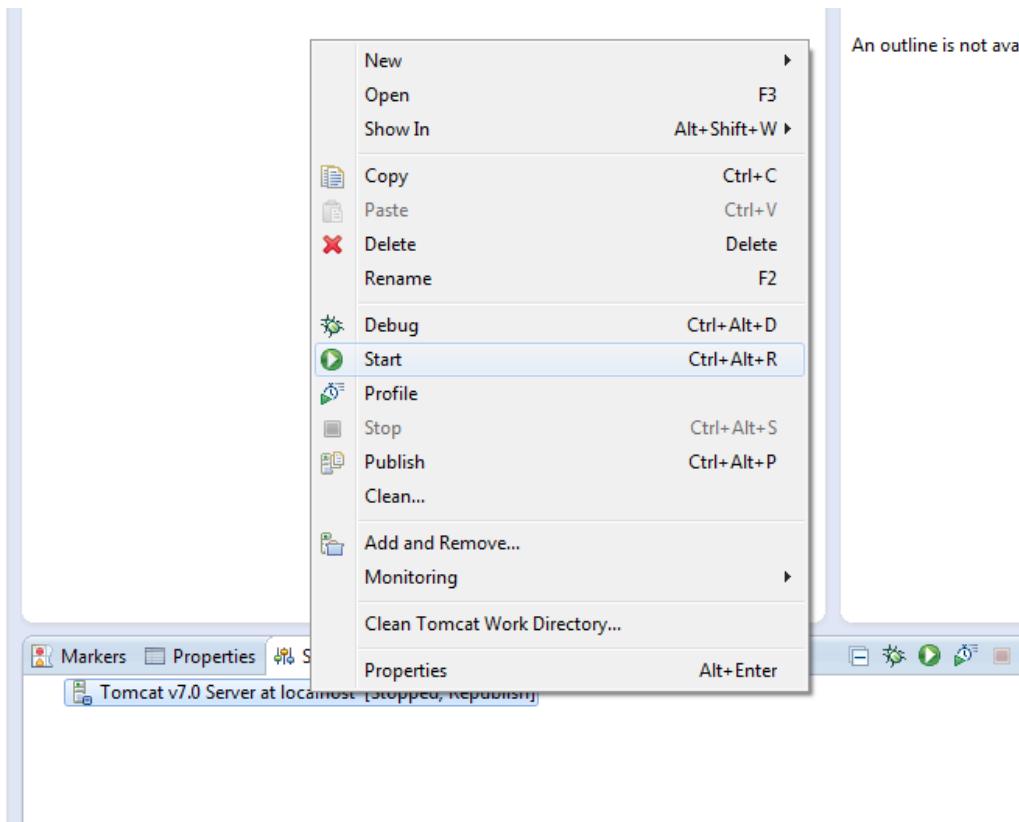
Intégration de Tomcat dans Eclipse JEE

▶ Ajouter un serveur à partir de la vue



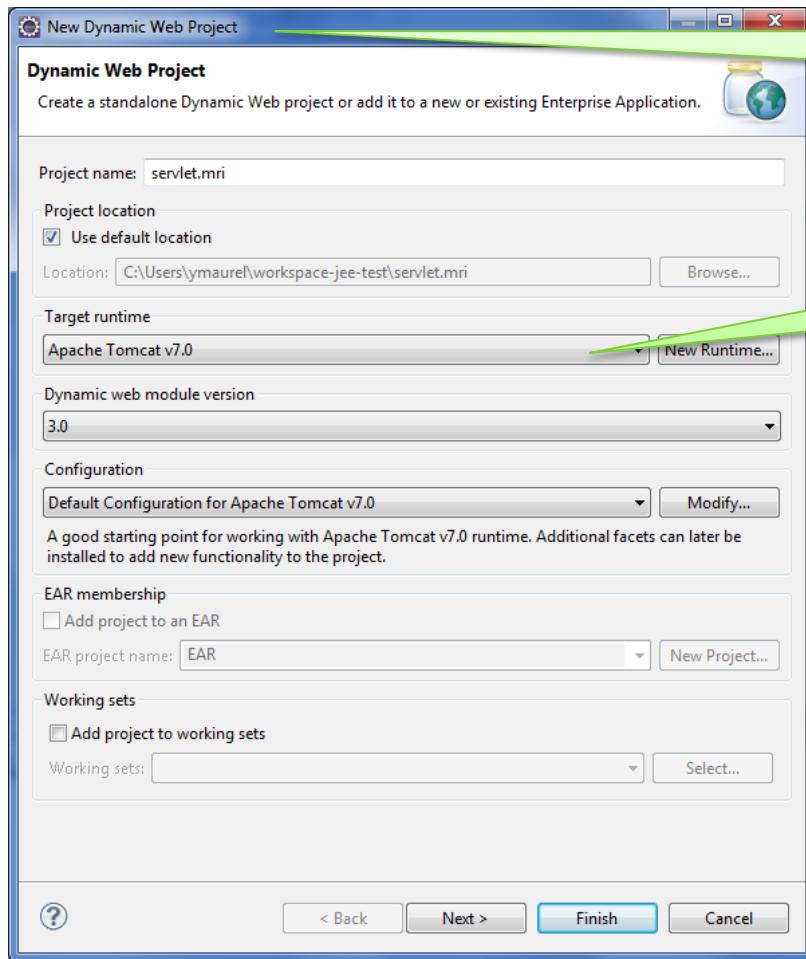
Intégration de Tomcat dans Eclipse JEE

- ▶ La vue serveur permet de gérer le cycle de vie d'un ou plusieurs serveurs



Création d'une application Web

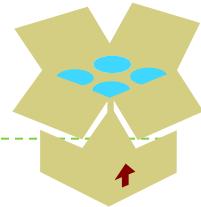
Dans Eclipse JEE, on utilise les Dynamic Web Project



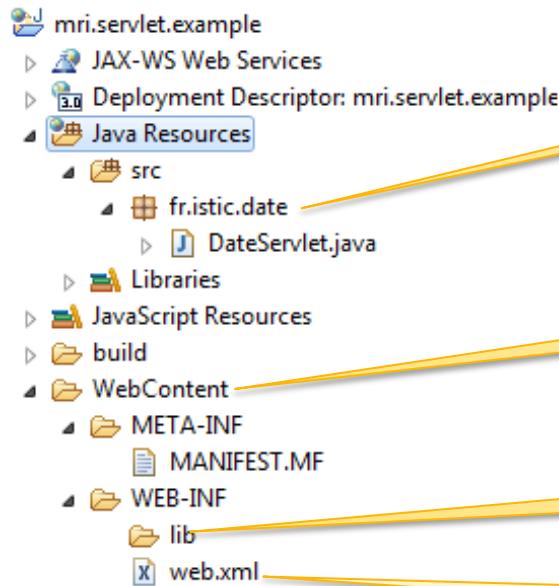
On créé un projet du type
Dynamic Web Project

On indique le serveur sur
lequel déployer le projet

Packaging des applications Web



- ▶ Les applications Web sont packagés dans un jar particulier **le war**
- ▶ Il est paramétré par le fichier **web.xml**
- ▶ Dans Eclipse :



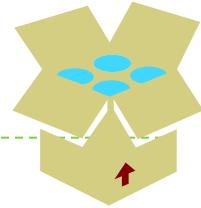
classes et servlet

JSP et ressources statiques (html, css, js)

Dépendances à d'autres librairies (jar)

fichier de configuration web.xml

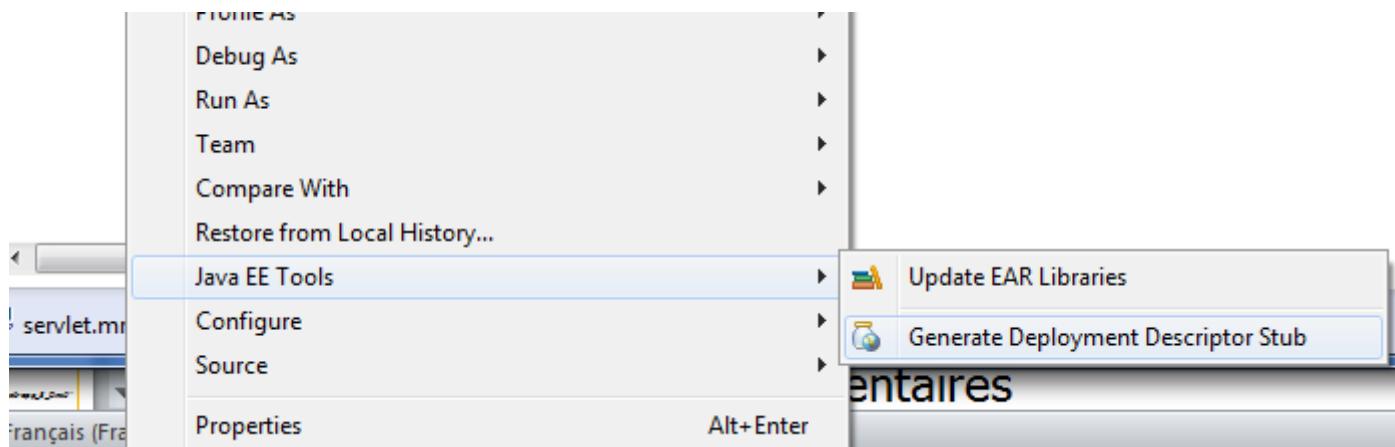
Structure du war (contenu de l'archive)



- ▶ ***.html, *.png, *.jsp, ..., applets.jar, midlets.jar**
- ▶ **WEB-INF/web.xml**
 - ▶ Fichier de déploiement
 - ▶ Paramétrage des servlets, types MIME additionnels, ...
- ▶ **WEB-INF/classes/**
 - ▶ .class des servlets et des classes (JavaBean, ...) associées
 - ▶ ressources additionnelles (localstring.properties, ...)
- ▶ **WEB-INF/lib/**
 - ▶ .jar additionnels provenant de tierce parties (comme des drivers JDBC, TagLib (jsf, ...), ...)
- ▶ **WEB-INF/tlds/**
 - ▶ .tld décrivant les TagLibs (voir JSP plus loin).

Si le fichier web.xml n'existe pas ...

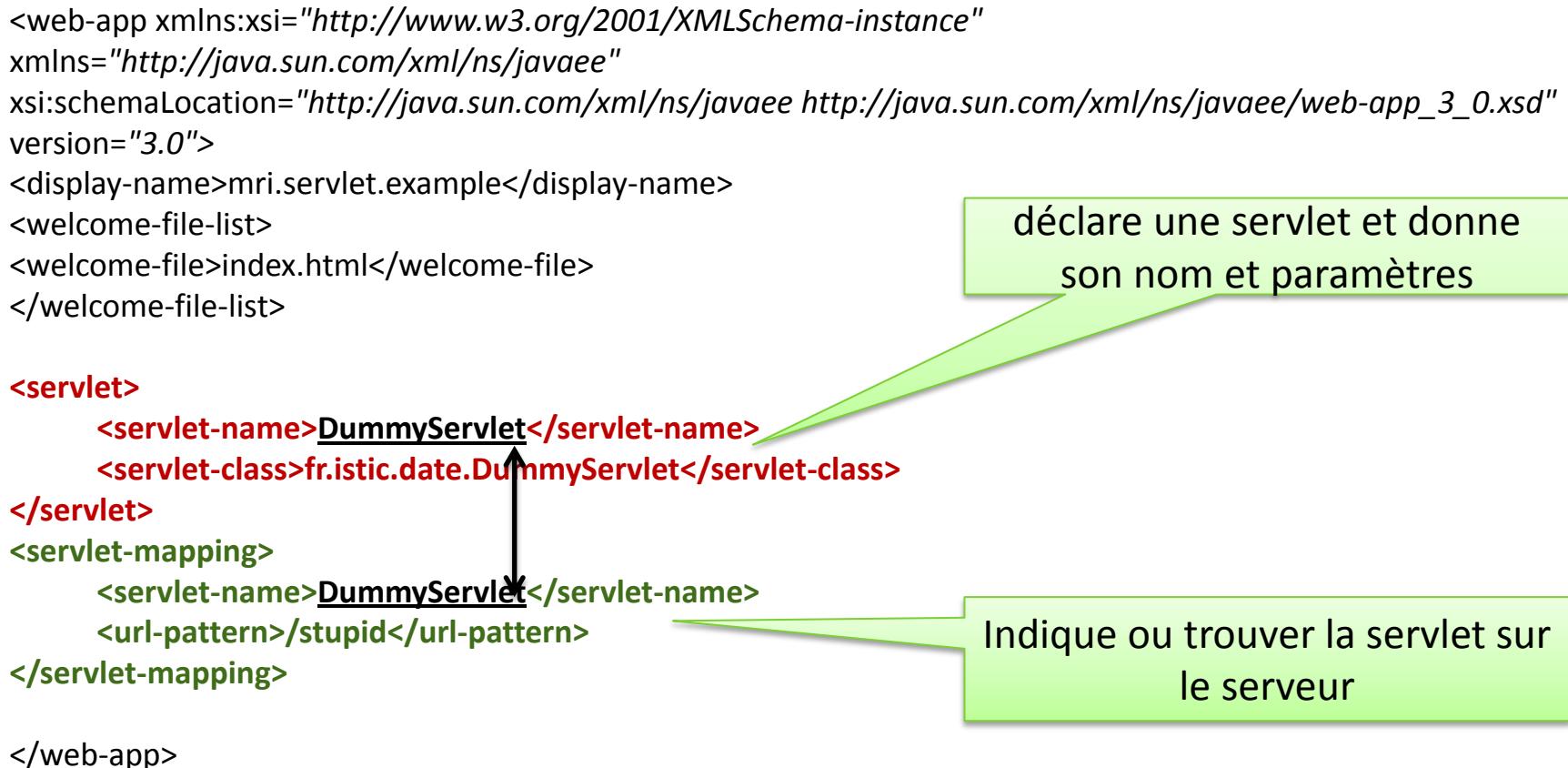
- ▶ Vous pouvez demander sa génération en utilisant "Generate Deployment Descriptor Stub" dans "Java EE Tools" sur votre projet.



Où trouver la servlet ? (web.xml)

- Dans le fichier web.xml. Deux balises principales **servlet** et **servletMapping**

```
<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_3_0.xsd"  
         version="3.0">  
    <display-name>mri.servlet.example</display-name>  
    <welcome-file-list>  
        <welcome-file>index.html</welcome-file>  
    </welcome-file-list>  
  
    <servlet>  
        <servlet-name>DummyServlet</servlet-name>  
        <servlet-class>fr.istic.date.DummyServlet</servlet-class>  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>DummyServlet</servlet-name>  
        <url-pattern>/stupid</url-pattern>  
    </servlet-mapping>  
</web-app>
```

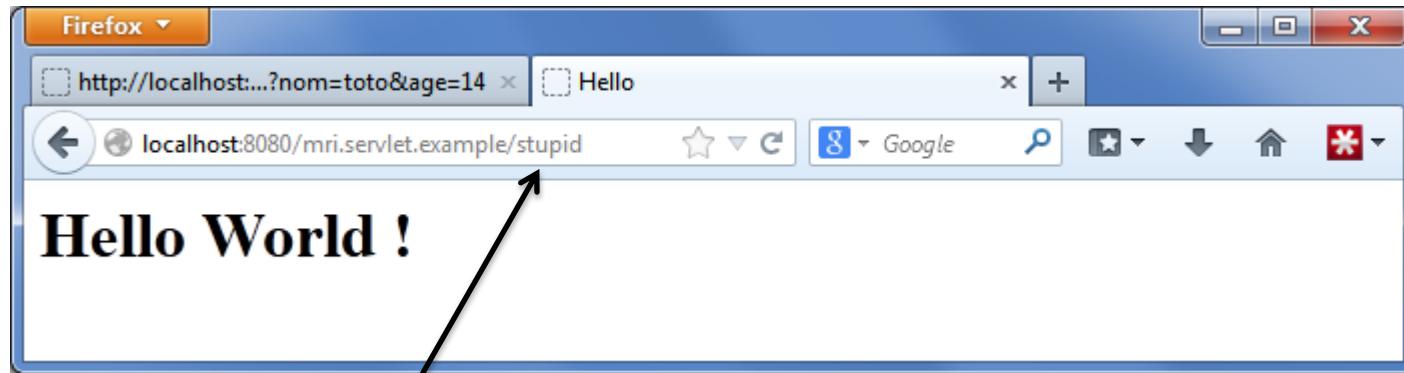


déclare une servlet et donne son nom et paramètres

Indique où trouver la servlet sur le serveur

Où trouver la servlet ? (annotations)

- ▶ On peut aussi utiliser des annotations
- ▶ **@WebServlet("/HelloServlet")**
 - ▶ indique que la servlet se trouve à :
 - ▶ **http://mon-serveur:port/nom-du-projet/HelloServlet**



```
@WebServlet("/stupid")
public class HelloServlet extends HttpServlet {
```

Où trouver la servlet ? (annotations)

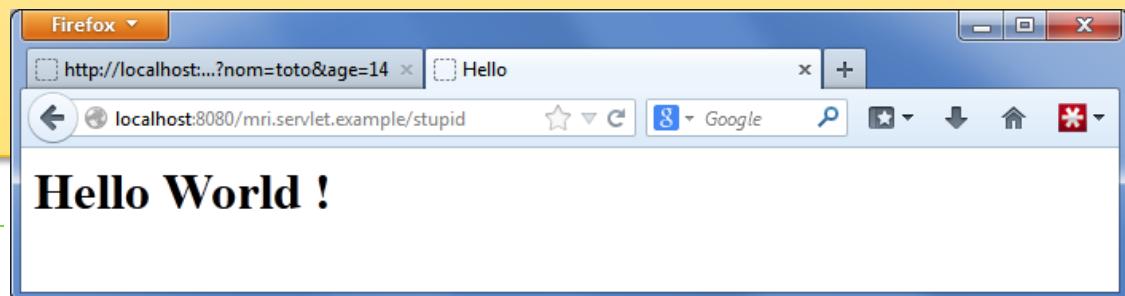
Le mapping ne correspond pas forcément au nom de la classe

```
@WebServlet("/studid")
```

```
public class BasicHello extends HttpServlet {
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,  
IOException {  
    response.setContentType("text/html"); // Set the Content-Type header  
    PrintWriter out = response.getWriter(); // Get the output
```

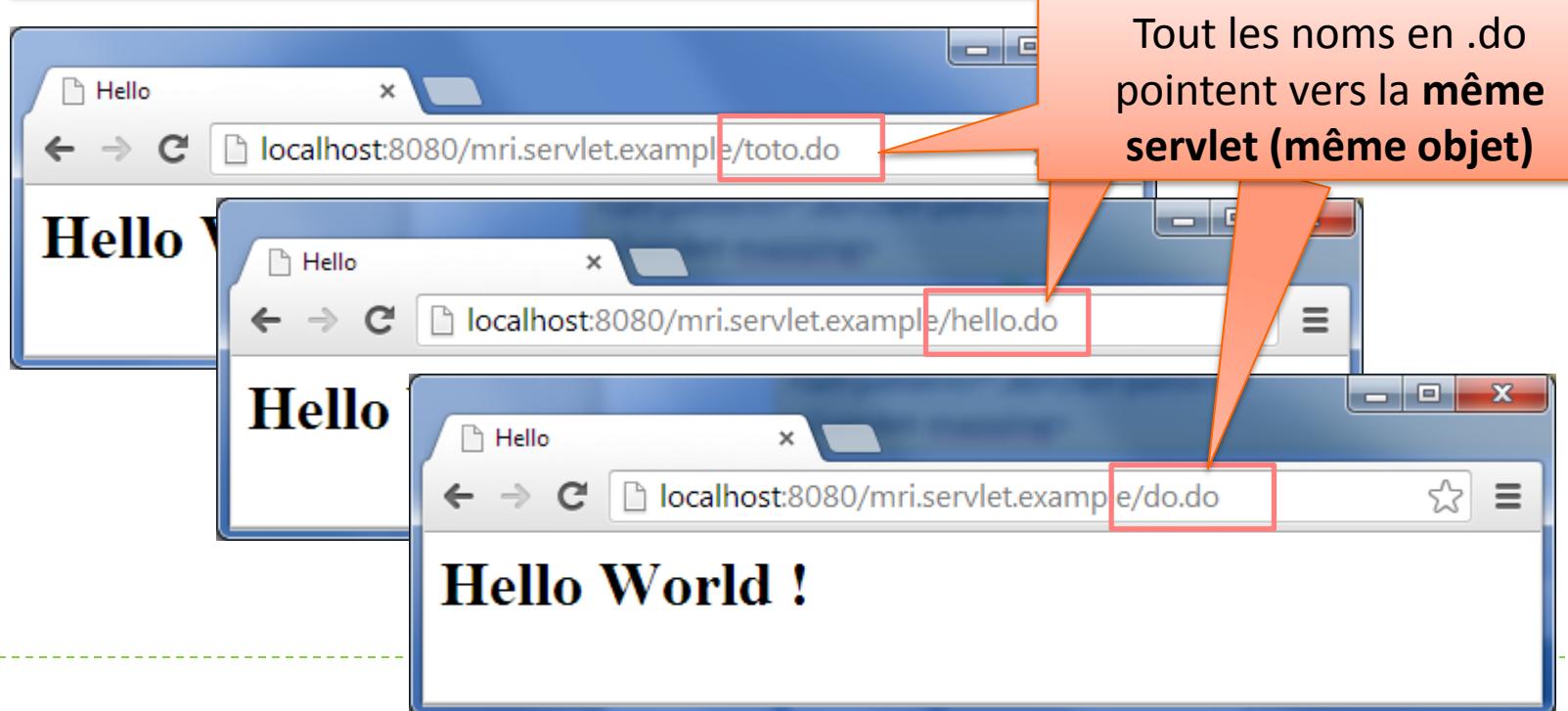
```
String pname = request.getParameter("name"); // Get a parameter  
if (pname == null) {  
    pname = "World !";  
}  
out.println("<html>");  
out.println("<head><TITLE>Hello, " + pname + "</title></head>");  
out.println("<body>");  
out.println("Hello, " + pname);  
out.println("</body></html>");  
out.flush();  
}
```



Mapping sur plusieurs URL

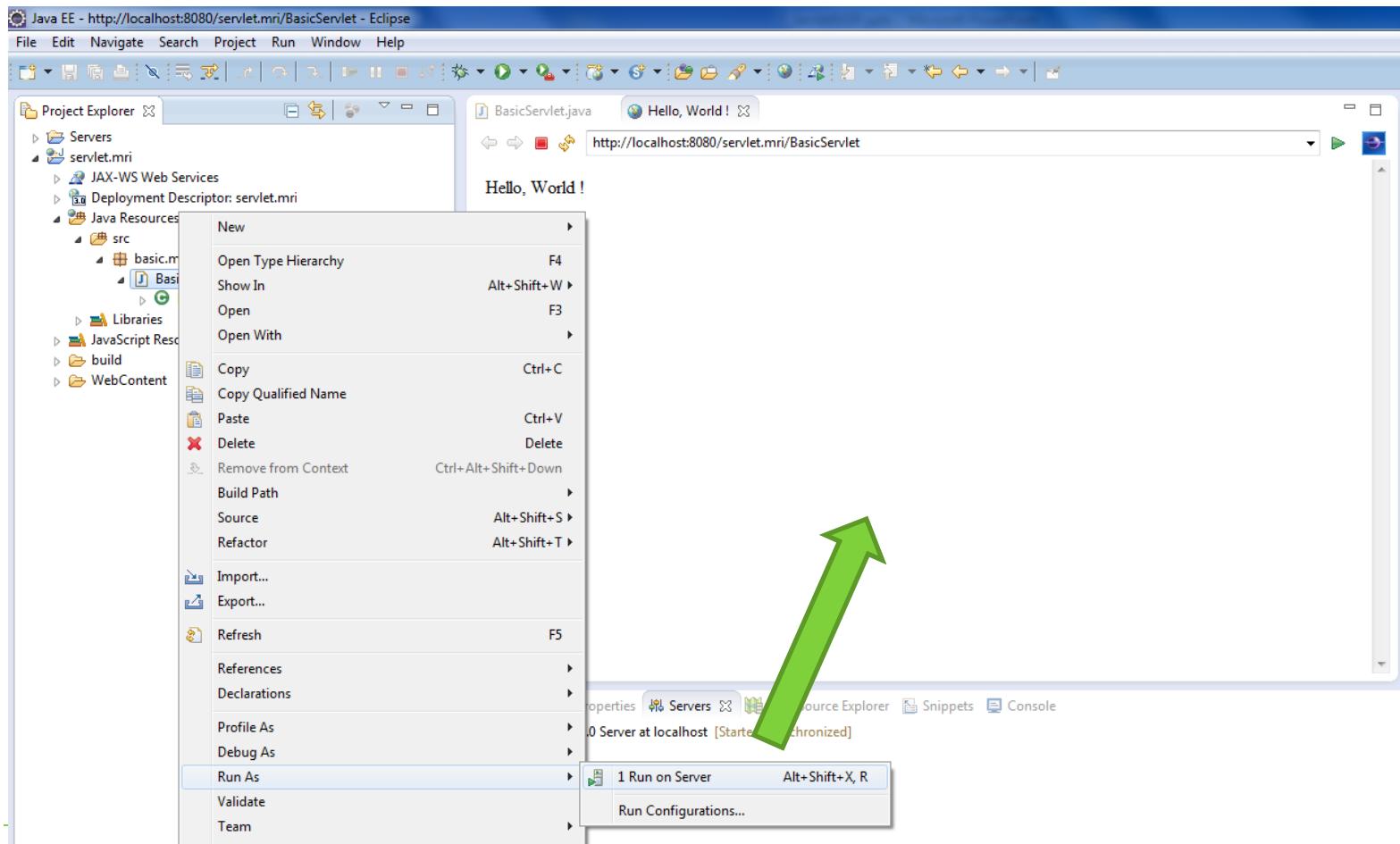
- ▶ Une servlet peut être associée à plusieurs URL

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



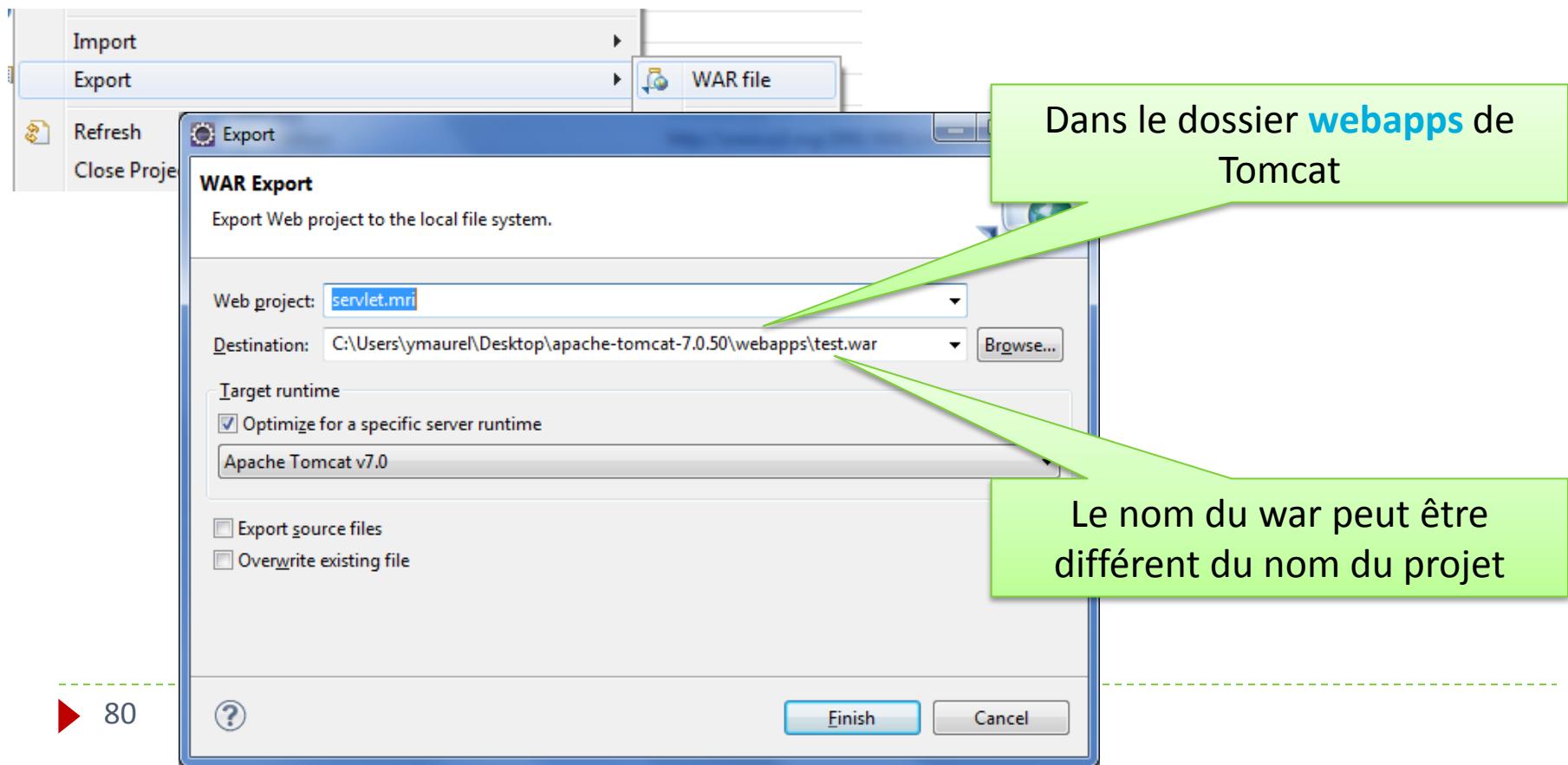
Test de votre projet

▶ Run As > Run On Server



Lancer sans Eclipse : Génération d'un WAR

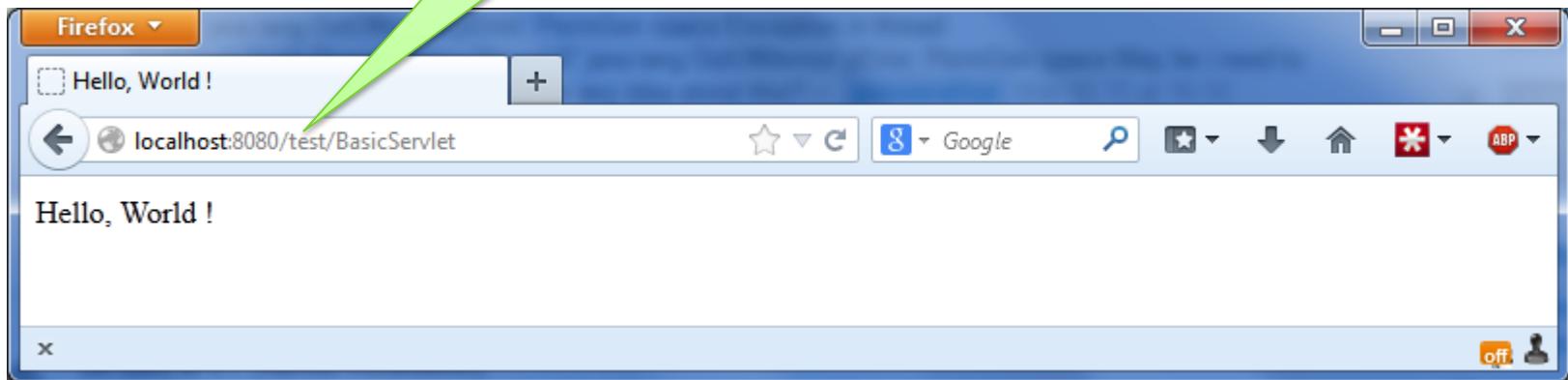
- ▶ Vous pouvez exporter votre WAR dans le dossier **webapps** de Tomcat lorsque vous êtes satisfait

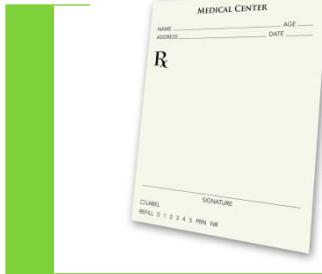


Lancer sans Eclipse : Lancer le serveur

- ▶ On redémarre le serveur en dehors d'Eclipse (via le script startup)

Attention c'est le nom du war
et pas celui du projet





Les formulaires

Les formulaires en HTML

► Contenu par une balise form

Nom du formulaire

URL de la servlet

Méthode : get, post, ...

```
<form name="monform" action="Maservlet" method="get">  
  
<blabla>  
  
</form>
```

Les formulaires en HTML

- ▶ Une multitude de balises pour générer des boutons, champs textes,
- ▶ Attributs importants :
 - ▶ name : le nom qui permet de récupérer la valeur dans la servlet
 - ▶ value : la valeur par défaut

```
<form name="NameForm" action="hello.do" method="post">
<input type="text" name="name" value="Bob">
<input type="submit" value="OK">
</form>
```

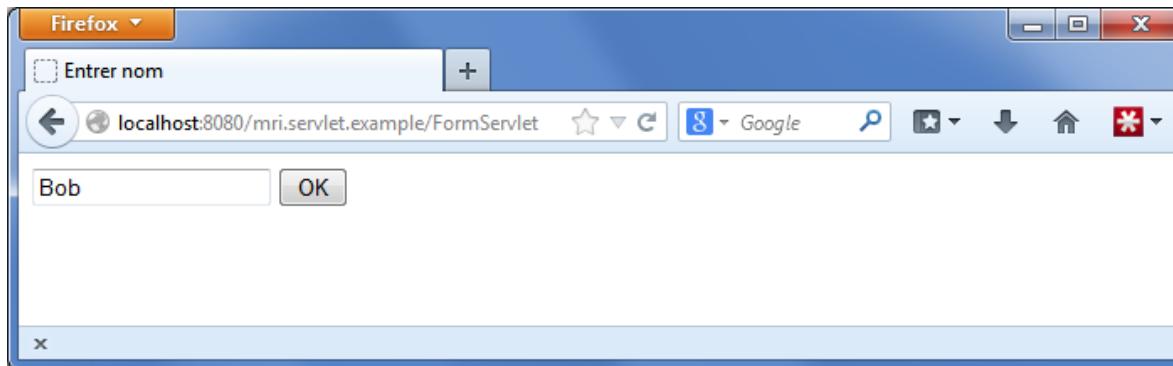
Différence entre doGet et doPost

- ▶ **doGet :**
 - ▶ lit les paramètres dans l'URL
 - ▶ Paramètres passés dans l'url :
 - ▶ **http://chemin-vers-la-servlet/Servlet?nom=toto&age=15**
 - ▶ limite la taille max des données échangées
 - ▶ paramètres peuvent être mis en cache
- ▶ **doPost :**
 - ▶ lit les paramètres dans la requête HTTP

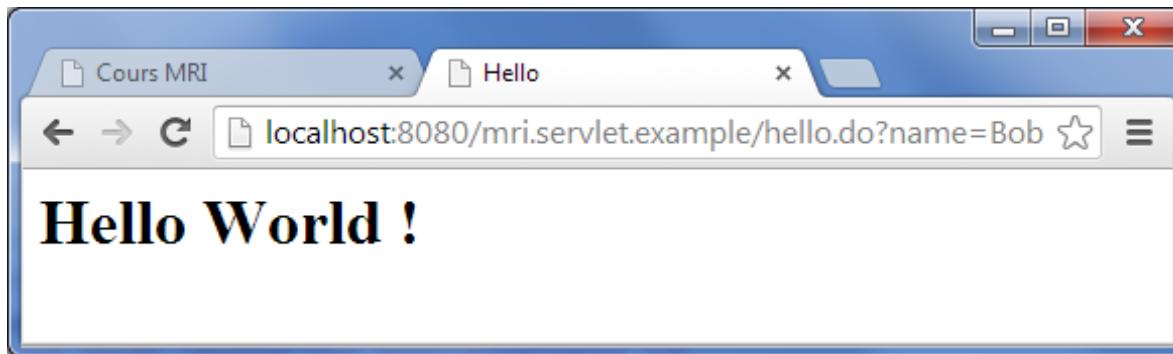
```
POST /guides/php/examples/simple-calculate/calcul4.php HTTP/1.1
Host: tecfa.unige.ch
...
choice%5B%5D=Green&choice%5B%5D=Orange&choice%5B%5D=White
```
 - ▶ les données peuvent être plus grosses
 - ▶ les données ne sont pas mises en cache
- ▶ on peut très bien appeler doGet depuis doPost et vice versa

Exemple

- ▶ On va créer deux servlets



une pour le formulaire



une pour la réponse

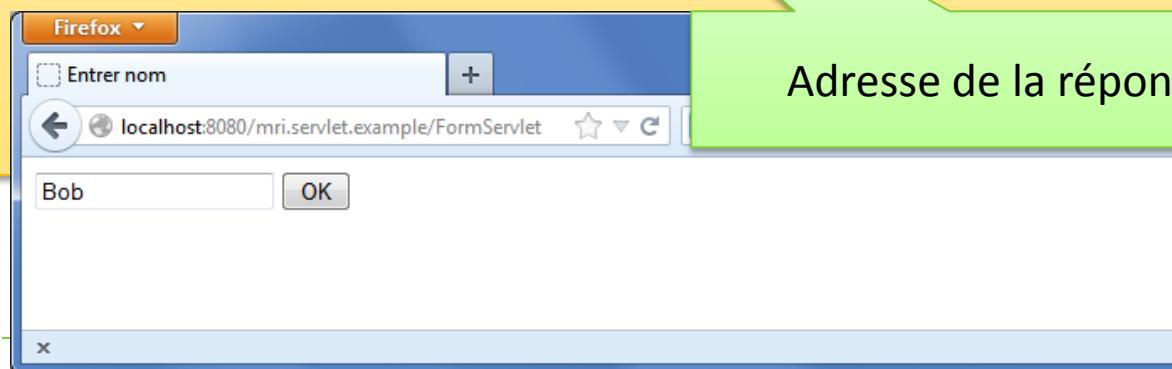
Dans la servlet du formulaire (/FormServlet)

```
@WebServlet("/FormServlet")
public class FormServlet extends HttpServlet {
    ...
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.print("<html>");
        out.print("<head><title>Entrer nom</title></head>");
        out.println("<form name=\"NameForm\" action=\"hello.do\" method=\"post\"");
        out.println("<input type=\"text\" name=\"name\" value=\"Bob\"");
        out.println("<input type=\"submit\" value=\"OK\"");
        out.println("</form>");
        out.print("</html>");
        out.flush();
    }
}
```

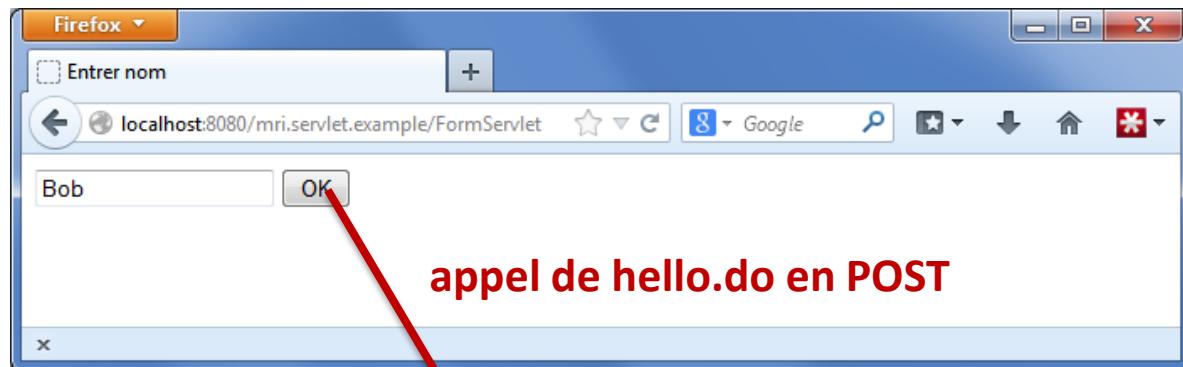
Attention à échapper les guillemets lors de l'écriture

On utilise la méthode POST

Adresse de la réponse

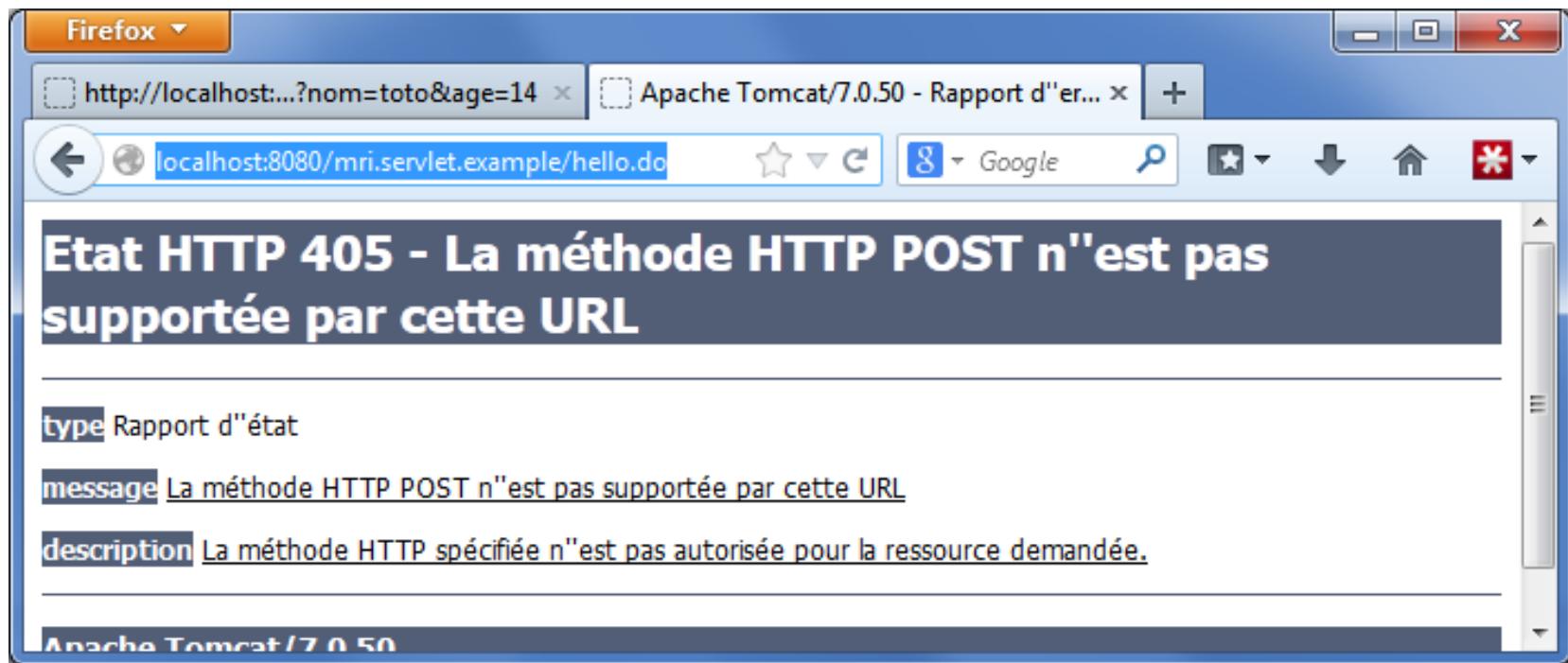


Quand on valide le formulaire



Pourquoi ça marche pas ?

- ▶ La servlet hello.do n'a pas de méthode POST
 - ▶ on obtient une erreur 405

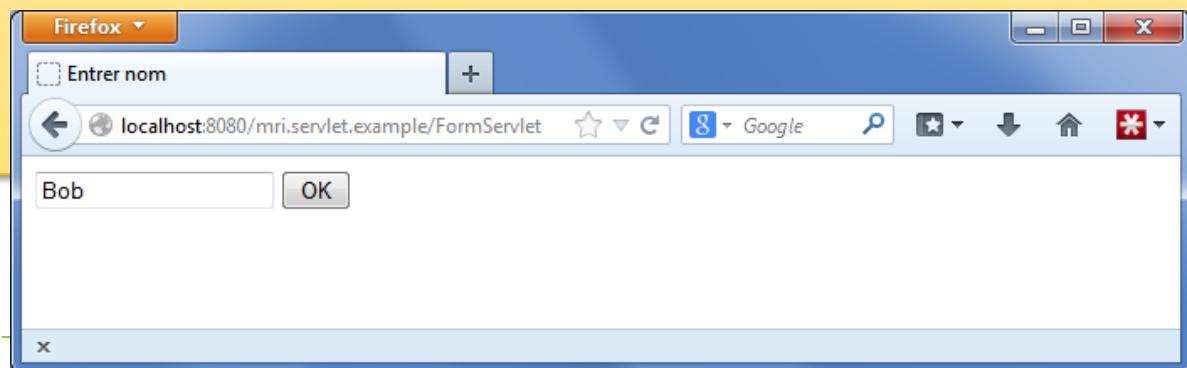


Dans la servlet on change pour GET

```
@WebServlet("/FormServlet")
public class FormServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.print("<html>");
        out.print("<head><title>Entrer nom</title></head>");
        out.println("<form name=\"NameForm\" action=\"hello.do\" method=\"get\"");
        out.println("<input type=\"text\" name=\"name\" value=\"Bob\"");
        out.println("<input type=\"submit\" value=\"OK\"");
        out.println("</form>");
        out.print("</html>");
        out.flush();
    }
}
```

On change vers la méthode
GET



Quand on valide le formulaire

The image shows two screenshots of a Java application. The top screenshot is a modal dialog titled 'Entrez nom' (Enter name) with an 'OK' button. The input field contains 'Bob'. A red arrow points from this dialog to the bottom screenshot, which is a browser window titled 'Hello'. The address bar shows the URL: 'localhost:8080/mri.servlet.example/hello.do?name=Bob'. The page content is 'Hello World !'.

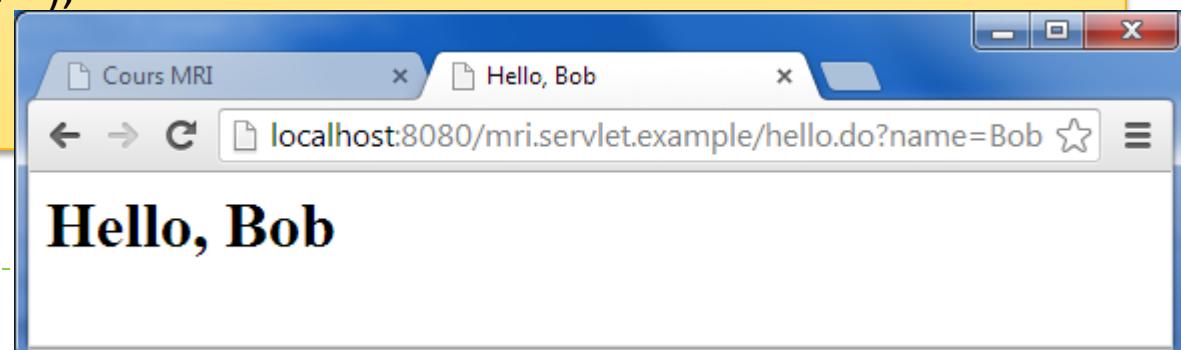
appel de hello.do en GET

on remarque que l'URL contient les paramètres

Utilisation des paramètres dans hello.do

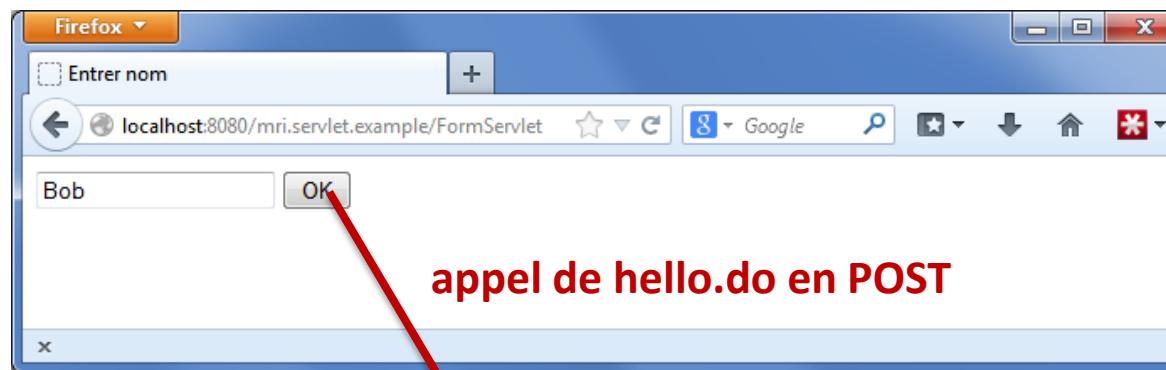
```
@Override  
protected void doGet(HttpServletRequest request, HttpServletResponse response)  
    throws ServletException, IOException {  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
  
    String pname = request.getParameter("name");  
    if (pname == null) {  
        pname = "World !";  
    }  
    out.println("<html>");  
    out.println("<head><TITLE>Hello, " + pname + "</title></head>");  
    out.println("<body>");  
    out.println("<h1>Hello, " + pname + "</h1>");  
    out.println("</body></html>");  
    out.flush();  
}
```

La valeur par défaut est null
si param non initialisé, on
doit gérer ce cas

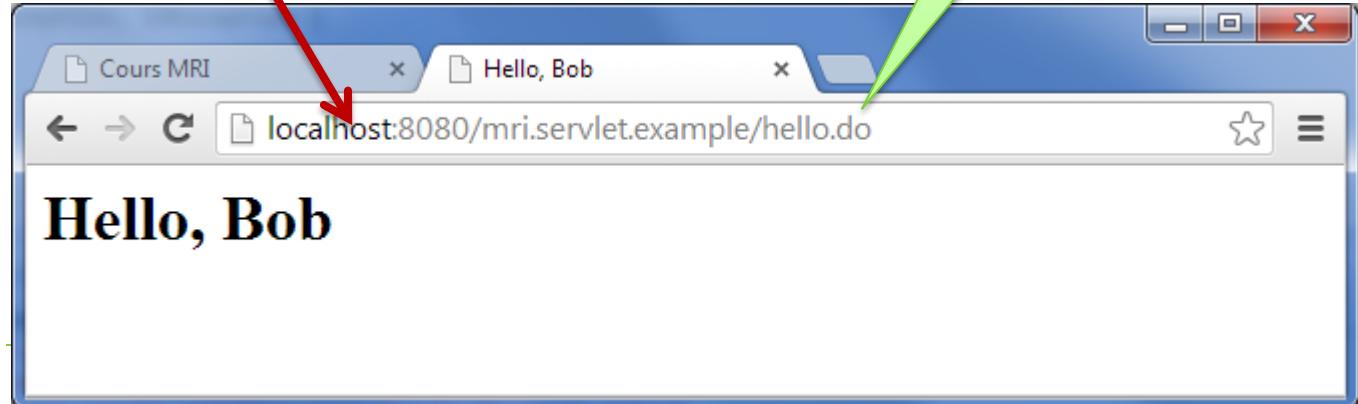


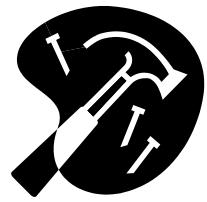
Avec la méthode POST

```
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    this doGet(req, resp);
}
```



on remarque que le paramètre n'est plus passé dans l'URL





Paramétrage d'une servlet

Configuration des paramètres application/servlet

- ▶ Le contexte de l'application
 - ▶ ils concernent toutes l'application
 - ▶ permettent d'avoir une configuration commune à chaque servlet
- ▶ Les paramètres d'initialisations de chaque servlet
 - ▶ se déclarent pour chaque servlet
 - ▶ sont passés à la méthode init à la création
 - ▶ permettent de configurer/reconfigurer une servlet sans modifier le code
- ▶ On modifie généralement ça dans le XML ce qui évite de devoir retoucher au code quand on veux faire un changement
 - ▶ on peut aussi déclarer les paramètres avec des annotations

Contexte global à l'application (i)

- ▶ Objet javax.servlet.ServletContext
 - ▶ Visible par toutes les sessions et toutes les pages d'une même application
- ▶ Ce contexte peut servir à
 - ▶ dialoguer avec le conteneur de servlet
 - ▶ Obtenir des information sur un fichier présent sur le serveur,
 - ▶ Gérer les requêtes d'une même application,
 - ▶ Écrire dans un fichier de log.
 - ▶ communiquer de l'information entre servlets d'une même application



Contexte global à l'application (ii)

- ▶ Ils sont définis dans après la balise web-app dans le web.xml

```
<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_3_0.xsd"
  version="3.0">
  <display-name>mri.servlet.example</display-name>

  <context-param>
    <param-name>name</param-name>
    <param-value>Joe</param-value>
  </context-param>
  <context-param>
    <param-name>password</param-name>
    <param-value>password</param-value>
  </context-param>

</web-app >
```



A la racine du web.xml de
votre war

Contexte global à l'application (iii)

- ▶ On récupère dans la fonction init de la servlet

```
log("Context init parameters:");
ServletContext context = getServletContext();
Enumeration e = context.getInitParameterNames();
while (e.hasMoreElements()) {
    String key = (String) e.nextElement();
    Object value = context.getInitParameter(key);
    log(" " + key + " = " + value);
}
```

```
févr. 16, 2014 11:38:23 AM org.apache.catalina.core.ApplicationContext log
Infos: DummyServlet: Context init parameters:
févr. 16, 2014 11:38:23 AM org.apache.catalina.core.ApplicationContext log
Infos: DummyServlet: name = Joe
févr. 16, 2014 11:38:23 AM org.apache.catalina.core.ApplicationContext log
Infos: DummyServlet: password = password
```

Utiliser le contexte de l'application

- ▶ Obtenir le contexte de l'application (2 étapes)
 - ▶ ServletConfig config = this.getServletConfig();
 - ▶ ServletContext appli = config.getServletContext();
- ▶ Mémoriser un objet
 - ▶ Objetc myObject = ...
 - ▶ appli.setAttribute("monObjet", myObject);
- ▶ Extraire un objet mémorisé
 - ▶ myObject = appli.getAttribute(" monObjet ");



Utiliser le contexte de l'application bis

▶ Problème :

- ▶ Accéder à un fichier local à l'application

▶ Solution :

- ▶ utiliser la méthode `getRealPath()`
- ▶ chemin absolu de la racine de l'application
 - ▶ `String racine = context.getRealPath("/");`
- ▶ chemin absolu vers un fichier quelconque de l'application
 - ▶ `String fichier = context.getRealPath("/xml/index.xml");`

*Notez la présence du "/"
en début de nom*



Paramètres d'initialisation de la servlet (i)

- ▶ Même chose mais pour une servlet donnée

```
<web-app //...>
//...
< servlet >
    < servlet-name > DummyServlet </ servlet-name >
    < servlet-class > fr.istic.date.DummyServlet </ servlet-class >
    < init-param >
        < param-name > language </ param-name >
        < param-value > en </ param-value >
    < /init-param >
    < init-param >
        < param-name > default-name </ param-name >
        < param-value > World </ param-value >
    < /init-param >
< / servlet >
//...
< / web-app >
```

A l'intérieur de la balise servlet !
(attaché à une servlet en particulier)

Paramètres d'initialisation de la servlet (ii)

▶ Dans la fonction init de la servlet

```
public void init(ServletConfig config) throws ServletException {  
    super.init(config);  
    log("Initialisation de la servlet");  
    Enumeration e = getInitParameterNames();  
    while (e.hasMoreElements()) {  
        String key = (String) e.nextElement();  
        String value = getInitParameter(key);  
        log(" " + key + " = " + value);  
    }  
}
```

```
févr. 16, 2014 11:19:51 AM org.apache.catalina.core.ApplicationContext log  
Infos: DummyServlet: default-name = World  
févr. 16, 2014 11:19:51 AM org.apache.catalina.core.ApplicationContext log  
Infos: DummyServlet: language = en
```

Exercice

- ▶ Ecrire une application qui
 - ▶ contient une propriété langue qui permet de définir la langue pour toutes les servlet
 - ▶ une servlet Hello qui parle dans une ou plusieurs langues
 - ▶ la servlet prend en paramètre le nom ("name") par défaut à afficher quand l'utilisateur n'est pas connu.



Les RequestDispatcher

Transfer de contrôle

▶ Objet RequestDispatcher

- ▶ Mis en œuvre par le serveur
- ▶ Les objets Request et Response sont transmis
- ▶ Transmission immédiate au moment de l'exécution de la méthode forward()

- ▶ Le navigateur ne voit rien

▶ Redirect response.sendRedirect(url)

- ▶ Mis en œuvre par le navigateur
- ▶ Le paramètre url est traité comme une nouvelle requête du client
- ▶ Le navigateur enregistre le changement de contrôle dans son historique



L'objet RequestDispatcher

```
RequestDispatcher dispatch;
```

- ▶ URL relatives : par le contexte de la requête
 - ▶ `dispatch=request.getRequestDispatcher("urlRelative");`
 - ▶ Accepte un chemin relatif en paramètre
 - ▶ Ceux qui ne commencent pas par un « / »
- ▶ URL absolues : par le contexte global
 - ▶ `dispatch=this.getServletContext().getRequestDispatcher("urlAbsolue");`
 - ▶ Accepte seulement les URLs absolues
 - ▶ Attention `getRequestDispatcher()` est susceptible de retourner une référence null



Utiliser un RequestDispatcher

▶ Inclure un contenu

- ▶ `dispatch.include(request,response);`
- ▶ Ressource statique : inclusion simple
- ▶ Ressource dynamique :
 - ▶ Envoi d'une requête
 - ▶ Exécution puis inclusion de la réponse
 - ▶ La ressource dynamique appelée ne peut pas modifier les entêtes

▶ Détourner vers un autre producteur

- ▶ `dispatch.forward(request,response);`
- ▶ Toute la sortie de l'appelante est remplacée par celle de l'appelée ... ou presque !



Servlet de login

```
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String userID = "bob";
    private final String password = "bob";

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // get request parameters for userID and password
        String user = request.getParameter("user");
        String pwd = request.getParameter("pwd");

        if (userID.equals(user) && password.equals(pwd)) {
            Cookie loginCookie = new Cookie("user", user);
            loginCookie.setMaxAge(30 * 60);
            response.addCookie(loginCookie);
            response.sendRedirect("LoginSuccess");
        } else {
            RequestDispatcher rd = getServletContext().getRequestDispatcher(
                "/login.html");
            PrintWriter out = response.getWriter();
            out.println("<font color=red>Either user name or password is wrong.</font>");
            rd.include(request, response);
        }
    }
}
```

Si le user/pwd sont corrects, on envoie un cookie au client

et on redirige vers la page "LoginSuccess"

En cas d'échec, on n'envoie pas de cookie

On redirige vers la page login

On insère un message d'erreur avant le login

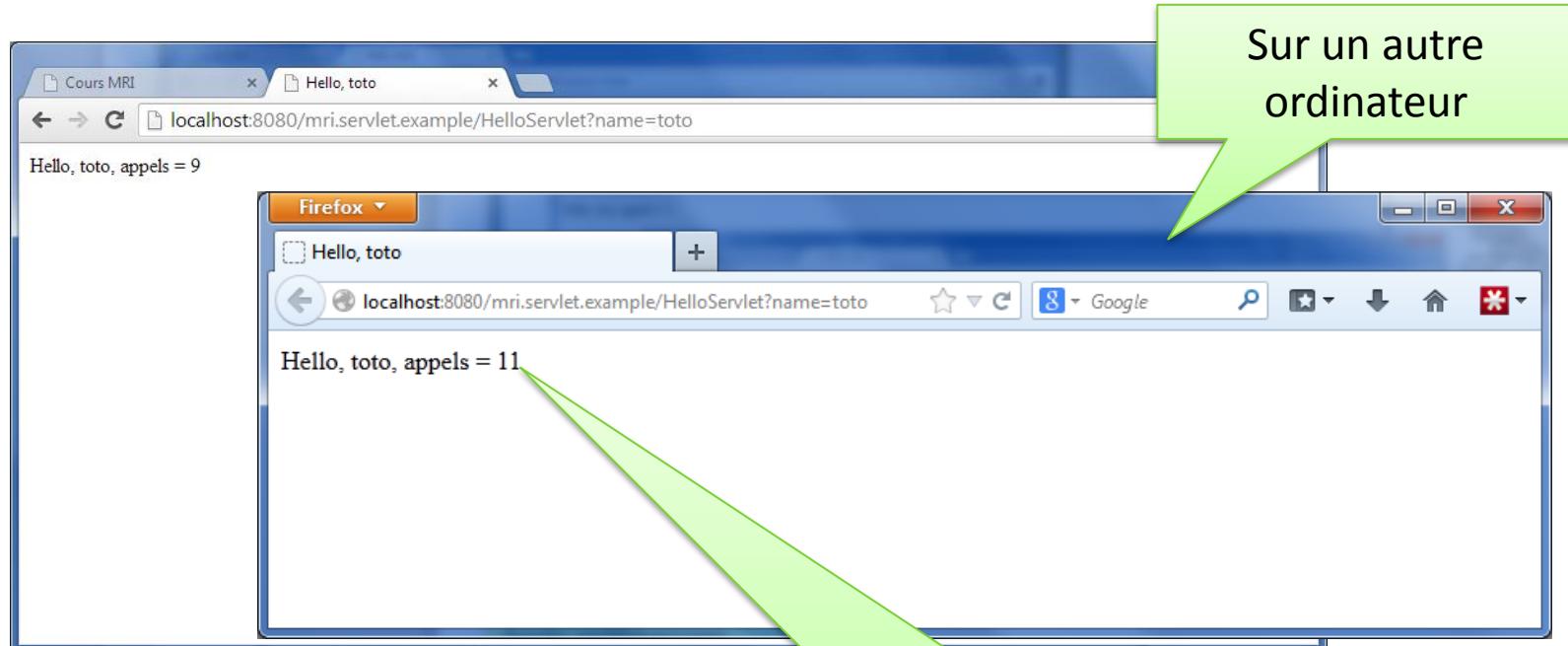
On inclus la réponse de la page login.html



Partage de la servlet et contexte

Partage de la servlet

- ▶ La servlet est un objet unique dans le conteneur :
 - ▶ tous les clients le partage (et donc partage son état)
 - ▶ il faut mettre en place des stratégies pour distinguer les clients



Problème de la concurrence d'accès

▶ Servlet non Thread-Safe

- ▶ Plusieurs Threads peuvent accéder aux données du même objet servlet
- ▶ Il faut gérer la concurrence

▶ Avec synchronized :

Dans la mesure du possible on limite les zones synchronisées

```
out.println("<body>");

synchronized (this) {
    out.println("Hello, " + pname + ", appels = " + appels);
}
```

- ▶ Avant on pouvait utiliser **SingleThreadModel** mais c'est deprecated (et c'était une mauvaise pratique).

HTTP est non connecté

- ▶ pour le serveur, 2 requêtes successives d'un même client sont **indépendantes**
- ▶ Il faut être capable de retrouver l'utilisateur d'une page à l'autre et distinguer les utilisateurs entre eux
- ▶ Plusieurs façons de faire :
 - ▶ faire passer un paramètre d'identification dans l'URL
 - ▶ mais pose des pb de sécurité si l'utilisateur échange des liens
 - ▶ utiliser des cookies
 - ▶ mais les cookies peuvent être désactivés sur le navigateur
 - ▶ utiliser les Session Http qui utilisent une de ces deux méthodes de façon transparente

Utilisation des cookies

- ▶ Le cookie est un petit fichier
 - ▶ généré par le serveur et envoyé au client
 - ▶ stocké sur le navigateur client
 - ▶ renvoyé par le client à chaque requête
- ▶ Permet de garder une trace de la connexion
- ▶ Création d'un cookies :

```
doGet(...) {....
```

```
// Creation du cookie
```

```
Cookie cookie= new Cookie( "sonNom", "saValeur" );  
cookie.setMaxAge(24*3600);  
response.addCookie(cookie);
```

```
}
```

Création avec nom permettant de le retrouver et une valeur spécifique à chaque client

Durée de vie du cookie en secondes

envoie du cookie sur le navigateur du client

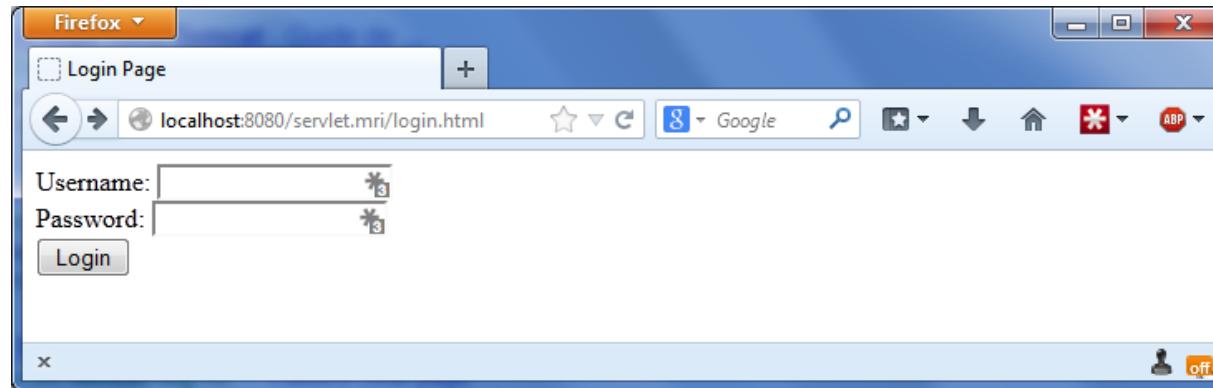
Lecture d'un cookies (différents paramètres)

```
@WebServlet("/CookiesServlet")
public class CookiesServlet extends HttpServlet {

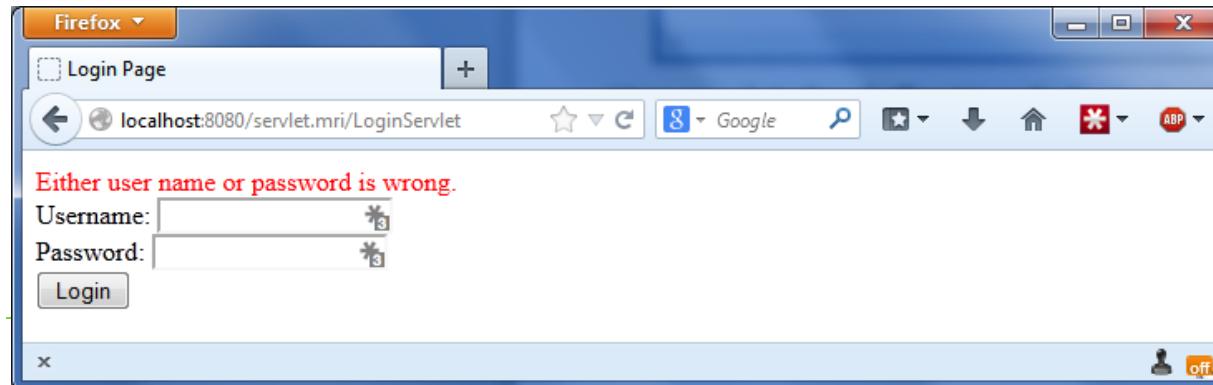
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        out.println("Cookies:");
        javax.servlet.http.Cookie[] cookies = req.getCookies();
        for (int c = 0; c < cookies.length; c++) {
            out.print("Name:" + cookies[c].getName());
            out.print("Value:" + cookies[c].getValue());
            out.print("Domain:" + cookies[c].getDomain());
            out.print("Path:" + cookies[c].getPath());
            out.print("Secure:" + cookies[c].getSecure());
            out.print("Version:" + cookies[c].getVersion());
            out.print("MaxAge:" + cookies[c].getMaxAge());
            out.println("Comment :" + cookies[c].getComment());
        }
    }
}
```

Exemple avec 3 pages

- ▶ Une page login pour afficher un formulaire

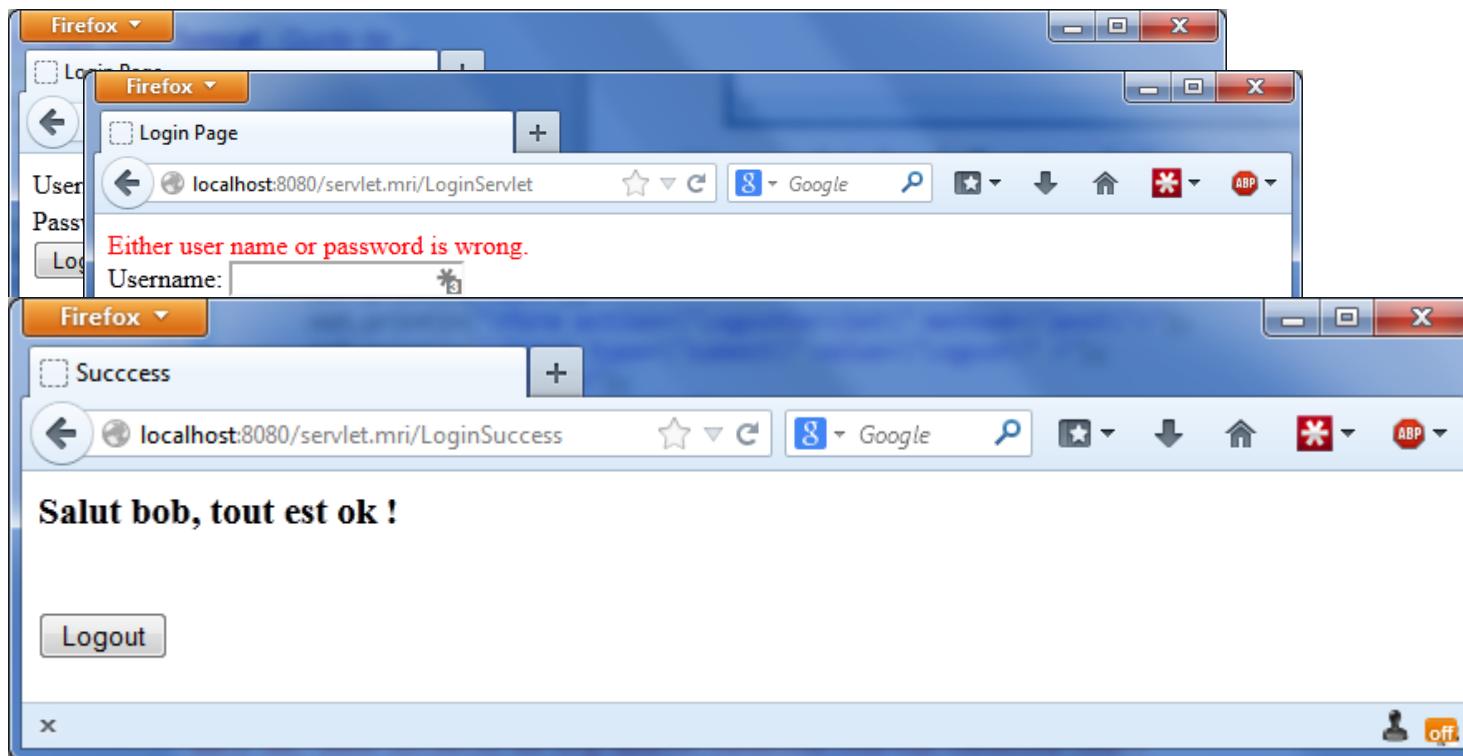


- ▶ Une servlet de vérification de login (qui insère un message en cas d'erreur) et ajoute un cookie en cas de succès



Exemple avec 3 pages

- ▶ Une page pour afficher les résultats qui utilise le cookie pour savoir que l'utilisateur est authentifié :



Formulaire de login

► Une page HTML simple

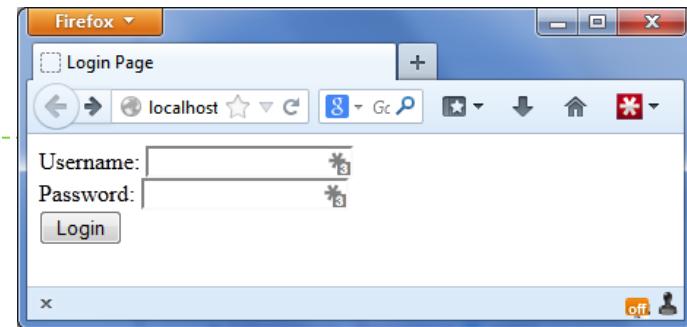
```
<!DOCTYPE html>
<html>
<head>
<meta charset="US-ASCII">
<title>Login Page</title>
</head>
<body>

<form action="LoginServlet" method="post">
Username: <input type="text" name="user"> <br>
Password: <input type="password" name="pwd"> <br>
<input type="submit" value="Login">
</form>

</body>
</html>
```

On redirige vers la servlet de login

On utilise les paramètres user/pwd



Servlet de login

```
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final String userID = "bob";
    private final String password = "bob";

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // get request parameters for userID and password
        String user = request.getParameter("user");
        String pwd = request.getParameter("pwd");

        if (userID.equals(user) && password.equals(pwd)) {
            Cookie loginCookie = new Cookie("user", user);
            loginCookie.setMaxAge(30 * 60);
            response.addCookie(loginCookie);
            response.sendRedirect("LoginSuccess");
        } else {
            RequestDispatcher rd = getServletContext().getRequestDispatcher(
                "/login.html");
            PrintWriter out = response.getWriter();
            out.println("<font color=red>Either user name or password is wrong.</font>");
            rd.include(request, response);
        }
    }
}
```

Si le user/pwd sont corrects, on envoie un cookie au client

et on redirige vers la page "LoginSuccess"

En cas d'échec, on n'envoie pas de cookie

On redirige vers la page login

On insère un message d'erreur avant le login

On inclus la réponse de la page login.html

En cas de succès on affiche un message d'accueil

```
@WebServlet("/LoginSuccess")
public class LoginSuccess extends HttpServlet {

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

        PrintWriter out = response.getWriter();
        response.setContentType("text/html"); // Set the Content-Type header

        String userName = null;

        Cookie[] cookies = request.getCookies();
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                if (cookie.getName().equals("user")) {
                    userName = cookie.getValue();
                }
            }
        }
        if (userName == null) {
            response.sendError(HttpServletResponse.SC_FORBIDDEN);
        }

        out.println("<html><head><title>Success</title></head><body>");
        out.println("<h3>Salut " + userName + ", tout est ok !</h3>");
        out.println("<br>");
        out.println("<form action=\"LogoutServlet\" method=\"post\">");
        out.println("<input type=\"submit\" value=\"Logout\" >");
        out.println("</form>");
        out.println("</body></html>");

    }
}
```

On parcourt les cookies et on cherche le cookie nommé "user"

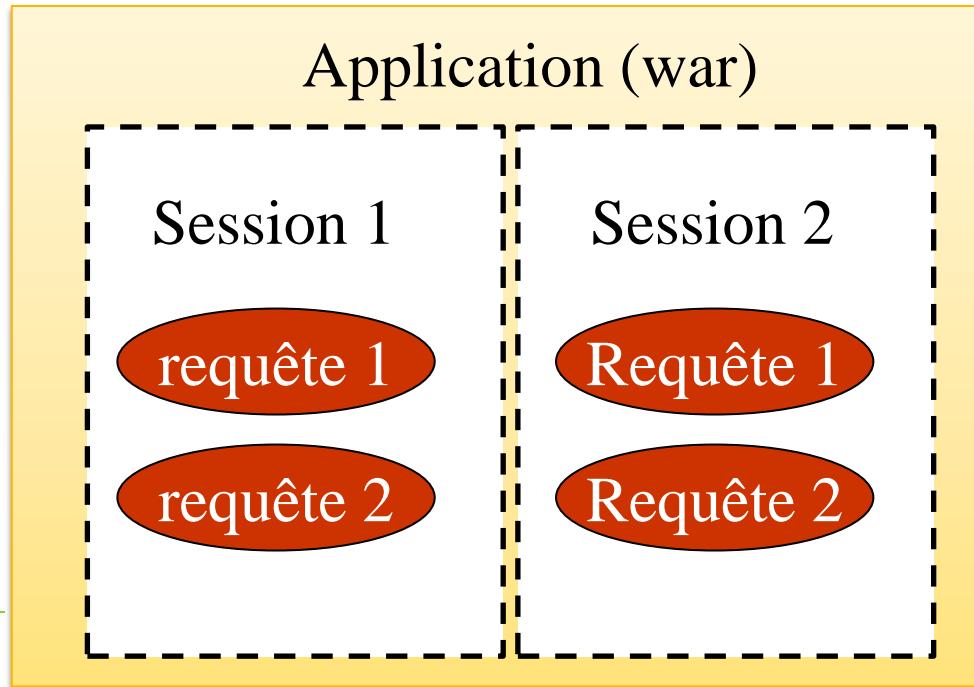
Si cookie trouvé, on récupère la valeur

Si cookie non trouvé on interdit l'accès à la page

On affiche un message d'accueil

Conserver l'information dans les contextes

- ▶ Les servlets offrent trois portées de contextes par application
 - ▶ L'application (ServletContext)
 - ▶ **La session (HttpSession)**
 - ▶ La requête (HttpServletRequest)



HttpSession

- ▶ Rôle
 - ▶ suivi de session automatique par le serveur de servlet
- ▶ Objet javax.servlet.http.HttpSession
- ▶ Conserver de l'information entre deux requêtes liées d'un même utilisateur
 - ▶ Une session dure un temps borné et traverse une ou plusieurs connections d'un et un seul utilisateur.
 - ▶ L'utilisateur peut ainsi visité un même site plusieurs fois en étant reconnu.
 - ▶ Le serveur dispose de diverses méthodes pour mainenir la notion de session : cookies ou le réécriture d'URLs.



Objet HttpSession

- ▶ Obtenir la session

- ▶ HttpSession session = request.getSession();

- ▶ Gérer des attributs de session

- ▶ void setAttribute(String name, Object value)
 - ▶ void removeValue(String name)
 - ▶ Object getAttribute(String name)
 - ▶ java.util.Enumeration getAttributeNames()

- ▶ Divers

- ▶ Long getCreationTime()
 - ▶ String getId()



Problèmes pour obtenir la session

- ▶ L'identification de la session est faite de façon transparente par le conteneur de servlet
 - ▶ utilise un cookie
 - ▶ ou une réécriture d'URL
- ▶ la méthode getSession() doit être appelée avant tout envoi de données au navigateur
 - ▶ la méthode doit être invoquée avant toute écriture sur le flux de sortie de la servlet



Gérer la durée d'une session

- ▶ **int setMaxInactiveInterval(int interval)**
 - ▶ Définit l'intervalle de temps maximum entre deux requêtes avant que la session n'expire
- ▶ **int getMaxInactiveInterval(int interval)**
 - ▶ Retourne l'intervalle de temps maximum entre deux requêtes avant que la session n'expire



Exemple de session avec compteur

```
@WebServlet("/Session")
public class SessionSample extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        // Step 1: Get the Session object
        boolean create = true;
        HttpSession session = request.getSession(create);

        // Step 2: Get the session data value
        Integer ival = (Integer) session.getAttribute("sessiontest.counter");
        if (ival == null)
            ival = new Integer(1);
        else
            ival = new Integer(ival.intValue() + 1);
        session.setAttribute("sessiontest.counter", ival);

        // Step 3: Output the page
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>Session Tracking Test</title></head>");
        out.println("<body>");
        out.println("<h1>Session Tracking Test</h1>");
        out.println("You have hit this page " + ival + " times" + "<br>");
        out.println("Your " + request.getHeader("Cookie"));
        out.println("</body></html>");
    }
}
```

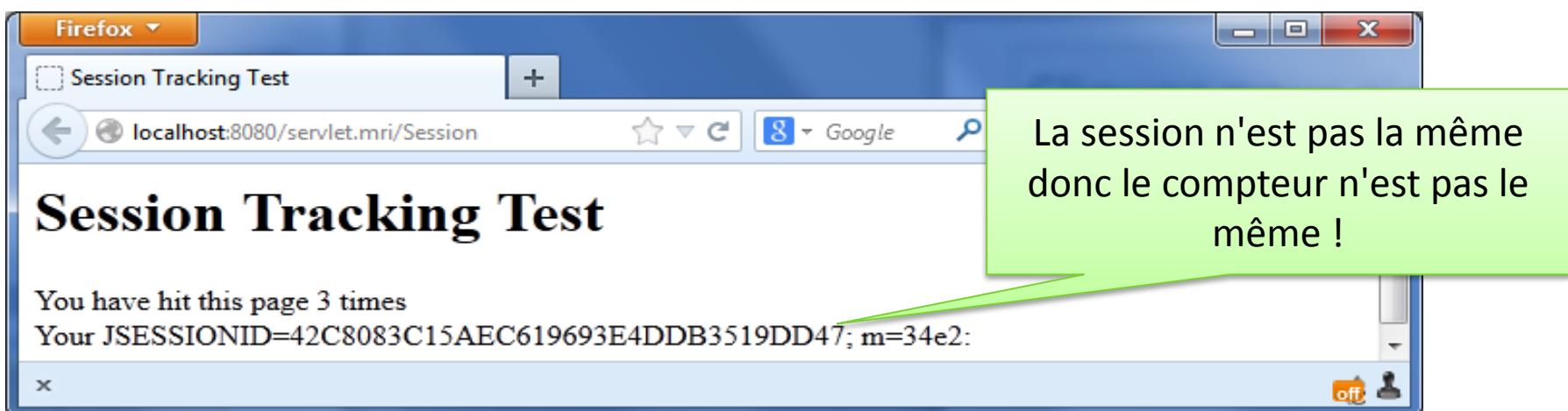
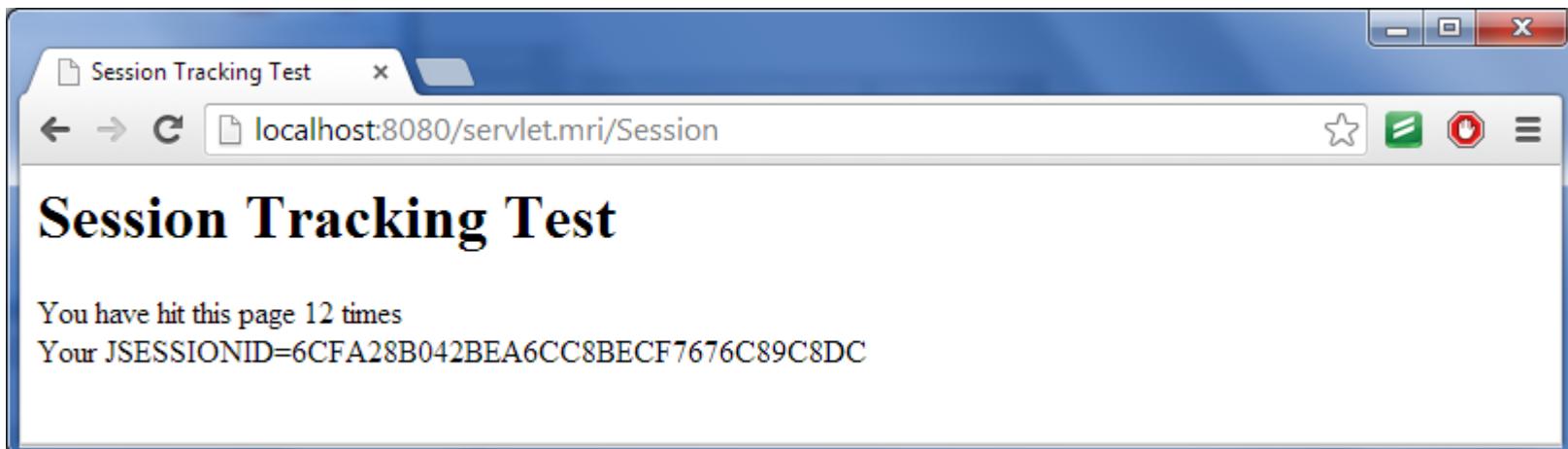
Création

lecture de la valeur précédente

incrémentation

affichage

Sur deux clients différents



Les JSP

Embarqué dans du HTML

- ▶ Servlet = des fichiers .class sur le serveur
- ▶ JSP = sources du programmes dans le .html
 - ▶ attention ! ne veux pas dire que le client reçoit les sources JSP, la page est générée à la manière des servlets (ressemble à PHP sur le fonctionnement)

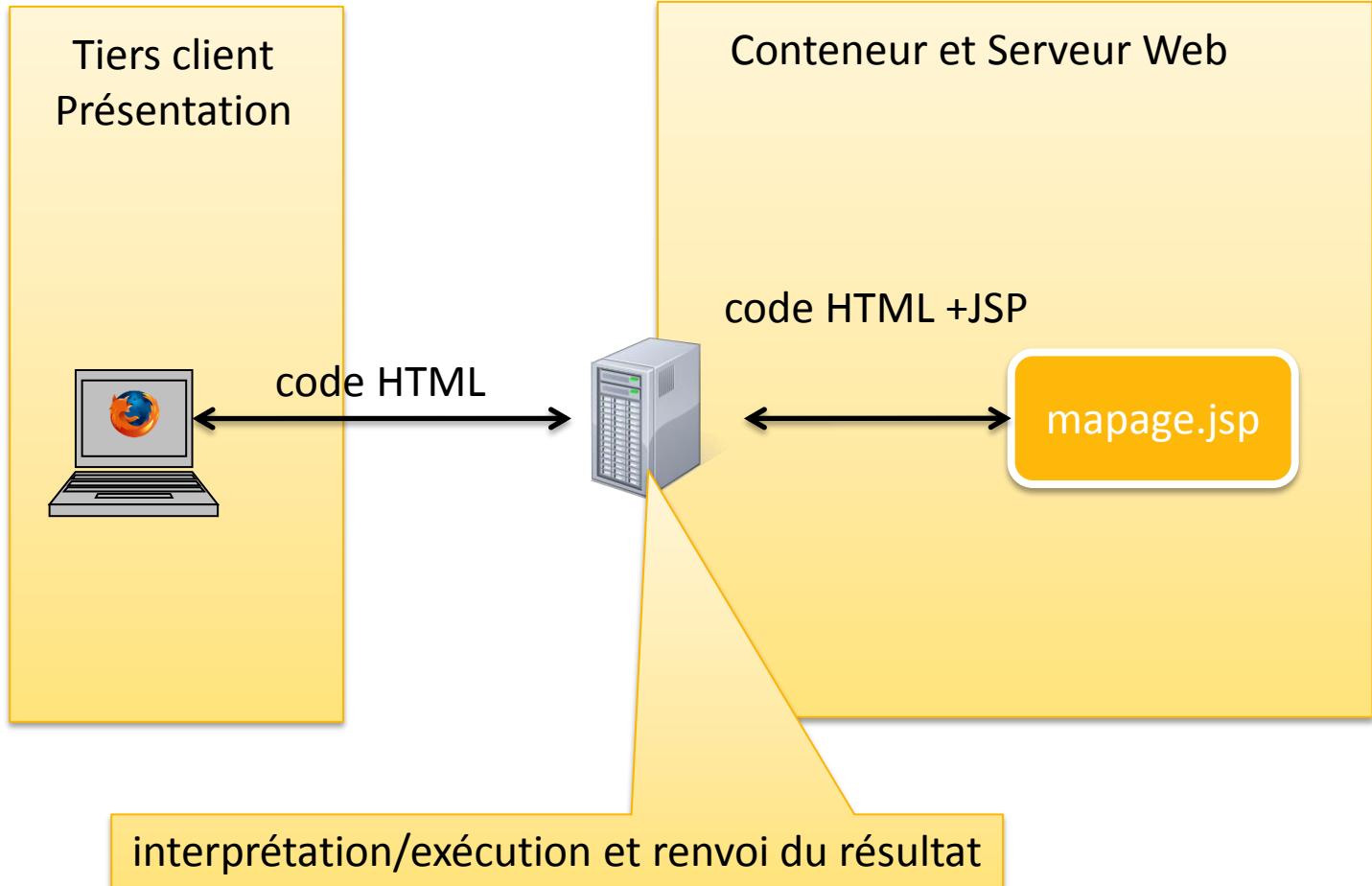
	Client	Serveur
des classes java qui génère le HTML	applets JAVA	les servlets
du code inclus dans une page HTML	le javascript	les JSP

Le client ne reçoit que de l'HTML !

A quoi ressemble une jsp

- ▶ Dans des fichiers d'extension .jsp
- ▶ Une page HTML classique avec du code embarqué via des balises spéciales <% et %>
 - ▶ ressemble au PHP avec les <?php et ?>
- ▶ Les fichiers jsp sont accessibles sur le serveur comme des .html classique via une URL (et un mapping si on le souhaite)
 - ▶ l'accès à la page provoque l'exécution de la JSP

Fonctionnement simplifié



Exemple

On indique qu'on fait une JSP

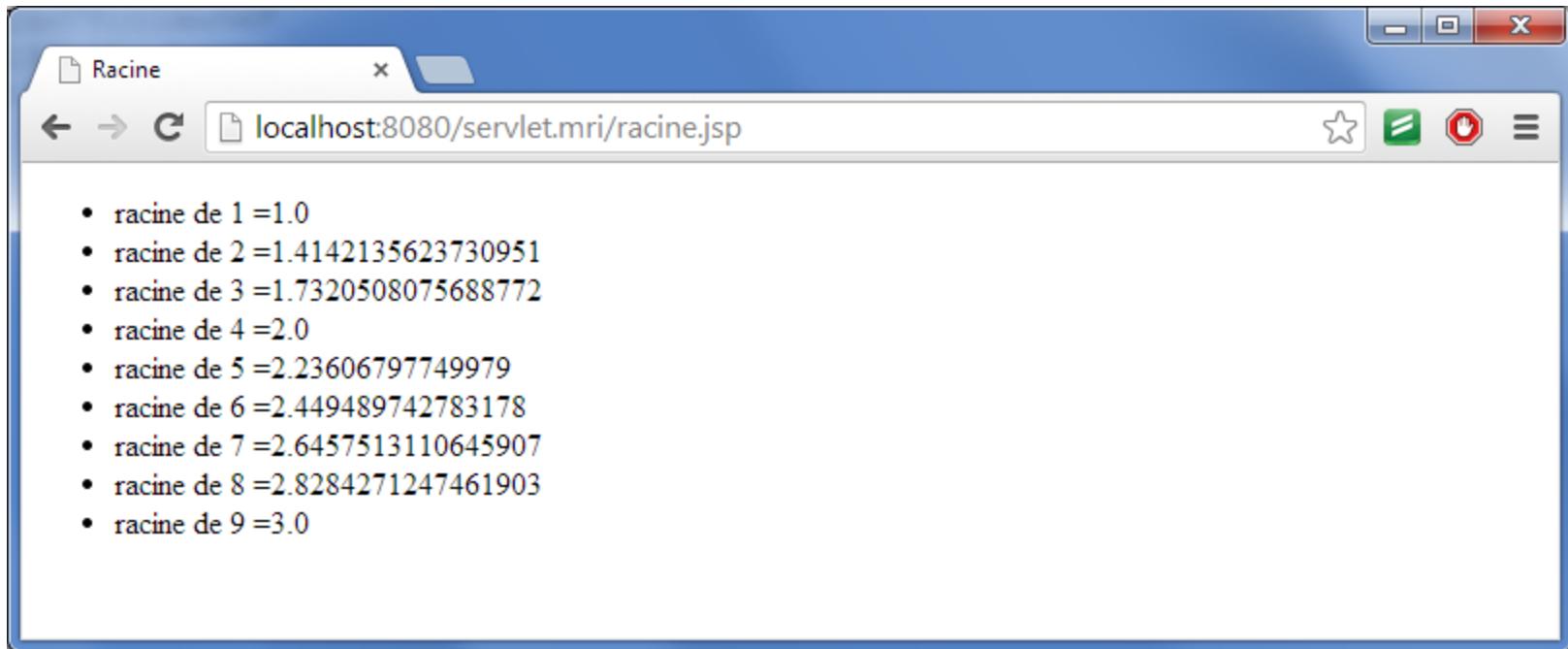
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Racine</title>
</head>
<body>
<ul>
<%
for ( int i=1;i < 10; i++ ) {
out.print( "<li> racine de " + i + " =" + Math.sqrt(i) + "</li>" );
}
%>
</ul>
</body>
</html>
```

On met la balise <% lorsqu'on veux entrer du code java

On a accès à la variable out qui permet d'écrire le résultat

On peut faire appel à des classes de java

Résultat



Code source reçu par le navigateur

```
<!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>Racine</title>
</head>
<body>

<ul>
<li> racine de 1 = 1.0 </li>
<li> racine de 2 = 1.4142135623730951 </li>
<li> racine de 3 = 1.7320508075688772 </li>
<li> racine de 4 = 2.0 </li>
<li> racine de 5 = 2.23606797749979 </li>
<li> racine de 6 = 2.449489742783178 </li>
<li> racine de 7 = 2.6457513110645907 </li>
<li> racine de 8 = 2.8284271247461903 </li>
<li> racine de 9 = 3.0 </li>
</ul>
</body>
</html>
```

Directive <%= ... %>

- ▶ On peut simplifier l'affichage via l'utilisation des balises `<%= maValeur %>` qui permette d'afficher la valeur `maValeur`

```
<ul>
<%
for ( int i=1;i < 10; i++ ) {
%>
<li> racine de <%= i %> = <%=Math.sqrt(i)%> </li>
<%
}
%>
</ul>
```

On n'a pas encore terminé la boucle mais on a le droit d'insérer du code HTML

On utilise la directive `<%=` pour l'affichage

On pense à fermer notre boucle

Même résultat

- ▶ Mais le code est plus lisible



The screenshot shows a web browser window titled "Racine". The address bar displays "localhost:8080/servlet.mri/racine.jsp". The page content is a list of square roots from 1 to 10, followed by the corresponding Java code that generates this list.

Racine de i	Résultat
racine de 1	=1.0
racine de 2	=1.4142135623730951
racine de 3	=1.7320508075688772
racine de 4	=2.0
racine de 5	=2.23606797749979
racine de 6	=2.449489742783178
racine de 7	=2.6457513110645907
racine de 8	=2.8284271247461903
racine de 9	=3.0

```
<ul>
<%>
for ( int i=1;i < 10; i++ ) {
%>
    <li> racine de <%= i %> = <%=Math.sqrt(i)%> </li>
<%
}
%>
</ul>
```

Variables locales/membres

- ▶ Par défaut les variables sont locales lorsqu'elles sont déclarées entre <% et %>
 - ▶ elles sont réinitialisées à chaque interprétation
- ▶ On peut déclarer des variables membres de la classes en utilisant <%! ... %>
 - ▶ elles persistent entre différents appels (tant que l'objet JSP existe)

```
<body>
```

```
<% int locale = 0; %>
```

```
<%! int membre = 0; %>
```

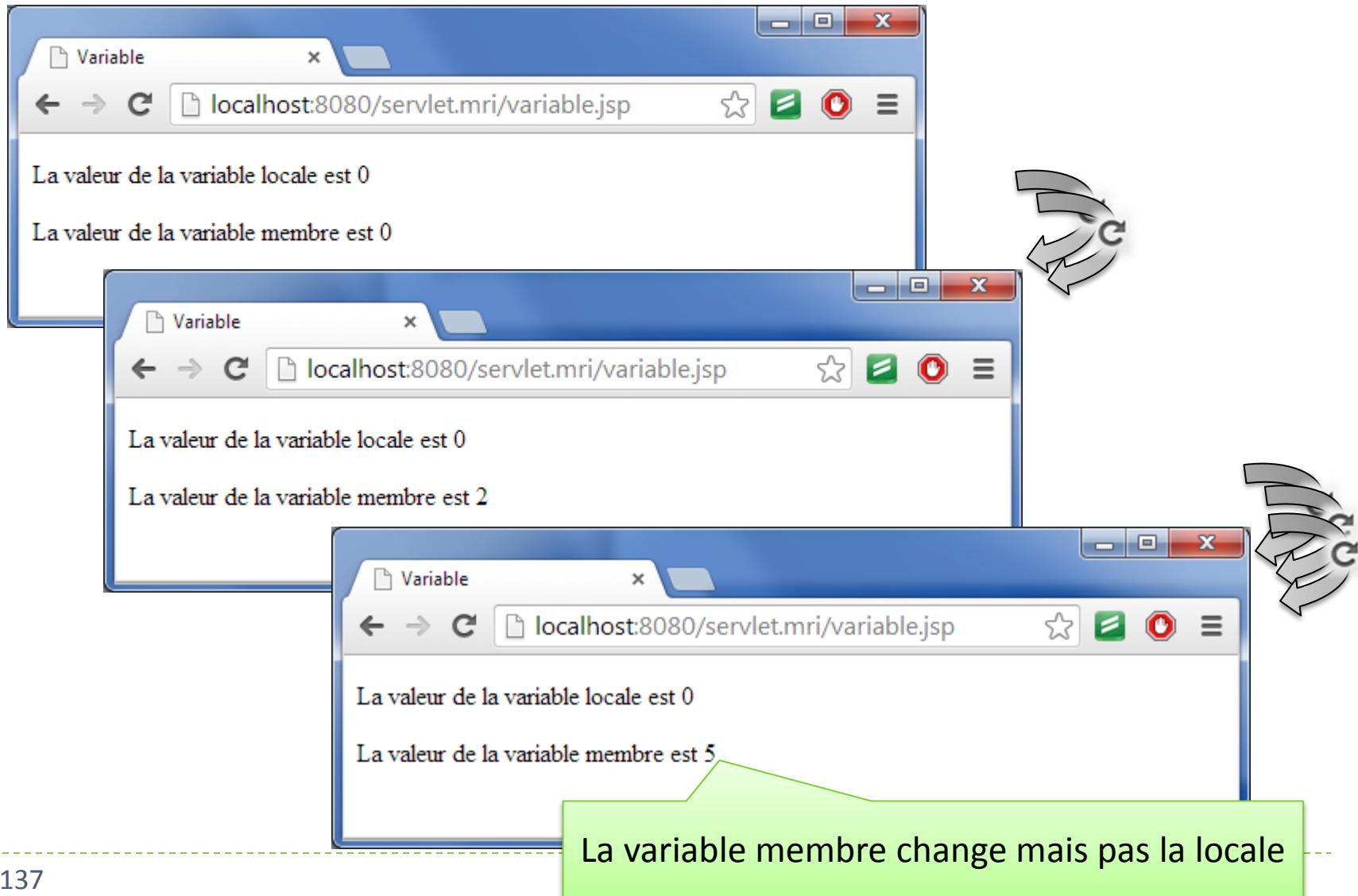
Cette variable sera conservée

```
<p> La valeur de la variable locale est <%= locale++ %> </p>
```

```
<p> La valeur de la variable membre est <%= membre++ %> </p>
```

```
</body>
```

Exemple de la persistances des variables

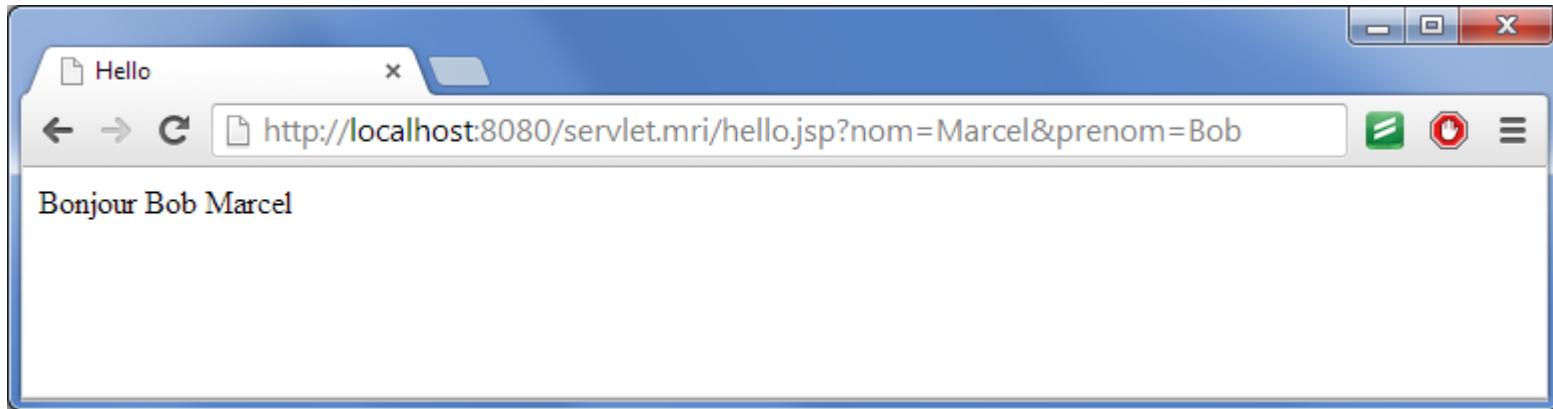


Les objets implicites fournis à la JSP

- ▶ La JSP a accès à un ensemble d'objet fournis implicitement lors de son chargement :
 - ▶ **out** le writer permettant d'écrire sur la sortie
 - ▶ **request** même chose que pour Servlet
 - ▶ **response** même chose que pour Servlet
- ▶ **exception** objet représentant la dernière exception générée
- ▶ **session** permet d'accéder à la session courante
- ▶ **application** le contexte applicatif
- ▶ **page** l'instance de l'objet qui représente la page JSP

Exemple de traitement d'un formulaire

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
//....
<body>
Bonjour <%= request.getParameter("prenom") %> <%= request.getParameter("nom")
%>
</body>
</html>
```



La directive <%@page ... %>

- ▶ Première ligne de la JSP
- ▶ Permet de modifier le comportement de la jsp :
- ▶ <%@page import="..."%> (ex. <%@ page import="fr.istic.mri.*"%> permet d'importer des packages
- ▶ errorPage="..." permet d'indiquer une page de traitement en cas d'erreur
 - ▶ la JSP de traitement d'erreur doit déclarer isErrorPage="true"

Exemple de gestion d'erreur

- ▶ Une page stupide (div0) avec un code stupide :

```
<body>  
  
<% int i = 0/0; %>  
  
</body>
```



On ajoute une page de traitement d'erreur

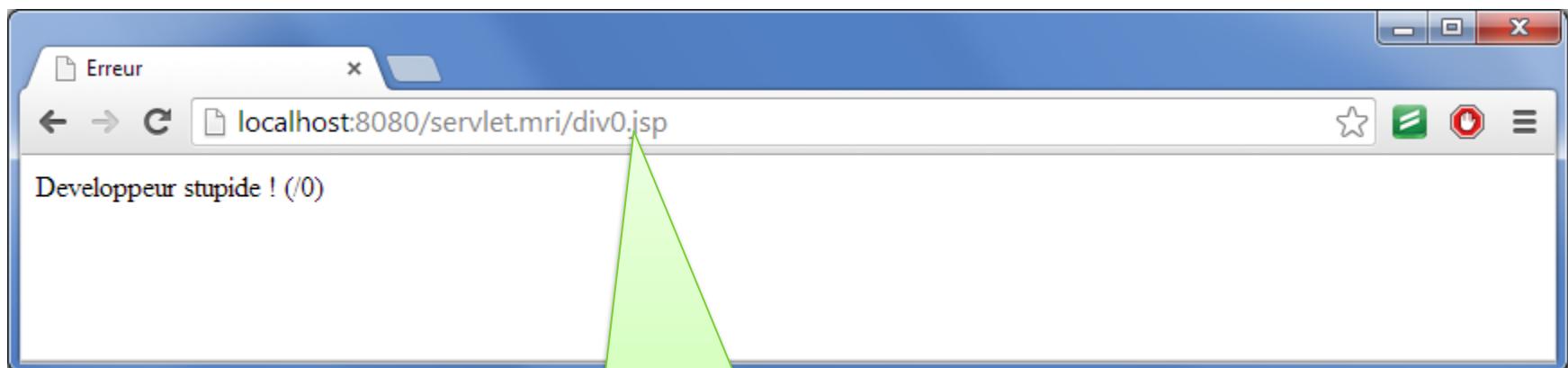
- ▶ Dans la page on déclare une page de traitement d'erreur :

```
<%@ page language="java" errorPage="erreur.jsp"%>
//....
```

- ▶ La page d'erreur.jsp :

```
<%@ page language="java" isErrorPage="true"%>
<!DOCTYPE html>
<html>
<head>
<title>Erreur</title>
</head>
<body>
Developpeur stupide ! (/0)
</body>
</html>
```

Résultat



On remarque que l'adresse n'a pas changée

Inclusion de JSP <%jsp:include

- On peut inclure une page dans une jsp pour répartir les traitements via la directive include :

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

- Exemple on écrit une page main.jsp

```
<%@ page language="java" %>
<!DOCTYPE html >
<html>
<head>
<title>Les racines</title>
</head>
<body>
<jsp:include page="racine.jsp"/>
</body>
</html>
```

On inclus la page racine.jsp

On réécrit la page racine.jsp

- ▶ Elle ne doit plus contenir de balises <html>, <body>, ...
 - ▶ Juste le HTML nécessaire

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
pageEncoding="ISO-8859-1"%>

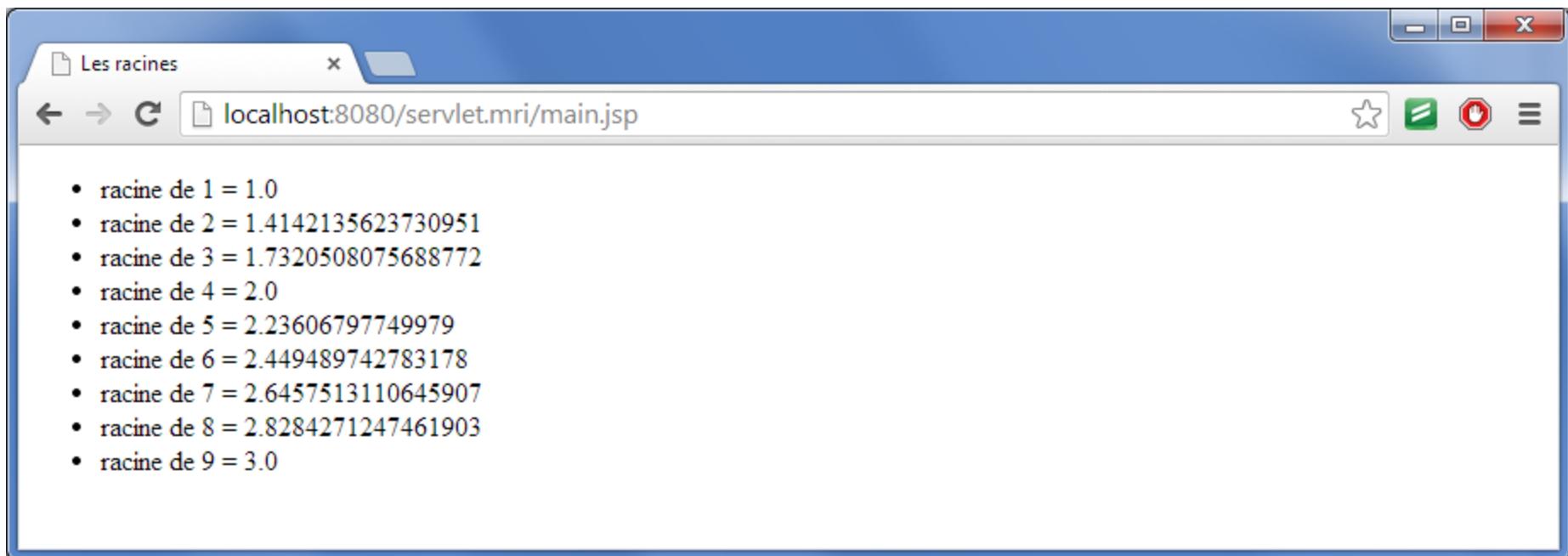
<ul>
<%
for ( int i=1;i < 10; i++ ) {
%>

<li> racine de <%= i %> = <%=Math.sqrt(i)%> </li>

<%
}
%>
</ul>
```

Pas de balise <html>,<head>,<body> ...

Fonctionne comme avant



Délégation de pages

- ▶ On peut aussi déléguer vers une autre pages (faire un renvoi comme avec les dispatcher).
- ▶ On utilise la directive forward :

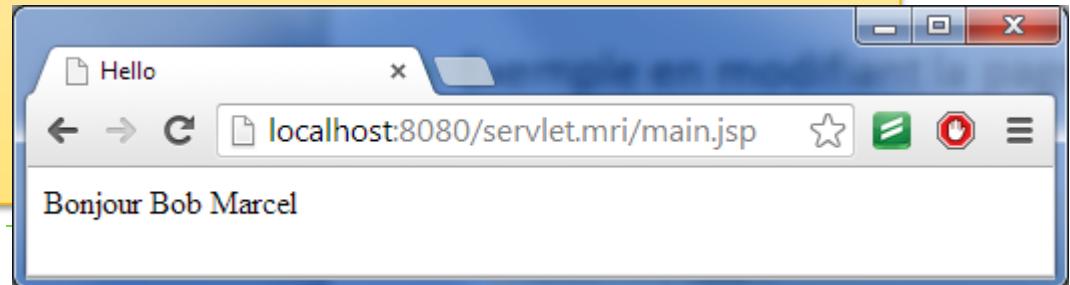
```
<jsp:forward page="uneAutrePage.jsp" />
```

- ▶ Dans ce cas la page destination doit être complète avec <html>,<body>,

On peut passer des paramètres lors d'un forward

- Exemple en modifiant la page main pour rediriger vers hello.jsp

```
<%@ page language="java" %>
<!DOCTYPE html >
<html>
<head>
<title>Youhhouuuu !</title>
</head>
<body>
<jsp:forward page="hello.jsp">
    <jsp:param name="nom" value="Marcel" />
    <jsp:param name="prenom" value="Bob" />
</jsp:forward>
</body>
</html>
```



JSTL :

- ▶ Un ensemble d'extension aux instructions
- ▶ Il faut les importer, par exemple :

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

- ▶ On peut alors les utiliser via le prefix

```
<c:if test="${applicationScope:booklist == null}" >
    <c:import url="/books.xml" var="xml" />
    <x:parse xml="${xml}" var="booklist" scope="application" />
</c:if>
```

Exemple de TagLib

- ▶ jsp (action standard)
- ▶ c (core)
- ▶ fmt (formattage et internationalisation)
- ▶ xml (xml, xpath et xslt)
- ▶ sql (sql)
- ▶ jsf/core, jsf/html, ...
- ▶ logic (Jakarta Struts)
- ▶ tags (TagLib)