

# Servlet et JSP

## Objectifs

version 1.5

## Objectifs

- Connaître la spécification Servlet/JSP
- Connaître le fonctionnement des composants
  - Servlet, JSP, Listener et Filter
- Savoir utiliser les tags personnalisés et l'EL
- Savoir développer la couche de présentation en JSP et Servlet
- Appliquer les bonnes pratiques d'architecture
- Savoir communiquer en Ajax avec JSON
- Savoir gérer l'envoi de fichier vers le serveur



# Chapitres

0. Objectifs
1. Protocole HTTP
2. Application web et Java EE
3. Les servlets
4. Les JSP
5. Cookies et sessions
6. Bonnes pratiques
7. Les listeners
8. Utilisation des POJO et EL dans les JSP
9. Configuration des sources de données
10. Balises personnalisées et JSTL

## Annexes

Annexe A - Les filtres

Annexe B - Envoi de fichiers vers le serveur

Annexe C - Ajax et JSON

Annexe D - Les servlets asynchrones

Annexe E - Les web-sockets

Support de formation créé par

Franck SIMON

<http://www.franck-simon.com>



Cette œuvre est mise à disposition sous licence  
Attribution

Pas d'Utilisation Commerciale

Partage dans les Mêmes Conditions 3.0 France.

Pour voir une copie de cette licence, visitez  
<http://creativecommons.org/licenses/by-nc-sa/3.0/fr/>

ou écrivez à

Creative Commons, 444 Castro Street, Suite 900,  
Mountain View, California, 94041, USA.

# Servlet et JSP

## Protocole HTTP

# HTTP

- HyperText Transfert Protocol
  - permet la communication entre le navigateur et le serveur
  - protocole sans état
    - les paires requête / réponse sont indépendantes les unes des autres
  - protocole textuel
    - les en-têtes des requêtes et des messages circulent en clair entre le navigateur et le serveur

```
GET /PHP/ HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.9.0.7) Gecko/2009021910 Firefox/3.0.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cache-Control: max-age=0
```

# Méthodes HTTP

- GET demande une ressource au serveur
- POST envoi de données au serveur
- HEAD demande l'en-tête d'une ressource au serveur
- PUT envoi d'une ressource au serveur
- OPTIONS permet de connaître les opérations disponibles sur le serveur (GET, POST, ...)
- DELETE suppression d'une ressource sur le serveur
- TRACE demande au serveur de renvoyer la requête reçue

# Codes d'état HTTP

- La réponse du serveur au navigateur comporte un code d'état
- Codes d'état courant
  - 200 OK
  - 300 Moved la ressource a été déplacée
  - 304 Not Modified la ressource n'a pas été modifiée et peut être obtenue par le cache
  - 400 Bad Request syntaxe de requête incorrecte
  - 401 Unauthorized ressource protégée
  - 403 Forbidden ressource non autorisée
  - 404 Not Found
  - 500 Server Error
  - 503 Service Unavailable

# Codes d'état HTTP

- Exemple d'en-tête de réponse HTTP

```
HTTP/1.x 200 OK
Date: Wed, 18 Mar 2009 10:02:57 GMT
Server: Apache/2.2.8 (Win32) PHP/5.2.5 mod_jk/1.2.26
Content-Length: 872
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

## Évolutions de HTTP

- HTTP a été inventé par Tim Berner-Lee
- HTTP/1.0 - RFC 1945 (mai 1996)
- HTTP/1.1 - RFC 2068 (janvier 1997)
- HTTP/2.0 - RFC 7540 (mai 2015)
  - ajoute le protocole SPDY

# Servlet et JSP

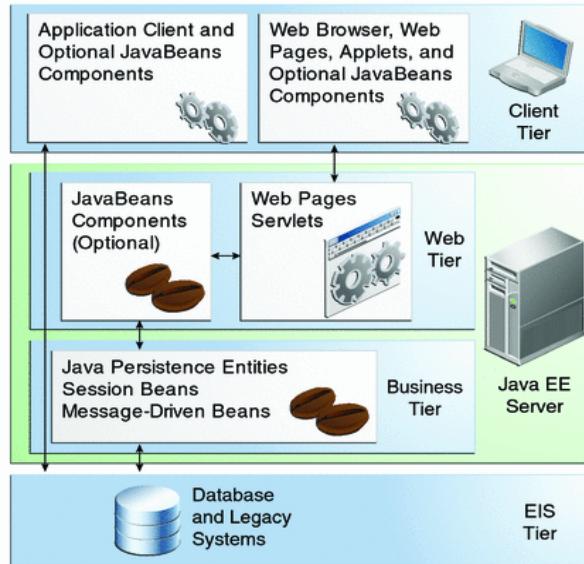
## Application web Java EE

## Introduction

- Java EE favorise une architecture N tiers
  - tiers client sur la machine cliente
    - client léger, ou client lourd (swing)
  - sur le serveur Java EE
    - couche web
    - couche métier
    - couche SI

# Introduction

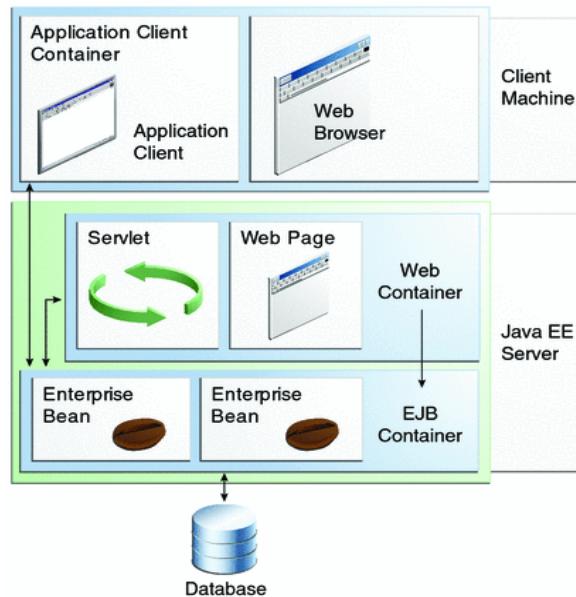
- Application N tiers



## Serveur Java EE

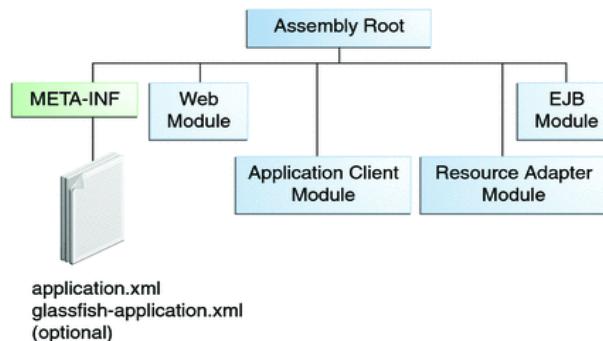
- Un serveur Java EE est constitué de conteneurs
  - conteneur Web
  - conteneur EJB
- Un conteneur gère le cycle de vie des applications qui y sont déployées
  - et donc des objets codés par le développeur
- Le conteneur fournit des services
  - sécurité, recherche JNDI, connectivité, transactions, sources de données, etc,

# Serveur Java EE



## Déploiement des applications Java EE

- Les applications Java EE peuvent être déployées sous forme
  - d'archive web : fichier *.war* (web archive)
  - d'archive jar : fichier *.jar* (java archive)
  - d'archive ear : fichier *.ear* (enterprise archive)

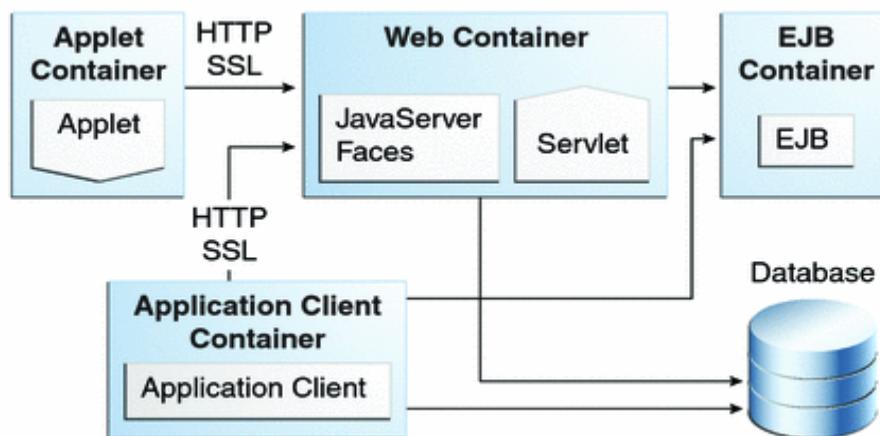


# Équipe Java EE

- Les modules Java EE permettent également une distribution des responsabilités au sein de l'équipe de développement
  - développement de la couche métier
    - les EJB, la couche de persistance
  - développement de la couche web
  - développement des applications swing
  - équipe de déploiement et d'administration des serveurs

## Java EE - les technologies

- Technologies dans les couches applicatives



# Java EE - les technologies

- Couche métier : EJB – Enterprise Java Bean
  - EJB Session : interaction avec le client
    - sans état (stateless) : service métier
      - calcul d'un taux d'intérêt
    - avec état (stateful) : objet métier
      - panier d'achat
      - pas de persistance en base
  - EJB MDB (Message Driven Bean)
    - gestion de la réception de messages en mode asynchrone
    - architecture MOM (Message Oriented Middleware)
      - utilise JMS (Java Message Service)

# Java EE - les technologies

- Couche Web
  - servlet : classe Java invoquée lors des requêtes HTTP
  - JSP (JavaServer Page) : page compilée par le serveur pour répondre aux requêtes HTTP
    - comparable aux technologies PHP, ASP, ...
  - JSF (JavaServer Faces) : framework de construction de la couche web
    - composants graphiques sophistiqués
      - validation des formulaires, gestion des événements, gestion de la navigation entre les pages, ...

# Java EE - les technologies

- Couche de persistance
  - JPA (Java Persistence API)
    - remplace les EJB entités de la version EJB 2
    - utilisable aussi avec JSE
  - liaison déclarative entre le modèle métier et le modèle de données
    - annotations ou fichier XML
  - langage de requêtes orienté objet

# Java EE - les technologies

- Les services
  - transaction : JTA (Java Transaction API)
  - web services
  - injection de ressources
    - JSR 316 : Managed Bean
      - injection de ressources, intercepteurs, méthodes callback sur le cycle de vie
    - JSR 299 : CDI (Context and Dependency Injection)
      - utilisation des EJB dans la technologie JSF
  - Java EE Connector Architecture
    - adaptateurs de ressources pour les SI tiers

# Java EE - les technologies

- Les services

- JavaMail API : service d'envoi de mails
- JAAS (Java Authentication and Authorization Service)
  - gestion de groupes d'utilisateurs, de leur authentification et de leurs droits
- et bien d'autres
  - JMS, JNDI, JDBC, ...

## Serveur d'application Java EE

- Un serveur d'application doit pouvoir
  - supporter une grande nombre d'utilisateurs
  - doit faire face à la montée en charge
    - équilibrage de charge
    - partage des ressources
  - garantir une haute disponibilité
    - gestion de la tolérance aux pannes
    - reprise d'incidents
    - règles des '9' : pourcentage de disponibilité d'un service
      - 99% : service indisponible 3,65 J/an
      - 99,9% : service indisponible 8,75 H/an
      - 99,99% : service indisponible 52 mn/an
      - 99,999% : service indisponible 5,2 mn/an

# Application web

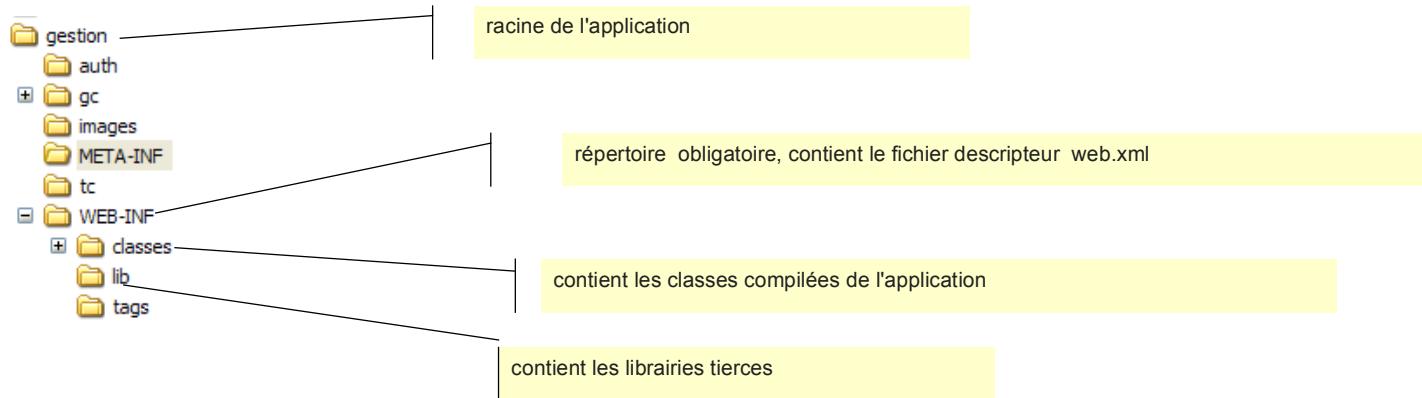
- C'est l'ensemble des ressources nécessaires au bon fonctionnement d'un site Web
  - pages HTML, JSP, classe compilées des servlets, images, fichiers PDF, etc
- Une application Web Java est prise en charge par un conteneur de Servlet/JSP
  - Tomcat est l'implémentation de référence
- L'application Web est déployée sur le serveur
  - le répertoire de déploiement varie selon les serveurs

# Application web

- Une application Web peut être déployée dans une archive
  - WAR (Web Archive)
  - l'archive WAR peut être incluse dans une archive EAR (Enterprise Archive)
    - seulement dans les serveur d'application
      - JBoss, WebSphere, WebLogic, Geronimo, GlassFish, ...
      - pas sous Tomcat

# Application web

- Une application Web Java doit obéir à une structure de répertoire



## Servlet et JSP

### Les servlets

## Premier exemple : HelloServlet

- Une servlet est un classe qui :
  - hérite de la classe `javax.servlet.http.HttpServlet`
  - qui possède des méthodes de réponse aux méthodes HTTP
    - `doGet()`, `doPost()`, `doPut()`, ...

```
public class HelloServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        PrintWriter out = response.getWriter();  
        out.print("<html><head><title>HelloServlet</title></head>");  
        out.print("<body><h2>Hello, world</h2></body></html>");  
    }  
}
```

# Premier exemple : HelloServlet

- La servlet doit être déclarée dans le fichier *web.xml* de l'application web
  - associer la classe de la servlet avec un alias
  - associer des URL avec l'alias de la servlet
    - l'URL doit commencer par /
    - l'appel de la ressource sera
      - `http://hote:port/<nom_application>/<url_pattern>`
      - exemple : <http://localhost:8080>Hello/HelloServlet>
- Depuis la spécification Servlet 3.0 certaines déclarations du fichier *web.xml* peut-être remplacée par des annotations

# Premier exemple : HelloServlet

- Configuration XML dans *web.xml*

```
...
<servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>org.antislashn.web.hello.HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/HelloServlet</url-pattern>
</servlet-mapping>
...
```

- Configuration par annotation

- `javax.servlet.annotation.WebServlet`

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

# Codage de la servlet

- La servlet est une classe Java
  - elle doit être compilée
  - puis déployée dans le répertoire *WEB-INF/classes*
    - Eclipse automatise la compilation et le déploiement
- A chaque requête HTTP la méthode `service(...)` de la servlet est invoquée par le conteneur
  - la méthode `service(...)` invoque ensuite la méthode `doXXXX(...)` correspondant à la requête HTTP
  - le développeur doit coder la (les) méthode(s) `doXXXX(...)`

# Codage de la servlet

- Méthodes HTTP et méthodes invoquées dans la servlet
  - GET            `doGet(...)`
  - POST          `doPost(...)`
  - HEAD          `doHead(...)`
  - OPTIONS      `doOptions(...)`
  - PUT            `doPut(...)`
  - DELETE        `doDelete(...)`
  - TRACE         `doTrace(...)`

# Codage de la servlet

- Les méthodes `doXXXX(...)` ont toutes la même signature de paramètres

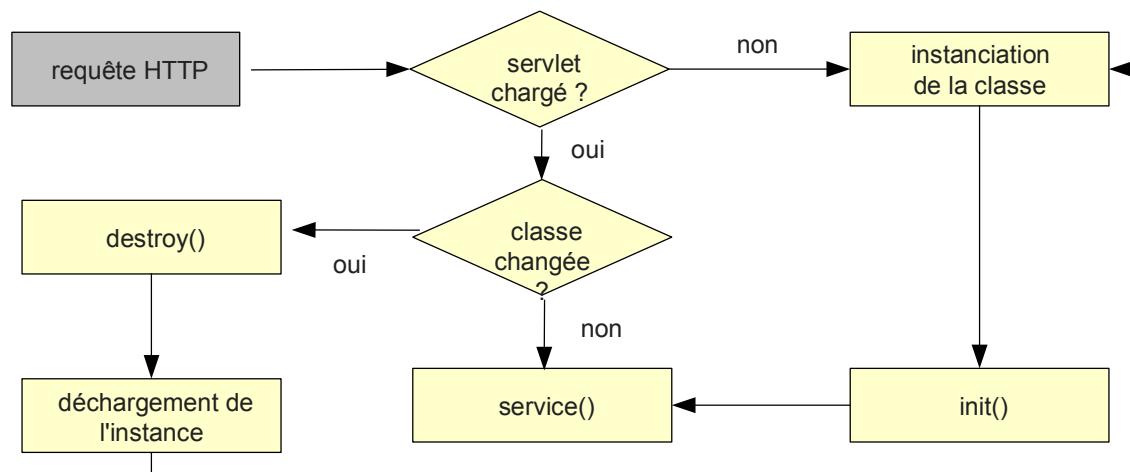
```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

- Les paramètres reçus correspondent aux objets encapsulant la requête et la réponse

- `javax.servlet.http.HttpServletRequest`
  - lecture des paramètres de requête, des données de formulaire, des cookies, ...
- `javax.servlet.http.HttpServletResponse`
  - écriture dans le flux de réponse, ajout de cookies, ajout de paramètres d'en-tête HTTP

## Cycle de vie

- Le conteneur gère le cycle de vie de la servlet
  - instanciation de la classe
  - invocation des méthodes



# HttpServlet

- Dérive de javax.servlet.GenericServlet
  - possède de nombreuses méthodes
    - récupération des paramètres d'initialisation de la servlet
    - récupération du contexte de l'application
      - interface javax.servlet.ServletContext
      - contexte commun à toutes les servlets et JSP de l'application web

## Quelques méthodes de GenericServlet

- getInitParameter (String name)
  - permet de récupérer un paramètre de servlet déclaré dans le *web.xml*, ou par l'annotation

```
<servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>test.servlet.HelloServlet</servlet-class>
    <init-param>
        <param-name>message</param-name>
        <param-value>Hello, world</param-value>
    </init-param>
</servlet>
```

```
@WebServlet(urlPatterns = "/HelloServlet",
    initParams = @WebInitParam(name = "message", value = "Hello, world"))
public class HelloServlet extends HttpServlet {...}
```

- récupération du paramètre

```
String message = getInitParameter("message");
```

# Quelques méthodes de GenericServlet

- `getServletContext ()`
  - renvoie une instance de `javax.servlet.ServletContext`, contexte dans lequel la servlet est exécuté
    - contexte commun à toutes les servlets et JSP => contexte applicatif
  - possède de nombreuses méthodes dont
    - `getInitparameter(String name)` permettant de récupérer des paramètres pour l'application
      - fichier `web.xml`, élément `<context-param>`
    - `getRequestDispatcher(String path)` permettant de récupérer un wrapper de suivi de la requête vers une autre ressource
      - pattern *dispatcher*
      - utilisé dans le pattern *front controller*

## HttpServletRequest - méthodes

- `getParameter(String name)`
  - retourne la valeur d'un paramètre de formulaire ou d'URL
- `getCookies()`
  - retourne le tableau des cookies
- `getHeader(string name)`
  - retourne la valeur d'un paramètre d'en-tête HTTP
- `getSession()`
  - retourne l'instance de la session HTTP courante
- ... cf. la documentation

# HttpServletResponse - méthodes

- `getWriter()`
  - retourne un flux texte vers le navigateur
- `getOutputStream()`
  - retourne un flux binaire vers le navigateur
- `setContentType(String type)`
  - définit le type MIME de la réponse envoyée au client
    - peut-être suivi par le jeu de caractères utilisé  
`text/html; charset=UTF-8`
- `addCookie(Cookie cookie)`
  - ajoute un cookie à la réponse du client
- ... cf. la documentation

## Traitement de formulaire

- `ServletRequest` possède les méthodes permettant de récupérer les informations d'un formulaire
  - `String getParameter(String name)` : renvoie la valeur du champ `name`
  - `String[] getParameterValues(String name)` : renvoie le tableau des valeur d'un champ `name` de type `select, checkbox`
  - `Enumeration<String> getParameterNames()` : renvoie l'ensemble des noms des champs du formulaire sous forme d'énumération

# Traitement de formulaire

projet Eclipse 'Formulaire'

- Exemple - HTML

Nom et prénom

M Votre nom Votre prénom

Langages informatiques

Java  
 C et C++  
 JavaScript  
 PHP

Langues parlées

Français ▲  
Anglais  
Espagnol  
Portugais ▼

Envoyer

```
<select name="civilite" id="civilite">
    <option value="M">M</option>
    <option value="Mme">Mme</option>
    <option value="Mlle">Mlle</option>
</select>
<input type="text" name="nom" id="nom" placeholder="Votre nom" />
<input type="text" name="prenom" id="prenom" placeholder="Votre prénom" />

<input type="checkbox" name="Langages" value="java"/>Java<br/>
<input type="checkbox" name="Langages" value="C"/>C et C++<br/>
<input type="checkbox" name="Langages" value="javascript"/>JavaScript<br/>
<input type="checkbox" name="Langages" value="php"/>PHP<br/>

<select name="Langues" id="Langues" multiple="multiple">
    <option value="french">Français</option>
    <option value="english">Anglais</option>
    <option value="spanish">Espagnol</option>
    <option value="portuguese">Portuguais</option>
    <option value="chinese">Chinois</option>
</select>
```

# Traitement de formulaire

projet Eclipse 'Formulaire'

- Lorsque le formulaire est soumis chaque champs est envoyé au serveur la forme *nom=valeur*
  - envoyé dans l'URL si le formulaire est soumis par GET

```
http://localhost:8080/Formulaire/FormServlet?
civilite=M&nom=SIMON&prenom=Franck&langages=java&langages=php&langues=french&langues=english
```

- envoyé dans le corps de la requête si le formulaire est soumis par POST
- si un champs multiple n'est pas renseigné, il n'est pas envoyé

```
http://localhost:8080/Formulaire/FormServlet?civilite=M&nom=&prenom=
```

# Traitement de formulaire

projet Eclipse 'Formulaire'

- Extraits du code de la servlet

- récupération des valeurs

```
String civilite = request.getParameter("civilite");
String nom = request.getParameter("nom");
String prenom = request.getParameter("prenom");
String[] langages = request.getParameterValues("langages");
String[] langues = request.getParameterValues("langues");
```

- pour les valeurs de type `String []`, vérifier si le tableau existe

```
if(langages==null || langages.length==0){
    out.append("Pas de langages informatiques<br/>");
} else{
    out.append("Langages informatiques : ");
    for(String langage : langages){
        out.append(langage).append(" ");
    }
    out.append("<br/>");
```

# Servlet et JSP

## JSP

## JSP

- Java Server Page
- Utiliser uniquement des servlets implique le mélange de code Java et de code HTML
  - lisibilité réduite
  - maintenance difficile
- Une page JSP est un document texte basé sur du HTML contenant des éléments JSP
  - la base du document peut-être aussi du XHTML, XML, SVG, WML
  - l'extension de la page JSP est *jsp*
- La syntaxe des éléments JSP peut-être une syntaxe standard, ou XML
  - les deux syntaxes ne doivent pas être utilisée dans la même page



# JSP Hello - syntaxe standard

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
        <title>Test JSP</title>
    </head>
    <body>
        <%
            String message = "Hello world";
        %>
        <h2><%=message%></h2>
    </body>
</html>
```

# JSP Hello - syntaxe XML

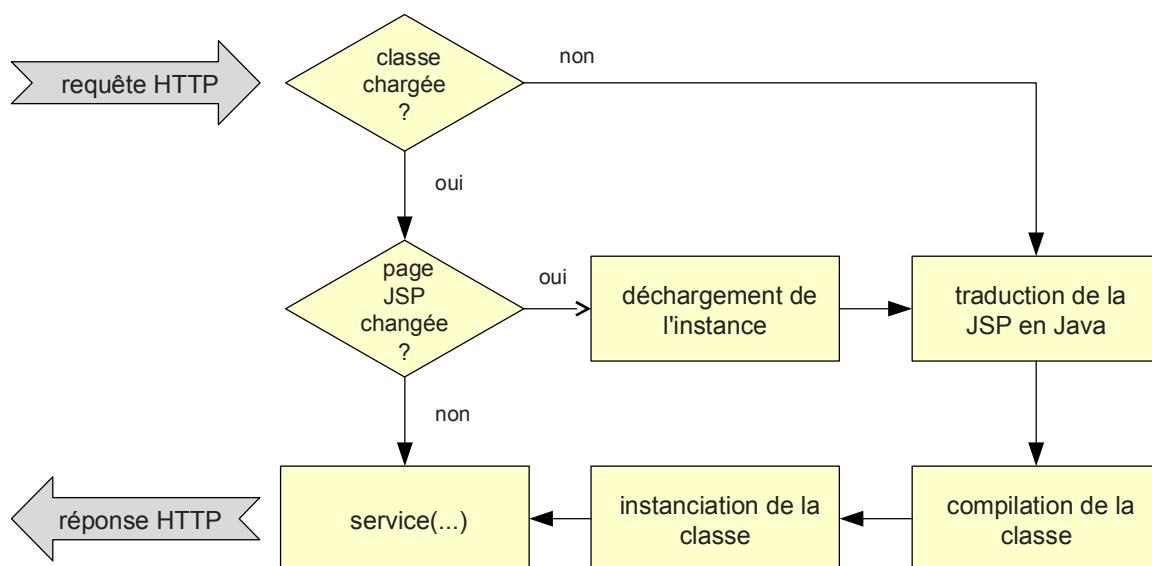
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
    <jsp:directive.page language="java"
        contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" />
    <jsp:text>
        <![CDATA[ <?xml version="1.0" encoding="ISO-8859-1" ?> ]]>
    </jsp:text>
    <jsp:text>
        <![CDATA[ <!DOCTYPE html> ]]>
    </jsp:text>

    <html xmlns="http://www.w3.org/1999/xhtml">
        <head>
            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
            <title>Test JSP</title>
        </head>
        <body>
            <jsp:scriptlet>String message = "bonjour";</jsp:scriptlet>
            <h2><jsp:expression>message</jsp:expression></h2>
        </body>
    </html>
</jsp:root>
```

# Cycle de vie

- Contrairement à un servlet le développeur n'a pas à s'occuper de la compilation et du déploiement d'une page JSP
  - la page JSP est considérée comme une ressource du site, à l'instar des pages HTML
- Le conteneur va gérer le cycle de vie de la page JSP
  - traduction de la page vers un source Java
  - compilation du source
  - instantiation de la classe
  - appel de la méthode `service(...)`

# Cycle de vie



# JSP - éléments de base

- Les éléments JSP permettent d'ajouter au code HTML du code Java qui sera exécuté côté serveur
  - commentaires
  - déclarations
  - expressions
  - scriptlets

# JSP - éléments de base

- Commentaires
  - les commentaires ne sont pas envoyés au navigateur
    - contrairement aux commentaires HTML
  - syntaxe standard : <%-- commentaire --%>
  - syntaxe XML : <!-- commentaire -->

# JSP - éléments de base

- Déclarations

- déclaration de variables ou de méthodes pour la page
- attention : les objets serveurs ne sont pas utilisables dans les déclarations
- les déclarations seront traduites par des propriétés et des méthodes de classe
- syntaxe standard : <%! code java%>
- **syntaxe XML :**  
<jsp:declaration>code java</jsp:declaration>

# JSP - éléments de base

- Expressions

- l'expression est évaluée, puis transformée sous forme de String et envoyée dans le flux de sortie
  - attention : pas de point virgule après l'expression
- **syntaxe standard :** <%= expression %>
- **syntaxe XML :**  
<jsp:expression>expression</jsp:expression>

# JSP - éléments de base

- Scriptlets

- la scriptlet est un fragment de code java
  - sera exécuté côté serveur
- peut utiliser des éléments déclaré
- la scriptlet sera traduite par du code directement dans la méthode service de la classe JSP
- syntaxe standard : <% code java%>
- syntaxe XML :  
<jsp:scriptlet>code java</jsp:scriptlet>

# JSP - éléments de base

```
<%@page import="java.util.Date"%>
...
<body>
<%-- Déclarations --%>
<%!
    Date date = new Date();
    int somme(int...entiers){
        int result = 0;
        for(int i : entiers)
            result+=i;
        return result;
    }
%>

<%-- Expression --%>
Date <%=date %>

<%-- Scriptlet --%>
<%
    int r = somme(1,2,3,4,5);
    out.print("Résultat de la somme : "+r);
%>
</body>
```

# JSP - directives de page

- La directive de page permet de préciser des attributs d'exécution pour la page JSP
  - référez-vous à la documentation pour une liste exhaustive de ces attributs
  - **syntaxe standard** : <%@ page [attribut=valeur] %>
  - **syntaxe XML** :  
<jsp:directive.page [attribut=valeur] />

# JSP - directives de page

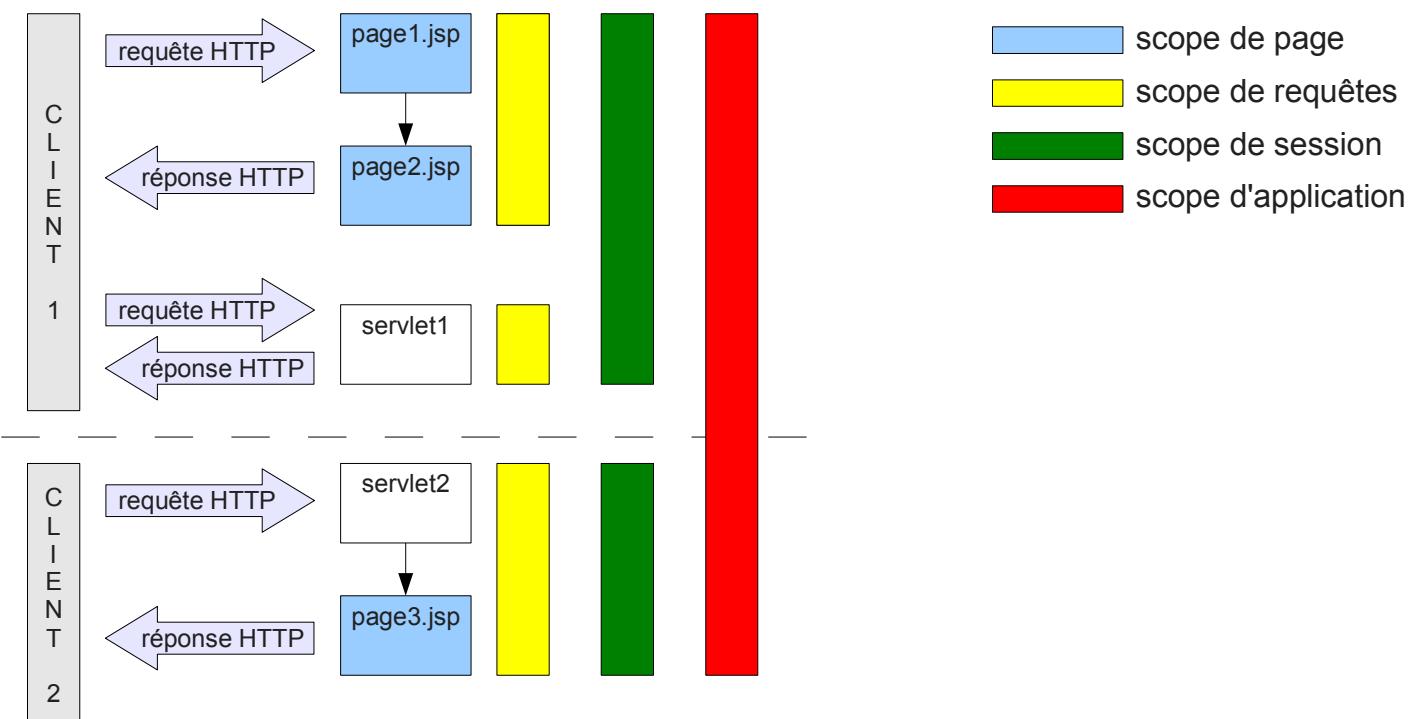
- Attributs courants
  - import : import des package (séparés par une virgule)
  - contentType : type MIME du contenu
  - pageEncoding: encodage des caractères
  - isErrorPage : page JSP d'erreur
  - isELIgnored : langage à expression (EL) activé ou non
  - d'autres directives existent (taglib, tag, ...)
    - seront présentées au fur et à mesure

# JSP - objets implicites

- Le conteneur met à la disposition des scriptlets des objets

| objet       | scope       | utilisation principale                              |
|-------------|-------------|---|
| request     | requête     | même que HttpServletRequest                         |
| response    | page        | même que HttpServletResponse                        |
| pageContext | page        | manipulation d'attributs                            |
| session     | session     | manipulation d'attributs                            |
| application | application | manipulation d'attributs                            |
| out         | page        | flux de sortie                                      |
| config      | page        | récupération de paramètres de configuration         |
| exception   | page        | dans page d'erreur uniquement, instance d'Exception |

## Les scopes



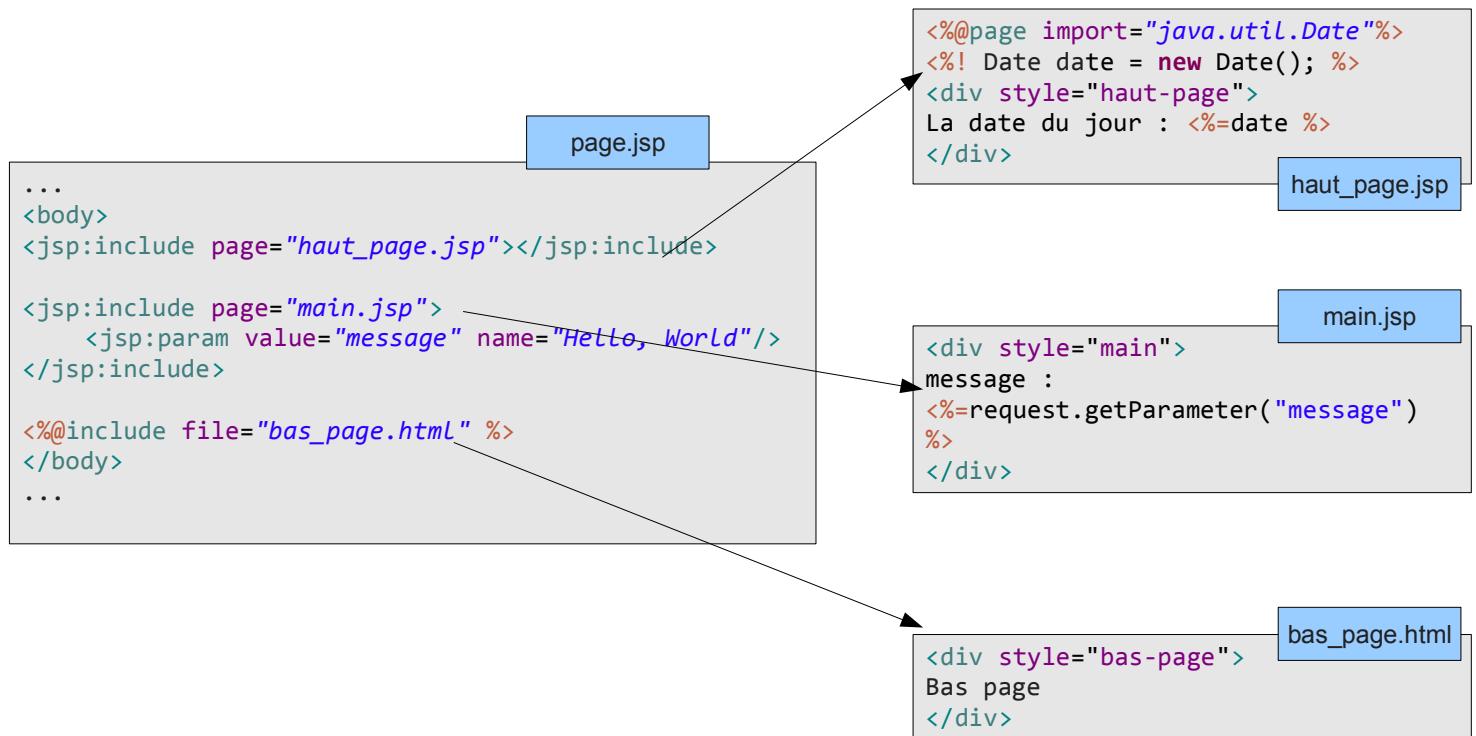
# Inclusion de pages

- Il est fréquent d'avoir des fragments de page JSP ou HTML commun à plusieurs pages JSP
  - hauts et pieds de page par exemple
- Deux manières d'inclure une page dans une autre
  - inclusion statique : le fragment de page inclus est évalué à la traduction
  - inclusion dynamique : le fragment inclus est évalué à la requête
    - possibilité de passer au fragment des paramètres

# Inclusion de pages

- Inclusion statique
  - format standard : `<%@ include file="...." %>`
  - format XML :  
`<jsp:directive.include file="...." />`
- Inclusion dynamique
  - format :  
`<jsp:include page="...."></ jsp:include>`
  - passage de paramètre par
    - `<jsp:param name="...." value="...." />` dans le corps de l'élément `<jsp:include>`

# Inclusion de pages



## Traitement des erreurs

- Lorsqu'une erreur ou exception survient dans une page JSP, l'exception remonte jusqu'au conteneur
  - la trace est alors envoyée dans le flux de sortie
    - pas très utile pour l'internaute
- Le développeur peut créer des blocs `try ... catch` pour gérer les erreurs
  - ce qui rend la JSP particulièrement illisible

# Traitement des erreurs

- Une page peut être déclarée comme page de gestion des erreurs
  - attribut `isErrorHandler="true"` dans la directive de page
  - cette page peut alors utiliser l'objet implicite `exception`
- La page susceptible de lever l'exception doit indiquer la page de traitement des erreurs
  - attribut `errorCode="url relative"` dans la directive de page

# Traitement des erreurs

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8" errorPage="erreur.jsp"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Page générant une erreur</title>
</head>
<body>
<ul>
<%
    String[] tab = {"un", "deux", "trois", "quatre"};
    for(int i=0 ; i<10 ; i++){
        out.print("<li>" +tab[i]+ "</li>");
    }
%>
</ul>
</body>
</html>
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
   pageEncoding="UTF-8" isErrorPage="true"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Erreur</title>
</head>
<body>
<h2>Oups !!! Il est mort, Jim</h2>
<%=exception %>
</body>
</html>
```

# JSP...

- Les JSP recèlent d'autre atouts
  - le EL - Expression Language
  - la possibilité de créer ses propres éléments - les tags
  - la possibilité d'utiliser des bibliothèques de tag
    - les taglibs
  - utilisation dans les patterns MVC
- A suivre dans les prochains chapitres ...



## Servlet et JSP Cookies et sessions

### Cookies

- Les cookies sont des informations textuelles déposées sur le navigateur
  - les cookies sont gérés par le navigateur
  - ils peuvent être refusés par l'utilisateur
  - ne font pas partie de la spécification HTTP
- Un cookie comporte au minimum une clé associée à une valeur
  - la nom de la clé est choisie par le développeur
  - la valeur est calculée ou correspond à une valeur indiquée par l'utilisateur



# Cookies

- Un cookie peut comporter une date de fin de validité
  - si la date est valide, le cookie est sauvegardé par le navigateur
    - cookie "permanent"
  - un cookie sans date n'existe dans la mémoire du navigateur que le temps de navigation sur le domaine
    - cookie de "session"
    - pas de sauvegarde

# Cookies

- Les cookies transitent dans toutes les en-têtes de requêtes et réponse
  - la taille maximale d'un cookie est de 4 Ko
  - une application peut déposer maximum 20 cookies
    - ne pas oublier que si une page comporte 20 images, il y a 21 requêtes de générées par le navigateur
- Un cookie possède d'autres attributs comme
  - indicateur de dépôt par JavaScript
  - indicateur de sécurisation
  - voir la spécification des cookies (RFC 2109) pour plus de détails

# Cookies

- Java EE possède une classe `Cookie` qui modélise un cookie
  - passage de la clé et de la valeur au constructeur
  - méthodes getteur et setteur pour lire et écrire les données du cookies
    - `getValue()`, `setValue(...)`, `getName()`, `setName(...)`

# Cookies

- Les cookies sont manipulés au travers des objets `request` et `response`
  - `request.getCookies()` permet de récupérer un tableau de cookies
- `response.addCookie(Cookie cookie)` ajoute un cookie à l'en-tête de la réponse

```
Cookie[] cookies = request.getCookies();
```

```
response.addCookie(new Cookie("ville",request.getParameter("ville")));
```

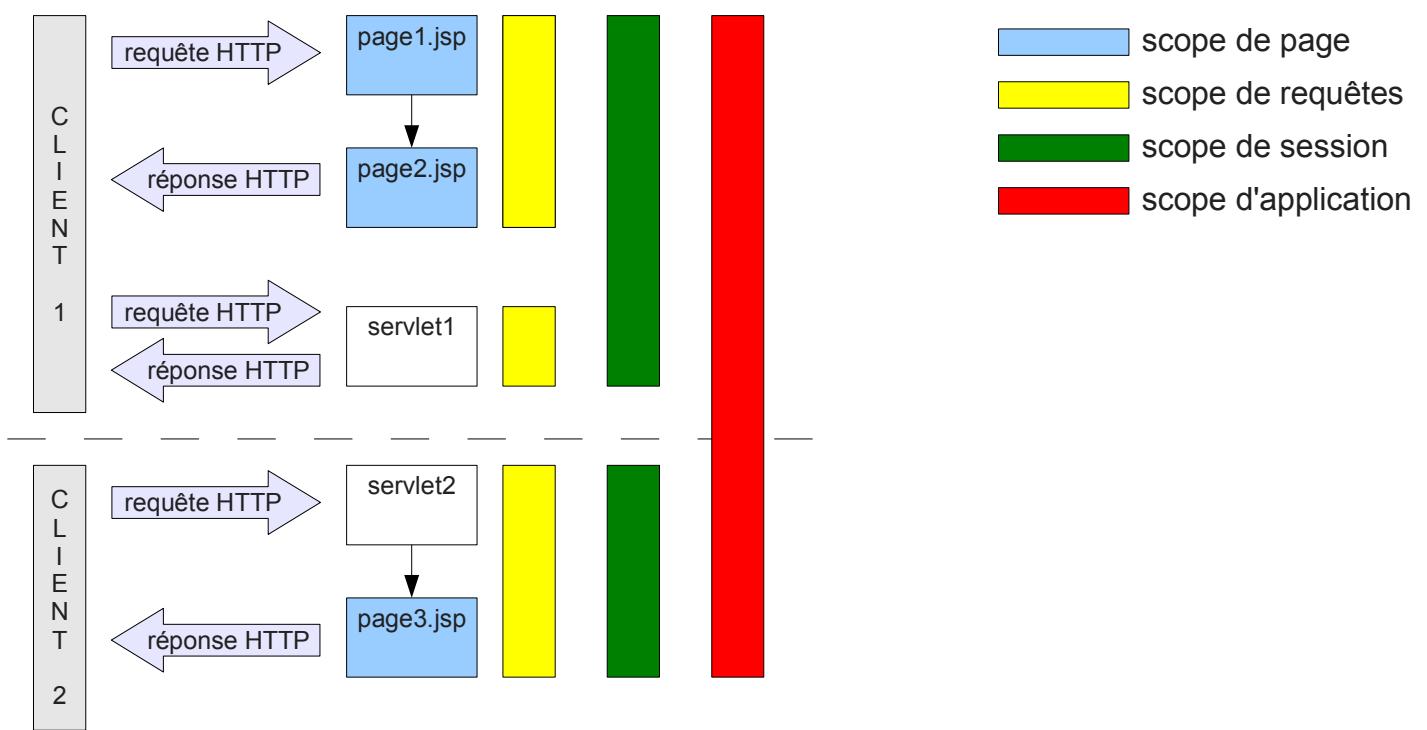
# Sessions

- HTTP est un protocole sans état
  - chaque paire requête - réponse est indépendante des autres
  - pas de suivi de session par le protocole
- Un cookie peut permettre de suivre un utilisateur
  - un nom de cookie est fixé : JSESSIONID
  - la valeur est calculée par le conteneur
    - cette valeur est réputée unique : timestamp + nombre aléatoire
  - le cookie créé n'a pas de date de validité fixée
    - il ne sera pas sauvegardé par le navigateur en fin de navigation

# Sessions

- Un contexte de session est créé au sein du conteneur
  - le contexte est lié à un client via le contenu du cookie de session
  - la classe `HttpSession` modélise la session HTTP
- D'autres contextes existent au sein du conteneur
  - le contexte de page (pour les JSP)
  - le contexte de requête
  - le contexte de session
  - le contexte d'application (`ServletContext`)

# Les scopes



## Sessions

- Sur ces contextes nous pouvons gérer des attributs
  - mécanisme central pour la gestion des sites Web
  - méthodes de l'instance représentant le contexte :
    - `getAttribute()`
    - `setAttribute(Object object)`
    - ne pas confondre avec `request.getParameter()`
  - obligation de transposer le résultat retourné par la méthode `getAttribute()`
- ATTENTION - pour les clusters de serveurs
  - toujours utiliser `setAttribute()` pour propager la réPLICATION de session

# Sessions

- Construction des sessions
  - automatique lors de l'appel des JSP
    - sauf si directive de page contraire
  - appel de la méthode `request.getSession()` dans les servlets

# Sessions

- Destruction d'une session
  - par `session.invalidate()`
    - doit être lié à une action de l'utilisateur
      - bouton "Déconnexion"
  - par la fin du temps de session (timeout de session)
    - valeur par défaut définie dans le `conf/web.xml` du serveur
    - peut-être adapté pour chaque application dans le `web.xml` de l'application (temps en minutes)
    - peut-être adapté par programmation par la méthode `session.setMaxInactiveInterval(int secondes)`

# Les contextes

- Contexte de page
  - uniquement pour les JSP
  - <jsp:forward ...> permet de faire suivre une requête sur une autre page JSP, les contextes de page seront différents
- Contexte de requête
  - les attributs définis sont conservés pour toute la requête
  - une même requête est partagée par les pages appelées par <jsp:forward ...>

# Les contextes

- Contexte de session
  - les attributs définis sont partagés par les JSP et servlets d'une même session
- Contexte d'application
  - les attributs définis sont partagés par toutes les servlets et JSP
- Différents listeners permettent d'être à l'écoute des événements survenant sur certains contextes
  - cf. plus loin

# Servlet et JSP

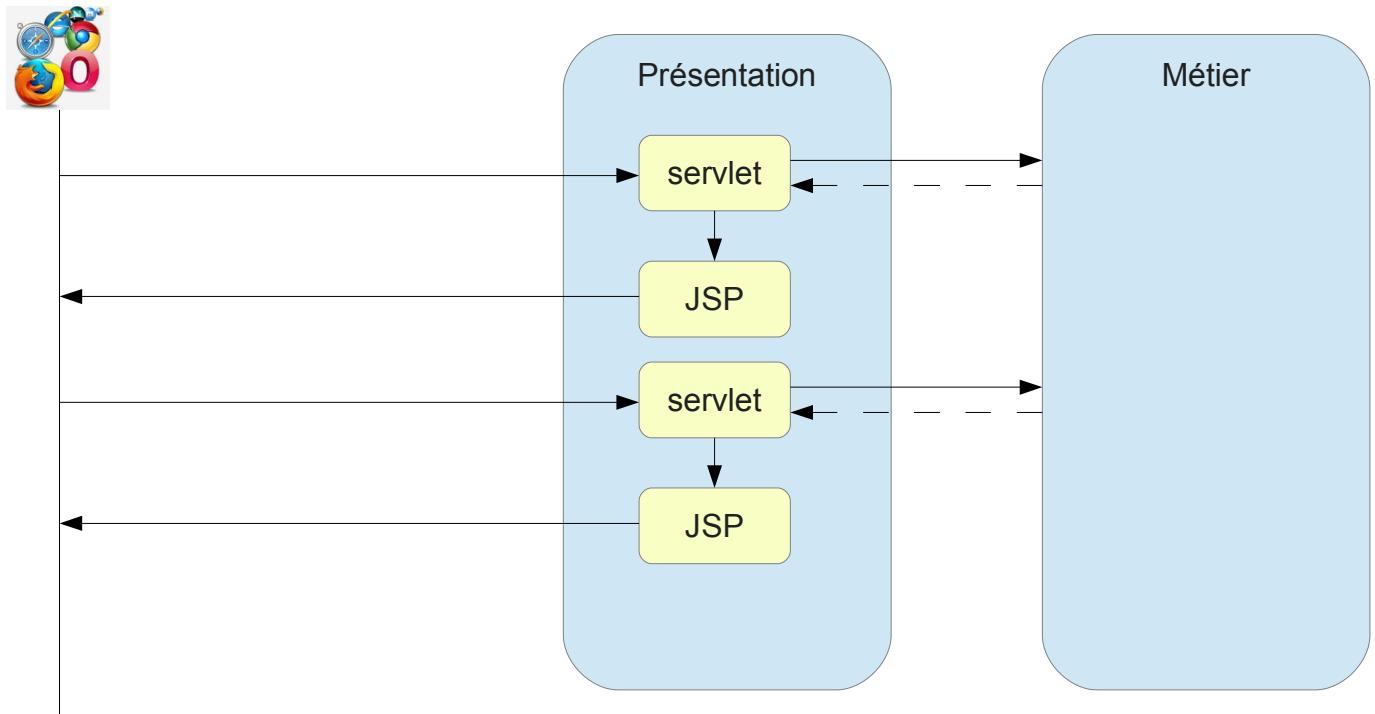
## Quelques bonnes pratiques

## Servlet Controller

- Une première bonne pratique est de séparer les traitements de la vue
  - la servlet
    - reçoit la requête
    - exécute un traitement auprès de la couche métier
    - met à disposition de la vue le résultat de ce straitement
  - la JSP
    - récupère le résultat du traitement
    - génère la vue appropriée
- Architecture MVC (Model - View - Controller)



# Servlet Controller



# Servlet Controller

- Comment faire suivre les requêtes au sein du serveur
  - il faut maintenir l'état de la requête, de la réponse, de la session vers une autre ressource
    - dans notre cas : de la servlet vers la JSP
- Il faut un routeur de requête / réponse vers la ressource
- L'interface RequestDispatcher joue le rôle de routeur
  - l'objet RequestDispatcher est créé par le conteneur

# Servlet Controller

- Utilisation du RequestDispatcher
  - récupération d'un RequestDispatcher
    - sur ServletContext ou ServletRequest
    - méthode `getServletDispatcher(String resource)`
      - ressource est le nom de la ressource cible
      - doit commencer par "/"
  - utilisation des méthodes du RequestDispatcher
    - `include(...)`
    - ou `forward(...)`
    - paramètres de ces deux méthodes :
      - ServletRequest et ServletResponse

# Servlet Controller

- Exemple de mise en œuvre du RequestDispatcher

```
@WebServlet("/TraitementServlet1")
public class TraitementServlet1 extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String message = "traitement 1";
        String page = "/accueil.jsp";
        request.setAttribute("message", message);
        RequestDispatcher rd = this.getServletContext().getRequestDispatcher(page);
        rd.forward(request, response);
    }
}
```

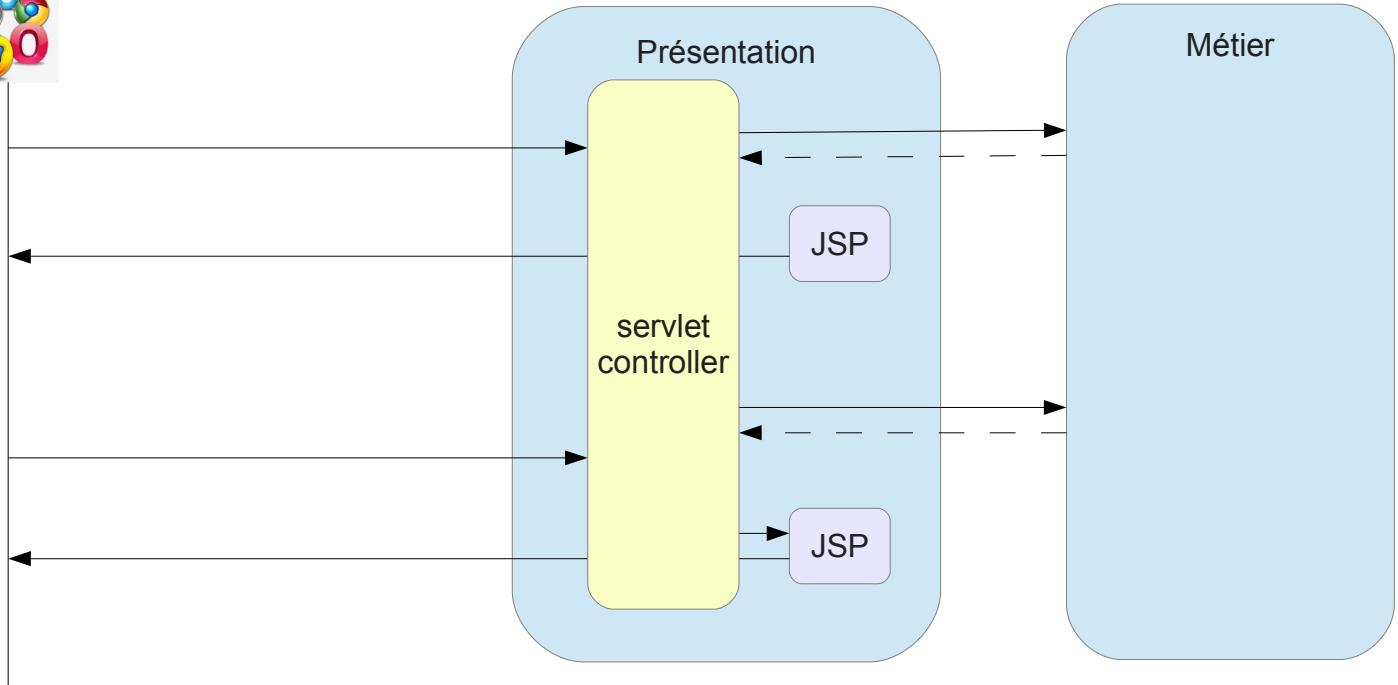
# Servlet Controller

- Bénéfices
  - la servlet ne s'occupe que des traitement
    - pas de génération de HTML dans la servlet
  - la JSP ne s'occupe que de générer la vue
    - pas de code Java de traitement métier
  - meilleure maintenabilité
  - pas de couplage entre la vue et les traitements
- Défaut
  - le suivi du flux applicatif est compliqué à suivre
    - les servlets servent de point d'entrée

# Front Controller

- L'interaction de l'utilisateur avec l'application web est effectuée au travers de nombreuses requêtes
- L'utilisation de multiples contrôleurs nuit au suivi du flux applicatif
- Duplication de code entre les contrôleurs
- Solution : un point d'entrée unique pour toutes les requêtes

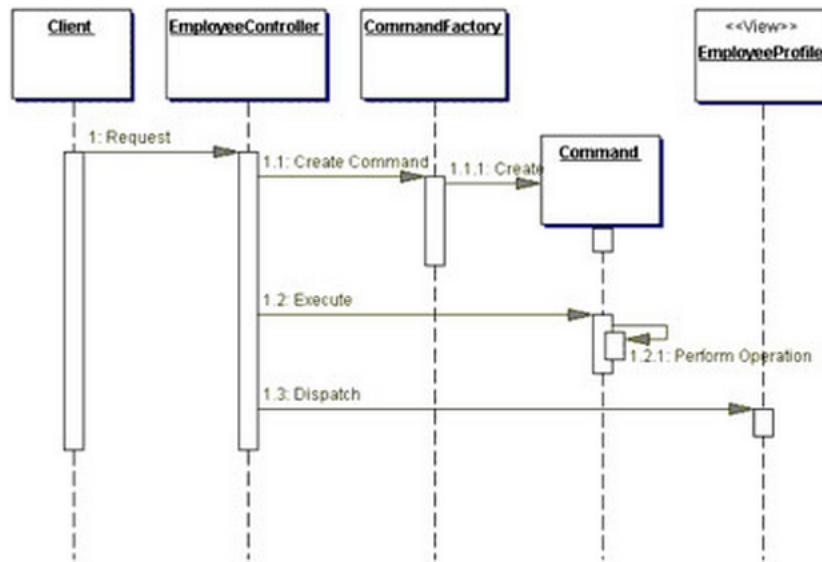
# Front Controller



# Front Controller

- Plusieurs stratégies existent
- Une stratégie récurrente est l'utilisation de "Command and Controller"
  - basé sur le pattern "Command" du GoF
  - une servlet comme FrontController
  - une interface Command utilisée par le contrôleur
    - Command est le nom donné par le GoF
    - de nombreux framework utilise une interface nommée Action
  - l'interface possède une méthode retournant la vue à afficher

# Front Controller



source : Oracle

# Front Controller

- Mise en œuvre
  - pour que le contrôleur puisse différencier les requêtes entre-elles il faut mettre en place un discriminateur
    - ajouter un paramètre dans l'URL des requêtes par exemple
  - il faut s'assurer que toutes les requêtes soient bien envoyées vers le FrontController

# Front Controller

- Exemple de servlet Front Controller

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String cde = request.getParameter("cde");
        if(cde==null)
            cde="accueil";
        String page = null;
        Action action = ActionFactory.getAction(cde);
        page = action.compute(request, response);
        RequestDispatcher rd = this.getServletContext().getRequestDispatcher(page);
        rd.forward(request, response);
    }
}
```

# Front Controller

- Exemple d'interface Action et d'une implémentation
  - pattern Command du GoF

```
public interface Action {
    String compute(HttpServletRequest request, HttpServletResponse response);
}
```

```
public class Traitement1 implements Action {
    @Override
    public String compute(HttpServletRequest request, HttpServletResponse response) {
        // Faire un traitement util ici
        String message = "traitement 1";
        request.setAttribute("message", message);
        return "/accueil.jsp";
    }
}
```

# Front Controller

- Exemple de Factory

```
public class ActionFactory {  
    public static Action getAction(String cde) {  
        switch(cde){  
            case "accueil":  
                return new Traitement1();  
            case "t2":  
                return new Traitement2();  
            ...  
        }  
        return null;  
    }  
}
```



## Servlet et JSP Les Listeners

### Listener

- Une instance de listener est à l'écoute d'événements
  - lancement et replis de l'application
  - début et fin de session
  - début et fin de requête
  - ajout et suppression d'objet dans un contexte de session
- Le conteneur de servlet invoque une méthode d'un listener lorsque l'événement se produit
  - un objet correspondant à l'événement est passé à la méthode

# Listener

- Développement d'un listener
  - implémentation de l'interface adéquate
  - enregistrement du listener auprès du conteneur de servlet
    - par l'annotation de classe `@WebListener`
    - ou par déclaration dans le fichier `web.xml`

```
<listener>
    <listener-class>org.antislashn.web.ApplicationListener</listener-class>
</listener>
```

## Types de Listener

- gestion du cycle de vie d'un contexte
  - application (`ServletContext`)
  - session (`HttpSession`)
  - requête (`ServletRequest`)
- ajout ou retrait d'attribut dans un contexte
  - méthodes `setAttribute()` et `removeAttribute()`
- gestion des sessions
  - migration, changement d'identifiant

# ServletContextListener

- Cycle de vie du contexte de l'application
  - la méthode `contextInitialized()` est appelée avant tout filtre ou servlet
  - `ServletContextEvent` possède une méthode : `getServletContext()`

```
@WebListener
public class ApplicationListener implements ServletContextListener {
    private static Logger LOGGER = Logger.getLogger("Projet Listener - 01");

    public void contextInitialized(ServletContextEvent evt) {
        LOGGER.info(">>> Contexte applicatif initialisé");
    }

    public void contextDestroyed(ServletContextEvent evt) {
        LOGGER.info(">>> Contexte applicatif détruit");
    }
}
```

# Cycle de vie de la session

- Création et destruction de la session
  - interface `HttpSessionListener`
  - la session est créée
    - première requête sur une JSP
    - premier appel de la méthode `getSession()` dans une servlet
  - la session est détruite
    - lors de l'arrivée du timeout de session
    - lors de l'appel de la méthode `invalidate()` sur `HttpSession`

# Cycle de vie de la session

- Le paramètre HttpSessionEvent possède une méthode getSession()
  - permet l'ajout d'objet de session
    - Caddy, ...

```
@WebListener
public class SessionListerner implements HttpSessionListener {
    private static Logger LOGGER = Logger.getLogger("Projet Listener - 01");

    public void sessionCreated(HttpSessionEvent evt)  {
        LOGGER.info(">>> Début de session : "+evt.getSession().getId());
    }

    public void sessionDestroyed(HttpSessionEvent evt)  {
        LOGGER.info(">>> Fin de session : "+evt.getSession().getId());
    }
}
```

# Cycle de vie de la session

- Passivation et activation de la session
  - interface HttpSessionActivationListener
  - passivation :
    - la session peut être sérialisée par le conteneur
    - la session peut passer d'un conteneur à un autre
      - cluster de serveurs
    - méthode sessionWillPassivate()
  - activation
    - dé sérialisation de la session
    - prise en charge par un nœud de cluster
    - méthode sessionDidActivate()

# Cycle de vie de la session

- Afin de lutter contre les attaques de type "*cross-site scripting*" ou "*session fixation*" l'identifiant de session peut être régénéré
  - si un utilisateur s'authentifie, et que son identifiant de session est volé, l'attaquant possède alors les mêmes droits que l'utilisateur authentifié
  - l'identifiant de session est régénéré après l'identification

# Cycle de vie de la session

- Interface `HttpSessionIdListener`
  - une seule méthode : `sessionIdChanged()`
    - deux paramètres
      - `HttpSessionEvent`
      - `String` : ancien identifiant de session

# Cycle de vie de la requête

- **Interface** ServletRequestListener
  - création du contexte
    - méthode contextInitialized()
  - destruction du contexte
    - méthode contextDestroyed()
  - paramètre des méthodes : ServletContextEvent

# Cycle de vie de la requête

- Lorsque la requête gère des traitement asynchrone il peut être nécessaire d'écouter le cycle de vie du traitement
  - @WebServlet(asyncSupported=true)
- **Interface** AsyncListener, méthodes
  - onComplete()
  - onError()
  - onStartAsync()
  - onTimeout()
  - paramètre de type AsyncEvent

# Listeners sur les attributs

- Trois interfaces, en fonction des contextes
  - ServletRequestAttributeListener
  - HttpSessionAttributeListener
  - ServletContextAttributeListener
- Les trois interfaces ont trois méthodes de même nom
  - mais signatures différente (type de l'événement)
  - attributeAdded() : ajout d'un objet
  - attributeRemoved() : suppression d'un objet
  - attributeReplaced() : l'attribut est remplacé

# Listeners sur les attributs

- Les trois méthodes précédentes reçoivent un paramètre de type différent en fonction du contexte
  - ServletRequestAttributeEvent
  - HttpSessionBindingEvent
  - ServletContextAttributeEvent
- Méthodes communes
  - getName() : nom de l'objet ajouté ou retiré du contexte
  - getValue() : valeur de l'objet ajouté ou retiré
- HttpSessionBindingEvent possède une méthode getSession()

# Listeners sur les attributs

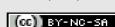
- Un objet souhaitant être prévenu lorsqu'il est ajouté ou retiré d'un contexte peut implémenter l'interface HttpSessionBindingListener
  - méthodes
    - valueBound()
    - valueUnbound()
  - paramètre de type HttpSessionBindingEvent



## Servlet et JSP JSP, POJO et EL

# Utilisation des JavaBeans

- JavaBean est une spécification de développement de classe
  - spécification de base
    - possède une constructeur par défaut
    - implémente Serializable
    - propriétés privées
    - accès aux propriétés par getteurs / setteurs
  - spécification des POJO
    - Plain Old Java Object



# JavaBean

```
public class Contact implements Serializable {
    private String civilite;
    private String nom;
    private String prenom;

    public Contact() {}

    public String getCivilite() {
        return civilite;
    }

    public void setCivilite(String civilite) {
        this.civilite = civilite;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}
```

## JSP et JavaBean

- Les éléments JSP facilitent la manipulation des JavaBeans
- Création d'une instance de bean
  - syntaxe standard et XML : <jsp:useBean>
  - attributs utilisés
    - id nom unique de l'instance du bean
    - class nom de la classe (avec le package)
    - scope contexte de vie du bean  
(page | request | session | application)
  - si l'instance du bean existe déjà dans le contexte le bean n'est pas recréé mais réutilisé.



# JSP et JavaBean

- Déclaration du JavaBean

```
<jsp:useBean id="compteur"  
class="org.antislashn.web.Compteur" />
```

- Visualisation d'une propriété

- syntaxe standard et XML : <jsp:getProperty>
- effectue un appel vers une méthode `getXXX()` et envoie le résultat dans le flux de sortie
- attributs utilisés
  - `name` nom de l'instance du bean (id de <jsp:useBean>)
  - `property` nom de la propriété

```
<jsp:getProperty name="compteur" property="valeur" />
```

# JSP et JavaBean

- Initialisation d'une propriété

- syntaxe standard et XML : <jsp:setProperty>
- effectue un appel vers une méthode `setXXX()`
- attributs utilisés
  - `name` nom de l'instance du bean (id de <jsp:useBean>)
  - `property` nom de la propriété
  - `value` valeur à affecter

```
<jsp:setProperty property="valeur" name="compteur" value="1" />
```

# JSP et JavaBean

- Initialisation d'une collection de propriétés avec un formulaire
  - souvent les beans sont liés aux valeurs entrées par l'utilisateur dans un formulaire
  - le conteneur peut explorer les paramètres de la requête pour appeler les getteurs
  - les nom des champs du formulaires doivent correspondre aux nom des propriétés
  - syntaxe standard et XML : <jsp:setProperty>
  - attributs utilisés
    - name nom de l'instance du bean (id de <jsp:useBean>)
    - property contient la valeur "\*"

# JSP et JavaBean

Civilité  M

Nom

Prenom



```
<jsp:useBean id="personne"
              class="org.antislashn.web.pojo.Personne"
              scope="session">
    <jsp:setProperty name="personne" property="*" />
</jsp:useBean>
```

- si <jsp:setProperty> se trouve en dehors du corps du <jsp:useBean> la mise à jour des propriétés sera effectuée à chaque requête
- si le noms des champs ne correspondent pas aux nom des propriétés il est possible d'utilisé l'attribut param
- une propriété qui n'est pas liée à un champ n'est pas initialisée

# EL - Introduction

- EL - Expression Language - permet de simplifier la manipulation des expressions dans les pages JSP
  - permet d'accéder simplement aux POJO des différents contextes
- Forme générale d'une EL : `$ {expression}`
  - l'expression est évaluée, puis envoyée vers le flux de sortie

## EL - introduction

- Une expression permet d'accéder simplement aux propriétés d'un POJO
  - `$ {objet.propriete}` ou `$ {objet[1]}`
- Une expression peut-être composée de plusieurs termes séparés par des opérateurs
  - `$ {expression1 OPERATEUR expression2}`

# JSP et EL

- Les EL sont utilisables depuis JSP 2.0
  - avec JSTL 1.0 sous certaines conditions (cf. chapitres suivants)
  - dans les JSP (fichiers *.jsp* et *.tag*)
  - dans des valeurs d'attributs de tags personnalisés

```
 ${compteur.valeur }
```

```
<c:forEach items="${destination.images }" var="img">
    <div></div>
</c:forEach>
```

- L'interprétation des EL peut-être désactivée
  - attribut `isELIgnored` de la directive de page

# JSP et EL

- Types manipulés par EL
  - types primitifs, via leur wrapper
  - `String`
  - valeurs prédéfinies
    - `null`
    - `true`
    - `false`

# JSP et EL

- Objets implicites permettant d'accéder aux composants d'une page

| objets           | descriptifs                                |
|------------------|--|
| pageContext      | objet PageContext de la JSP                |
| pageScope        | contexte de page                           |
| requestScope     | contexte de requête                        |
| sessionScope     | contexte de session                        |
| applicationScope | contexte d'application                     |
| param            | paramètre de la requête HTTP (String)      |
| paramValues      | paramètres de la requêtes HHTP (String[])  |
| header           | valeur du header HTTP (String)             |
| headerValues     | valeurs du header HTTP (String[])          |
| cookie           | accès aux cookies                          |
| initParam        | accès aux paramètres init-param du web.xml |

# JSP et EL

- Opérateurs arithmétiques

|          |                |
|----------|----------------|
| +        | addition       |
| -        | soustraction   |
| *        | multiplication |
| / ou div | division       |
| % ou mod | modulo         |

# JSP et EL

- Opérateurs relationnels

|                          |                   |
|--------------------------|-------------------|
| <code>== ou eq</code>    | égalité           |
| <code>!= ou ne</code>    | inégalité         |
| <code>&lt; ou lt</code>  | inférieur         |
| <code>&gt; ou gt</code>  | supérieur         |
| <code>&lt;= ou le</code> | inférieur ou égal |
| <code>&gt;= ou ge</code> | supérieur ou égal |

# JSP et EL

- Opérateurs logiques

|                                |                      |
|--------------------------------|----------------------|
| <code>&amp;&amp; ou and</code> | ET logique           |
| <code>   ou or</code>          | OU logique           |
| <code>! ou not</code>          | NON logique (unaire) |

- autres

|                    |   |
|--------------------|---|
| <code>empty</code> | vrai si null, chaîne vide, tableau vide, ... (unaire) |
| <code>()</code>    | modification de la préséance des opérateurs           |
| <code>?:</code>    | condition (ternaire)                                  |

# JSP et EL

- L'accès aux propriétés est effectué par réflexivité

- via les méthodes `getXxx`

- accès par point (.) ou crochets ([...])

```
 ${compteur.valeur }  
 ${compteur['valeur'] }
```

- si le scope du POJO n'est pas précisé, la recherche part du contexte de page vers le contexte d'application
- les propriétés peuvent être imbriquées

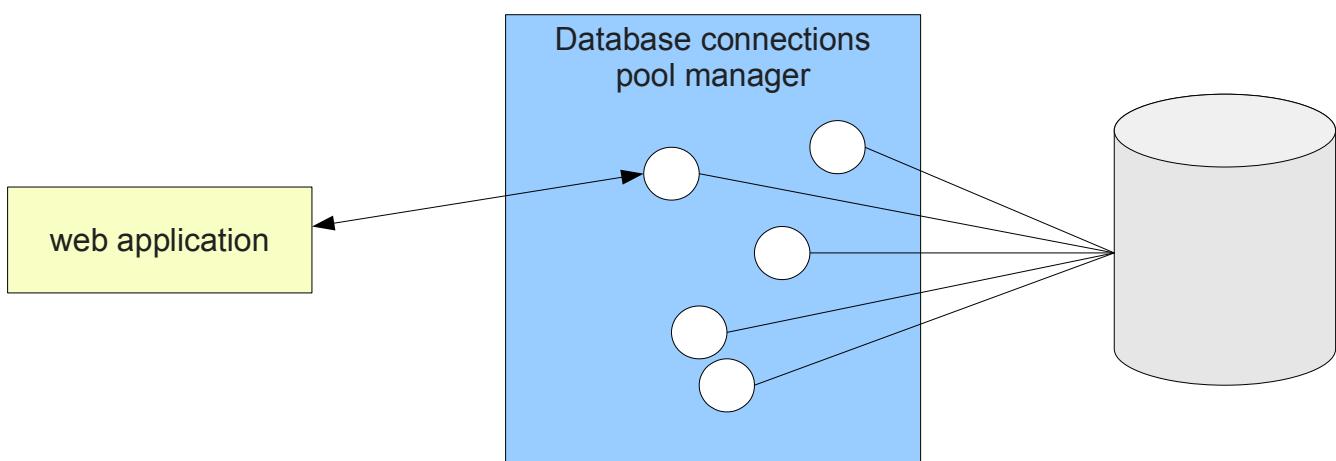
```
 ${contact.adresses[0].rue}
```

# Servlet et JSP

## Sources de données

## Sources de données

- Pool de connexions à la source de données
  - interface `javax.sql.DataSource`



# Sources de données

- Le gestionnaire de connexions
  - crée les connexions physiques à la base de données
  - gère la distribution des connexions physiques
    - l'application web récupère une connexion logique
    - la fermeture de la connexion logique ne ferme pas la connexion physique
  - gère les exceptions et erreurs
    - timeout, ...
- Le gestionnaire connexions est fourni par le serveur
  - Tomcat, JBos, Geronimo, GlassFish, ...

## Tomcat - configuration d'une DataSource

- Configuration via une ressource JNDI
  - Java Naming and Directory Interface
  - dans l'élément <context>
    - la DataSource est accessible uniquement par l'application web de ce contexte
  - dans l'élément <GlobalNamingResources> du fichier *server.xml*
    - la DataSource est alors disponible pour toutes les applications déployées

# Tomcat - configuration d'une DataSource

- Configuration JNDI pour MySql

```
<Context reloadable="true">
  <Resource
    name='jdbc/bovolage'
    auth='Container'
    type='javax.sql.DataSource'
    driverClassName='com.mysql.jdbc.Driver'
    url='jdbc:mysql://bovolage'
    username='toto'
    password='totopw'
    maxActive='20'
    maxIdle='10'
    maxWait='10000'
    removeAbandoned='true' />
</Context>
```

# Tomcat - configuration d'une DataSource

- Attributs principaux
  - name : le nom de la ressource telle qu'elle sera utilisée dans l'application web
  - auth : configure le type d'authentification auprès de la base de données
    - "Container" par le serveur
    - "Application" par l'application
  - type : type de fabrique
  - driverClassName : nom du driver fournit par l'éditeur de la base de données

# Tomcat - configuration d'une DataSource

- Attributs principaux
  - url : URL de connexion à la base de données
  - username et password : identifiants de connexion à la base de données
  - validationQuery : requête exécutée avant chaque distribution d'une connexion logique, pour vérifier la disponibilité de la connexion physique
  - maxActive : nombre maximum de connexions simultanées
  - maxIdle : nombre maximum de connexions à maintenir en permanence dans le pool

## Tomcat - accès à la DataSource

- La source de données est accessible via JNDI
- Une référence vers la source de données est configurée dans le fichier *web.xml*

```
<resource-ref>
    <res-ref-name>jdbc/bovoyage</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

# Récupérer la source de donnée

- Via JNDI

- le nom de la source de données est relatif au contexte, qui est défini par *java:comp/env*
- le nom de la ressource est *jdbc/bovoyage*
- le nom complet d'accès à la ressource sera *java:comp/env/jdbc/bovoyage*

```
Context contexteJndi = new InitialContext();
DataSource dataSource = (DataSource) contexteJndi.lookup("java:comp/env/jdbc/bovoyage");
```

# Récupérer la source de donnée

- Via l'injection de ressource

- directement dans une propriété d'un composant pris en charge par le conteneur

```
public class JndiServlet extends HttpServlet {
    @Resource(name="jdbc/bovoyage") DataSource ds;
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    ...
}
```

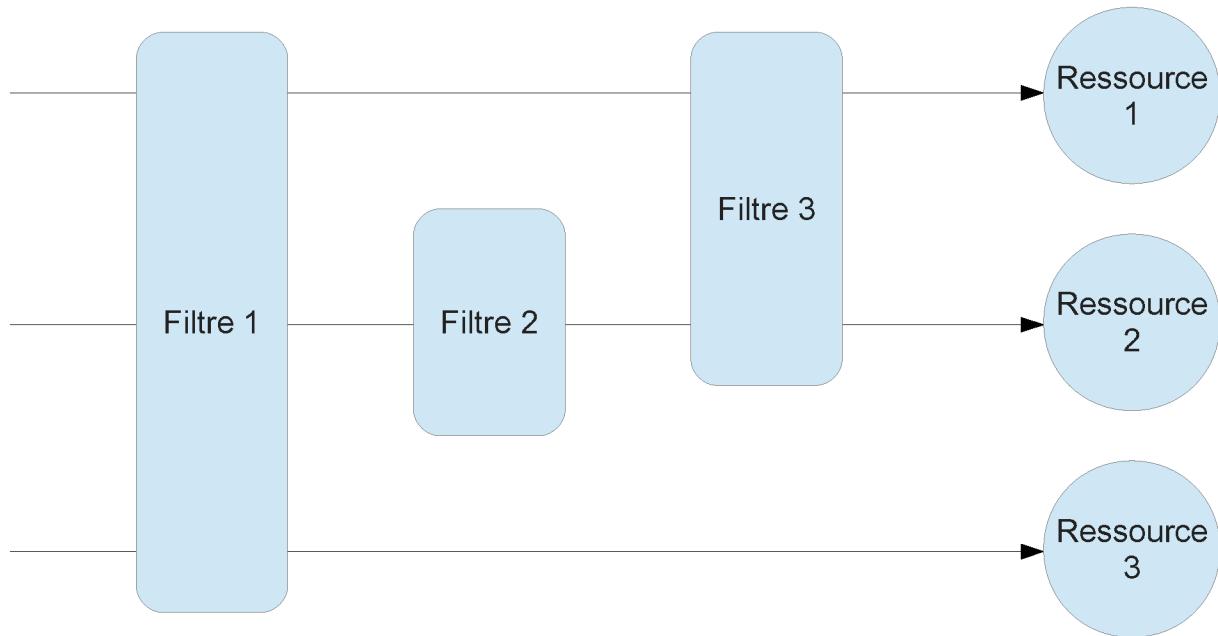
## Servlet et JSP Les Filters

### Les filtres

- **Un filtre est une instance de la classe Filter pouvant être appelée avant l'accès à une ressource**
  - un filtre peut transformer l'en-tête et/ou le contenu de la requête et/ou de la réponse
  - plusieurs filtres peuvent se suivre
  - un filtre est associé avec une ou plusieurs URLs
  - les filtres peuvent être chaînés



# Les filtres



# Les filtres

- Un filtre permet d'ajouter une fonctionnalité
  - authentification
  - journalisation
  - conversion d'images
  - compression
  - chiffrement
  - ...

# Développement d'un filtre

- Classes clés
  - dans le package `javax.servlet`
  - `Filter` : la classe devant être implémentée
  - `FilterChain` : représentant le chaînage des filtres
  - `FilterConfig` : utilisé pour passer de l'information au filtre durant son initialisation

# Développement d'un filtre

- Implémenter l'interface `Filter`
- Configurer le filtre
  - par annotation
  - ou dans le fichier `web.xml`

# Développement d'un filtre

- Implémentation de Filter

```
public class FilterOne implements Filter {  
  
    public void init(FilterConfig fConfig) throws ServletException {  
        // initialisation du filtre  
    }  
  
    public void destroy() {  
        // libération des éventuelles ressources  
    }  
  
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  
            throws IOException, ServletException {  
        // faire un traitement et faire suivre vers la chaîne de filtre  
        chain.doFilter(request, response);  
    }  
}
```

## Filtre : configuration par annotation

- Configuration par annotation
  - annotation @WebFilter

```
@WebFilter(filterName="FilterOne",urlPatterns={"/**"})  
public class FilterOne implements Filter{...}
```



- ordre d'invocation des filtres : **ordre alphanumérique des noms des filtres**

# Filtre : configuration par annotation

- Attributs principaux de l'annotation `@WebFilter`
  - `filterName` : nom du filtre
  - `urlPatterns` : tableau d'URLs sur lesquelles le filtre est appliqué
  - `servletNames` : tableau des noms de servlets sur lesquelles le filtre est appliqué
  - `initParams` : tableau de `@WebInitParam`
  - `dispatcherTypes` : tableau de `DispatcherType`
    - REQUEST, ASYNC, FORWARD, INCLUDE, ERROR
    - REQUEST par défaut

# Filtre : configuration par `web.xml`

- Comme pour les servlets
  - déclaration des filtres, avec leur nom
  - déclaration des mappings d'URLs
- ⚠ • l'ordre de déclaration dans le fichier détermine l'ordre d'invocation des filtres

# Filtre : configuration par web.xml

- Extrait du fichier web.xml

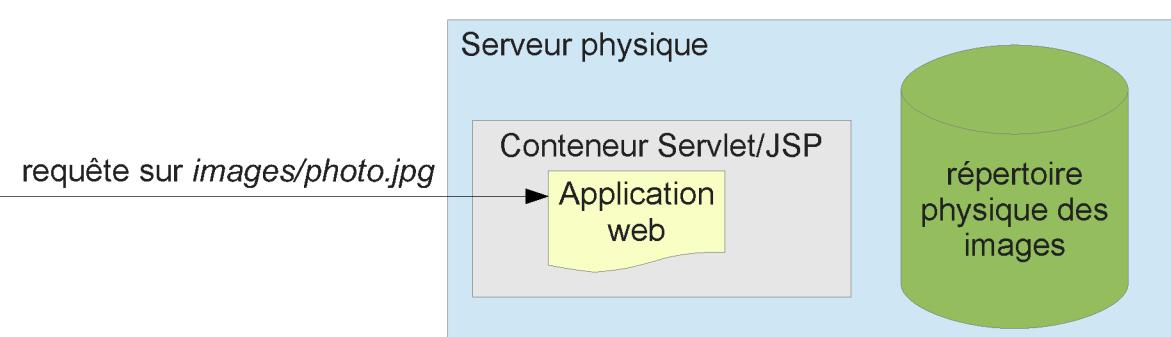
```
<filter>
    <filter-name>FilterOne</filter-name>
    <filter-class>org.antislashn.web.FilterOne</filter-class>
</filter>
<filter>
    <filter-name>FilterTwo</filter-name>
    <filter-class>org.antislashn.web.FilterTwo</filter-class>
</filter>

<filter-mapping>
    <filter-name>FilterOne</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>FilterTwo</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

## Exemple de filtre

- Ce filtre permet de servir des images qui ne sont pas dans le war
  - les images sont dans un répertoire externe à l'application web
    - exemple : répertoire d'enregistrement d'images envoyées vers le serveur, par un `<input type="file">`



# Exemple de filtre

- Le répertoire physique où se trouve les images est paramétré dans le fichier *web.xml*
  - extrait du fichier *web.xml*

```
<context-param>
    <param-name>upload-folder</param-name>
    <param-value>/test/</param-value>
</context-param>
```

- Les URLs des images commences par *images*

```

```

- Le filtre est donc chargé d'envoyer les images correspondant au répertoire logique *images*

# Exemple de filtre

- Récupération du répertoire physique des images dans la méthode `init()`

```
@WebFilter(dispatcherTypes = {DispatcherType.REQUEST }, urlPatterns = {"/*"})
public class ImageFilter implements Filter {
    private String folder;
    private final static Logger LOGGER = Logger.getLogger(ImageFilter.class.getCanonicalName());

    public void init(FilterConfig fConfig) throws ServletException {
        folder = fConfig.getServletContext().getInitParameter("upload-folder");
    }
    ...
}
```

```
<context-param>
    <param-name>upload-folder</param-name>
    <param-value>/test/</param-value>
</context-param>
```

# Exemple de filtre

- Méthode doFilter()

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
    throws IOException, ServletException {
    String url = ((HttpServletRequest)request).getRequestURL().toString();
    LOGGER.info(">>> url : "+url);
    String[] urlSplit = url.split("/");
    String image = urlSplit[urlSplit.length-1];
    LOGGER.info(">>> image : "+image);

    // Positionnement du type MIME
    FileNameMap fileNameMap = URLConnection.getFileMap();
    String mimeType = fileNameMap.getContentTypeFor(image);
    ((HttpServletResponse) response).setContentType(mimeType);
    OutputStream out = response.getOutputStream();

    // Depuis Java 7
    Path path = FileSystems.getDefault().getPath(folder, image);
    Files.copy(path, out);

    out.flush();
    out.close();
}
```

# Exemple de filtre

- La méthode doFilter() n'est pas appelée sur FilterChain car c'est directement le filtre qui renvoie la ressource
- Le type MIME doit être positionné
  - on récupère la ressource demandée (l'image)
  - FileNameMap permet de récupérer le type MIME
- Ce code n'est qu'un exemple, améliorations possibles
  - gestion de l'erreur 404

# Servlet et JSP

## Envoi de fichiers vers le serveur

## Envoi de fichiers vers le serveur

- L'upload de fichier est une tâche récurrente lors du développement d'un site Web
- Côté navigateur il est nécessaire d'avoir un formulaire gérant l'envoi du fichier
  - le formulaire est de type `multipart/form-data` et envoyé en `POST`
  - le champ input est de type `file`

```
<form method="post" enctype="multipart/form-data" action="SimpleUploadServlet">
    Votre nom : <input type="text" name="name"><br/>
    Téléchargement d'un fichier : <input type="file" name="simple-file">
    <input type="submit" value="OK" />
</form>
```



# Envoi de fichiers vers le serveur

- L'envoi du formulaire est alors différent d'un envoi par défaut (application/x-www-form-urlencoded)
  - exemple d'envoi d'un fichier texte et d'un champ nommé '*name*'

```
Content-Type: multipart/form-data; boundary=-----91787871417
Content-Length: 383

-----91787871417
Content-Disposition: form-data; name="name"

-----91787871417
Content-Disposition: form-data; name="simple-file"; filename="test.txt"
Content-Type: text/plain

Lorem ipsum dolor sit amet consectetuer interdum enim Pellentesque justo urna. Turpis tincidunt Aenean.
-----91787871417--
```

# Envoi de fichiers vers le serveur

- L'analyse du corps du message est plus complexe
  - il n'y a pas d'accès directes aux champs et à leur valeur
    - exemple pour le champ '*name*'
  - pas d'accès via `request.getParameter ("name")`
- **Servlet 3.0 a ajouté l'annotation**  
`@MultipartConfig`
  - l'analyse des champs est effectué, et les champs n'étant pas de type file peuvent être récupérés par la méthode `request.getParameter ()`
  - avant Java EE 6 il était courant d'utiliser la bibliothèque Common File Upload d'Apache

# Configuration de la servlet

- La servlet assurant la réception du fichier est configurée
  - par l'annotation `@MultipartConfig`
  - par l'équivalent XML dans le fichier `web.xml`
    - élément `<multipart-config>` présent dans l'élément `<servlet>`
    - l'élément `<multipart-config>` peut comporter des éléments enfants correspondant aux attributs de l'annotation
      - cf. slide suivant

# Configuration de la servlet

- Attributs de l'annotation `@MultipartConfig`
  - `location` : emplacement physique absolu du répertoire temporaire de réception de fichier
    - correspond à l'élément `<location>`
  - `fileSizeThreshold` : taille maximale de fichier pouvant être mis en mémoire
    - correspond à l'élément `<file-size-threshold>`
  - `maxFileSize` : taille maximale de fichier
    - correspond à l'élément `<max-file-size>`
  - `maxRequestSize` : taille maximale de la requête
    - correspond à l'élément `<max-request-size>`

# Réception du fichier

- La spécification Servlet 3.0 a ajoutée deux méthodes à la classe HttpServletRequest
  - Collection<Part> getParts()
  - Part getPart(String name)
- La classe Part représente un item reçu via une requête POST en *multipart/form-data*

## Exemple de servlet recevant un fichier

```
@WebServlet("/SimpleUploadServlet")
@MultipartConfig(location="c:/temp")
public class SimpleUploadServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private final static Logger LOGGER = Logger.getLogger(SimpleUploadServlet.class.getCanonicalName());

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String folder = getServletContext().getInitParameter("upload-folder");

        // getParameter fonctionne en enctype="multipart/form-data" grâce à l'annotation @MultipartConfig
        String name = request.getParameter("name");
        LOGGER.info("Paramètre 'name' == "+name);

        final Part filePart = request.getPart("simple-file");
        final String fileName = getFileName(filePart);

        // copie le fichier reçu vers son emplacement définitif
        Path path = FileSystems.getDefault().getPath(folder, fileName);
        InputStream in = filePart.getInputStream();
        Files.copy(in, path);
        in.close();

        // pour supprimer le fichier temporaire
        filePart.delete();

        RequestDispatcher rd = getServletContext().getRequestDispatcher("/index.jsp");
        rd.forward(request, response);
    }
}
```

cf. slide suivant

# Exemple de servlet recevant un fichier

```
...
private String getFileName(Part part) {
    final String partHeader = part.getHeader("content-disposition");
    LOGGER.log(Level.INFO, "Part Header = {0}", partHeader);
    for (String content : part.getHeader("content-disposition").split(";")) {
        if (content.trim().startsWith("filename")) {
            return content.substring(content.indexOf('=') + 1).trim().replace("\\\"", "\"");
        }
    }
    return null;
}
}
```

## Réception de plusieurs fichier

- Le champs input de type file peut permettre l'envoi de plusieurs fichiers

```
<input type="file" name="multi-file" multiple="multiple" />
```

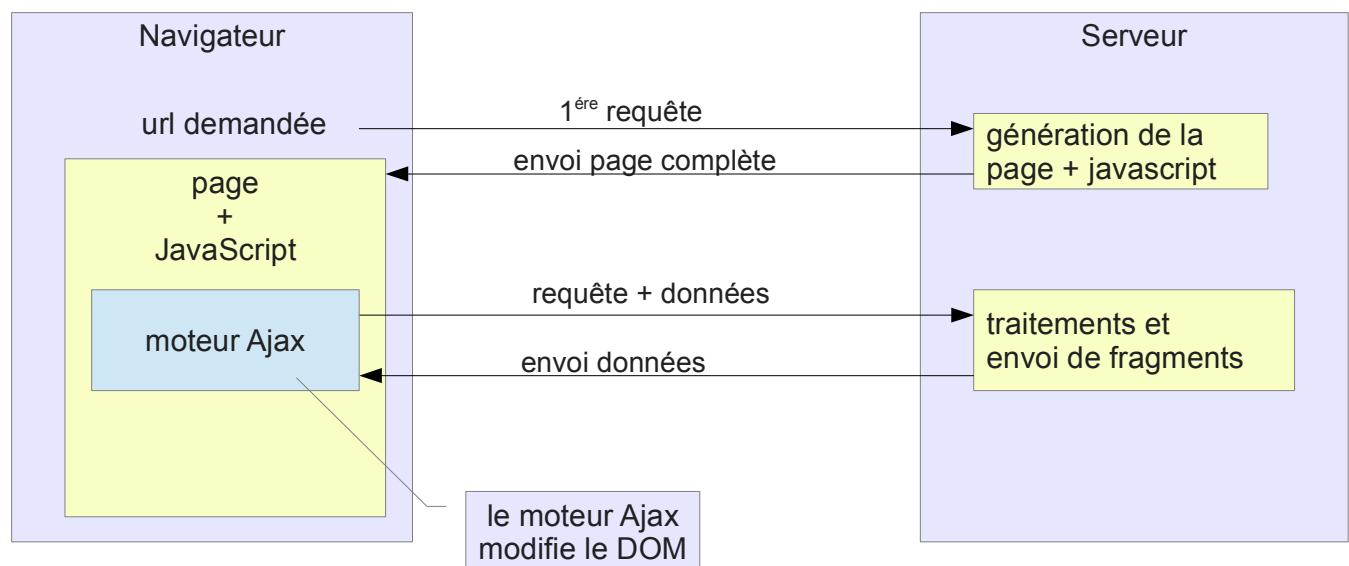
- extrait de code de la servlet

```
for(Part filePart : parts){
    final String fileName = getFileName(filePart);
    if(fileName == null)
        continue; // le part content n'est pas de type 'filename'
    Path path = FileSystems.getDefault().getPath(folder, fileName);
    if(!path.toFile().exists()){
        InputStream in = filePart.getInputStream();
        Files.copy(in, path);
        in.close();
    }
    else{
        // il faut prévoir un traitement spécifique si le fichier existe déjà
    }
}
```

# Servlet et JSP Ajax et JSON

## Ajax

- Application Ajax



# Ajax

- La classe XMLHttpRequest permet d'envoyer une requête HTTP vers le serveur et d'en contrôler la réception
  - la manière de récupérer une instance diffère selon les navigateurs
    - sous IE 6 et moins :
      - var reqAjax = new ActiveXObject('Microsoft.XMLHTTP');
    - sous autres navigateurs et à partir de IE 7
      - var reqAjax = new XMLHttpRequest();
  - l'utilisation de l'instance est ensuite homogène
- JQuery simplifie les interrogations asynchrones

# JSon

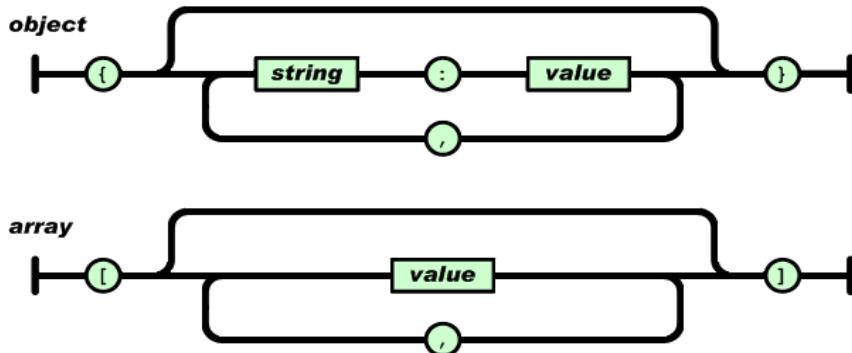
- JSon : JavaScript Object Notation
  - format d'échange humainement compréhensible
  - RFC 4627
  - souvent utilisé pour la sérialisation et la transmission d'objets
  - la fonction eval() de JavaScript permet ensuite d'évaluer la chaîne de caractères JSon

```
var r = eval("2+2");
console.log(r);
var json = '{civilite:"M",...,adresse:{rue:"Rue de Bruxelles",...}}';
var obj = eval("(" + json + ")");
console.log(obj.adresse.ville);
```

json\_1.html

# JSon

- Syntaxe de base
  - cf le site de référence <http://www.json.org/>



## Exemple

- L'exemple présenté utilise JQuery et une servlet
  - projet Eclipse "JSON"
  - mais possibilité d'utiliser directement XMLHttpRequest
  - autre solution côté serveur : utiliser des services REST Java EE
- Un formulaire permet d'envoyer au serveur des données, en Ajax
  - une ville et son code postal
  - La réponse à la requête Ajax est renvoyée par le serveur de la liste des villes

# Exemple

- Partie HTML

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <script type="text/javascript" src="js/jquery-2.1.4.min.js"></script>
        <script type="text/javascript" src="js/application.js"></script>
        <title>JSON</title>
    </head>
    <body>
        Ajouter une ville<br />
        Code postal <input type="text" id="cp" /><br />
        Ville <input type="text" id="nom" /><br />
        <input type="button" id="addButton" value="Ajouter" />
        <h3>Liste des villes</h3>
        <table id="container"></table>
    </body>
</html>
```

# Exemple

- Partie HTML

- deux fichiers JavaScript sont chargés
  - JQuery et scripts de l'application
- deux champs de formulaires peuvent être remplis par l'utilisateur
  - identifiés par 'cp' et 'nom'
- un champ de type button (' addButton') permettra l'appel de la fonction JavaScript qui sera chargée
  - d'envoyer la ville saisie vers le serveur au format JSON
  - de récupérer la liste des villes et de gérer l'affichage dans un tableau ('container')

# Exemple

- Partie JavaScript

```
function bindEvents(){
    $("#addButton").click(function(event){
        var ville = {};
        ville.codePostal = $("#cp").val();
        ville.nom = $("#nom").val();
        clearContainer();
        $.post("VilleServlet", {'ville':JSON.stringify(ville)},function(villes){
            for(var i=0 ; i<villes.length ; i++){
                var ville = {};
                ville.codePostal = villes[i].codePostal;
                ville.nom = villes[i].nom;
                addVilleToContainer(ville);
            }
        });
    });
}
function clearContainer(){
    $("#container").html("");
}
function addVilleToContainer(ville){
    var line = "<tr><td>" +ville.codePostal+ "</td><td>" +ville.nom+ "</td></tr>";
    console.log(line);
    $("#container").append(line);
}
$(document).ready(bindEvents);
```

# Exemple

- Code de la servlet

```
@WebServlet("/VilleServlet")
public class VilleServlet extends HttpServlet {
    private static final VilleService service = new VilleService();

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String json = request.getParameter("ville");
        Ville ville = VilleHelper.getVilleFromJson(json);
        service.add(ville);
        List<Ville> villes = service.getAllVilles();
        String jsonResponse = VilleHelper.getJson(villes);
        Writer out = response.getWriter();
        response.setContentType("application/json");
        out.write(jsonResponse);
        out.flush();
    }

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

# Exemple

- Un service est simulé par la classe VilleService
  - la classe Ville est un simple POJO

```
public class VilleService {  
    private List<Ville> villes = new ArrayList<Ville>();  
  
    public void add(Ville ville){  
        villes.add(ville);  
    }  
  
    public List<Ville> getAllVilles(){  
        return villes;  
    }  
}
```

# Exemple

- Une classe outil VilleHelper gère la transformation JSON ↔ Java
  - utilisation de la bibliothèque JSON.simple
    - <https://code.google.com/p/json-simple/>
  - conversion JSON vers java
  - conversion liste de ville en tableau JSON

# Exemple

- Conversion JSON vers Java

```
public static Ville getVilleFromJson(String json){  
    try{  
        Ville ville = new Ville();  
        JSONParser parser = new JSONParser();  
        JSONObject object = (JSONObject) parser.parse(json);  
        ville.setCodePostal((String)object.get("codePostal"));  
        ville.setNom((String)object.get("nom"));  
        return ville;  
    }  
    catch (ParseException e) {  
        return null;  
    }  
}
```

# Exemple

- Conversion List<Ville> vers JSON

```
public static String getJson(List<Ville> villes) {  
    JSONArray array = new JSONArray();  
    for(Ville ville : villes){  
        JSONObject object = new JSONObject();  
        object.put("codePostal", ville.getCodePostal());  
        object.put("nom", ville.getNom());  
        array.add(object);  
    }  
    return array.toJSONString();  
}
```

# Servlet et JSP

## Servlet asynchrone

## Concepts

- Une avancée majeure entre HTTP 1.0 et HTTP 1.1 a été de permettre les connexions persistantes
  - HTTP 1.0 : la connexion entre le navigateur et le serveur était fermée après un cycle requête-réponse.
  - HTTP 2.0 : plusieurs cycles requête-réponse peuvent être encapsulées dans la même connexion
    - pas besoin de renégocier la connexion TCP après chaque requête



# Concepts

- Chaque connexion persistante est gérée par un thread
  - le serveur gère un pool de thread
  - lorsque la connexion est fermée, le thread utilisé par cette connexion rejoint le pool
  - la consommation mémoire est directement liée au nombre de connexions
  - le nombre de threads de connexion est fixe
    - si une requête arrive alors qu'aucun thread de connexion n'est disponible, elle est rejetée
    - modèle nommé "thread per connection"

# Concepts

- Grâce à l'API NIO (i/o non bloquant) une connexion HTTP ne requiert plus d'être attachée à son thread
  - lorsqu'une connexion est inutilisée entre les requêtes, son thread peut-être recyclé
    - la connexion est alors placée dans une file d'attente centralisée, sans consommer un thread séparé
    - ce modèle (appelé "thread per request") permet au serveur d'accepter plus de connexion qu'il y a de threads disponibles dans le pool
  - l'ensemble des serveurs web Java fonctionnent sur ce modèle

# Concepts

- L'expérience utilisateur est améliorée avec l'utilisation d'Ajax
  - utilisation du modèle one-page plutôt que page-by-page
- Le modèle one-page requiert plus de requêtes du client vers le serveur
  - JavaScript demande régulièrement des mises à jour au serveur (polling)
  - plus de requêtes => plus de threads consommés
    - ce qui peut annuler les bénéfices du modèle thread-per-request

# Concepts

- Certaines routines serveur peuvent faire empirer la situation
  - une requête peut-être bloquée à cause d'une connexion JDBC, d'un accès à un web service, ...
  - le thread associé est alors bloqué
- Solution : mettre la requête dans une file d'attente centralisée et libérer le thread
  - le thread lié est alors recyclé
- Le support des servlets asynchrones permet gérer ce pattern (Servlet 3.0)

# Concepts

- `ServletRequest` possède de nouvelles méthodes
  - `AsyncContext startAsync()`
  - `AsyncContext startAsync(ServletRequest request, ServletResponse response)`
- Si l'attribut `asyncSupported` est positionné, la réponse (`ServletResponse`) n'est pas validée à la fin d'une méthode `doXXXX`
- `AsyncContext` encapsule les objets `request` et `response`

# Concepts

- Classe `AsyncContext`
  - encapsule la requête et la réponse en cours
    - méthodes `getRequest()`, `getResponse()`
  - permet de faire suivre vers une autre ressource
    - méthodes `dispatch(...)`
      - cf. documentation
  - permet de valider la fin de la requête
    - méthode `complete()`
  - accepte un listener
    - méthode `addListener(...)`

# Mise en place

- Configuration de la servlet

- par annotation

```
@WebServlet(value="/TraitementLongServlet", asyncSupported=true)
```

- ou dans le fichier web

```
<servlet>
  <servlet-name>TraitementLongServlet</servlet-name>
  <servlet-class>org.antislashn.web.TraitementLongServlet</servlet-class>
  <async-supported>true</async-supported>
</servlet>
```

# Mise en place

- Dans cet exemple nous utiliserons un ThreadPoolExecutor pour gérer les tâches devant être exécutées de manière asynchrone
  - associé au contexte applicatif

```
@WebListener
public class ApplicationListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent evt) {
        BlockingQueue<Runnable> jobQueue = new ArrayBlockingQueue<>(100);
        ThreadPoolExecutor executor = new ThreadPoolExecutor(100, 200, 50, TimeUnit.SECONDS, jobQueue);
        evt.getServletContext().setAttribute("EXECUTOR", executor);
    }

    public void contextDestroyed(ServletContextEvent evt) {
        ThreadPoolExecutor executor = (ThreadPoolExecutor) evt.getServletContext().getAttribute("EXECUTOR");
        executor.shutdown();
    }
}
```

# Mise en place

projet Eclipse : Servlet Asynchrone

- Création d'un worker simulant un traitement long

```
public class WorkerAsync implements Runnable {
    private static final Logger LOG = Logger.getLogger(WorkerAsync.class.getCanonicalName());
    private AsyncContext asyncContext;

    public WorkerAsync(AsyncContext asyncContext) {
        this.asyncContext = asyncContext;
    }

    @Override
    public void run() {
        LOG.info(">>> DEBUT TRAITEMENT ASYNCHRONE");
        int delay = (int)(Math.random() * 10) + 1;

        try {
            Thread.sleep(delay*1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        ServletRequest request = asyncContext.getRequest();
        request.setAttribute("delay", delay);
        asyncContext.dispatch("/fin.jsp"); ←
        LOG.info(">>> DEBUT TRAITEMENT ASYNCHRONE");
    }
}
```

on fait suivre vers la page JSP

# Mise en place

projet Eclipse : Servlet Asynchrone

- Codage de la servlet

```
@WebServlet(value="/TraitementLongServlet", asyncSupported=true)
public class TraitementLongServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static Logger LOGGER = Logger.getLogger(TraitementLongServlet.class.getCanonicalName());

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        ThreadPoolExecutor executor = (ThreadPoolExecutor)getServletContext().getAttribute("EXECUTOR");
        AsyncContext asyncContext = request.startAsync();
        WorkerAsync worker = new WorkerAsync(asyncContext);
        executor.execute(worker);
    }

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

la réponse n'est pas complétée à la fin de la méthode

# Mise en place

projet Eclipse : Servlet Asynchrone

- JSP affichant la fin du traitement

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>Servlet Asynchrone - 02</title>
    </head>
    <body>
        <h2>Fin de traitement</h2>
        Temps de traitement : <%=request.getAttribute("delay") %> s.<br/>
        <a href="TraitementLongServlet">Traitement long</a>
    </body>
</html>
```

attribut mis en place  
par le worker

# Mise en place

projet Eclipse : Servlet Asynchrone

- AsyncContext accepte aussi
  - un timeout
    - méthode setTimeout (long delay)
  - un listener
    - méthode addListener (...)
    - le listener doit implémenter l'interface AsyncListener

# Configurations serveur

- Indépendamment des configurations de l'application, les serveurs ont leurs propres timeouts sur les connecteurs HTTP
  - ce timeout peut-être réglé pour éviter les erreurs de type

```
java.lang.IllegalStateException: The request associated with the AsyncContext has already completed processing.
```

- Tomcat : *server.xml*

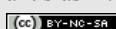
```
<Connector connectionTimeout="20000" port="8080" protocol="HTTP/1.1"
           redirectPort="8443"
           asyncTimeout="30000"/>
```



# Servlet et JSP WebSocket

## Concepts

- HTTP est le protocole standard du Web
  - bidirectionnel à l'alternat (half-duplex)
    - pattern request/response
  - verbeux
    - beaucoup d'informations présentes dans les en-têtes des requêtes et réponses
  - pas de server push
    - c'est toujours le client qui initie la requête
    - un JavaScript doit interroger régulièrement le serveur
      - polling, long pooling



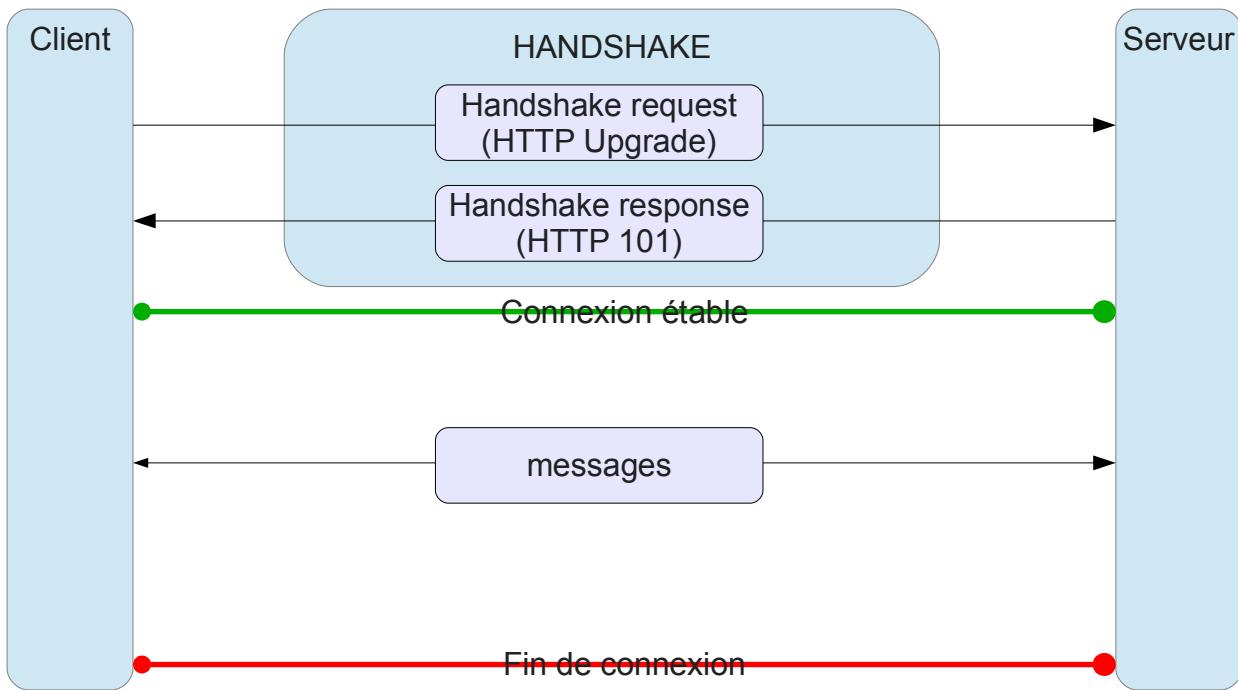
# Concepts

- Polling
  - un JavaScript interroge régulièrement le serveur sur la mise à jour de données
  - si le serveur n'a pas de mise à jour à communiquer, il peut renvoyer une réponse vide
    - la requête est complétée
- Long polling
  - un JavaScript interroge le serveur
  - si le serveur n'a pas de mise à jour à communiquer il garde la connexion jusqu'à ce de nouvelles données soient envoyées au client
    - cf. les servlets asynchrones

# Concepts

- WebSocket
  - protocole différent de HTTP
    - devait faire partie de HTML 5
    - fait l'objet d'une spécification IETF - RFC 6455
  - permet d'obtenir un canal de communication
    - sur un socket TCP
    - bidirectionnel simultané (full duplex)
- API WebSocket introduite dans Java EE 7
  - JSR 356

# Protocole WebSocket



# Protocole WebSocket

- Le handshake va permettre de passer du protocole HTTP au protocole WebSocket
  - URLs du type *ws://domaine:port/ressource*
    - exemple : *ws://127.0.0.1:8080/websocket/echo*
  - exemple de handshake

| En-têtes de la requête (0,540 Ko) |   |
|-----------------------------------|---|
| Host :                            | "127.0.0.1:8080"  |
| User-Agent :                      | "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:40.0) Gecko/20100101 Firefox/40.0" |
| Accept :                          | "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"           |
| Accept-Language :                 | "fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3"                                       |
| Accept-Encoding :                 | "gzip, deflate"   |
| DNT :                             | "1"   |
| Sec-WebSocket-Version :           | "13"  |
| Origin :                          | "http://localhost:8080"   |
| Sec-WebSocket-Extensions :        | "permessage-deflate"  |
| Sec-WebSocket-Key :               | "0j+K4xWkq/9nLUabiUojUg=="  |
| Connection :                      | "keep-alive, Upgrade"   |
| Pragma :                          | "no-cache"  |
| Cache-Control :                   | "no-cache"  |
| Upgrade :                         | "websocket"   |

| En-têtes de la réponse (0,313 Ko) |                                      |
|-----------------------------------|--------------------------------------|
| Connection :                      | "Upgrade"                            |
| Content-Length :                  | "0"                                  |
| Date :                            | "Tue, 15 Sep 2015 10:12:21 GMT"      |
| Origin :                          | "http://localhost:8080"              |
| Sec-WebSocket-Accept :            | "Tk6nmq8xOhcB7xgWhX1NSYbK9UI="       |
| Sec-WebSocket-Location :          | "ws://127.0.0.1:8080/websocket/echo" |
| Server :                          | "WildFly/9"                          |
| Upgrade :                         | "WebSocket"                          |
| X-Powered-By :                    | "Undertow/1"                         |

# WebSocket

- Le côté client nécessite une application JavaScript
  - ouverture de la connexion
  - gestion de la connexion
  - envoi et réception des messages
  - interface WebSocket
- Le côté serveur nécessite une programmation en Java
  - API WebSocket
  - configuration par annotations
  - sur serveur Java EE

## WebSocket API JavaScript

- Interface WebSocket
  - cd. <http://www.w3.org/TR/websockets/>
  - API relativement simple
    - l'URL de connexion est passée au constructeur qui ouvre la connexion
    - différentes méthodes `send(...)` permettent d'envoyer des données au serveur
    - une méthode `close()` ferme la connexion
    - des callbacks gèrent les états de la connexion et la réception des messages
      - `onopen`, `onerror`, `onclose`, `onmessage`
  - il existe un plugin pour JQuery <https://code.google.com/p/jquery-websocket/>

# API Java EE

projet Eclipse : WebSocket - 01

- Côté serveur - Principales annotations
  - packages `javax.websocket` et `javax.websocket.server`
  - `@ServerEndpoint` : annotation de classe spécifiant la classe d'entrée du WebSocket
  - `@OnClose` : annotation sur la méthode à invoquer lorsque la session est ouverte
  - `@OnMessage` : annotation sur la méthode à invoquer lors de la réception d'un message
  - `@OnClose` : annotation sur la méthode à invoquer lorsque la session est fermée
  - `@OnError` : annotation sur la méthode à invoquer en cas d'erreur

# API Java EE

projet Eclipse : WebSocket - 01

- `@ServerEndpoint` - attributs
  - `value` : valeur de l'URI ou modèle d'URI

```
@ServerEndpoint("/echo")
@ServerEndpoint("/chat/{user}")
```
  - `encoders`, `decoders` : classes Java de sérialisation/désérialisation
    - implémentent `Encoder` et `Decoder`
  - `configurator` : classe de configuration personnalisée
  - `subprotocols` : sous-protocoles supportés
    - strings de protocoles personnalisés
    - un même serveur peut alors servir plusieurs protocoles

# API Java EE

- @OnOpen
  - paramètres optionnels de la méthode annotée
    - Session
    - EndpointConfig
    - nombre variable de strings, chacune étant annotée avec @PathParam
      - en relation avec le modèle d'URL de @ServerEndpoint

```
@OnOpen  
public void onOpen(Session session, @PathParam("id") String id){...}  
  
@OnOpen  
public void onOpen(Session session){...}
```

# API Java EE

- @OnMessage
  - plusieurs signatures possibles pour la méthode
    - un paramètre pouvant être String, Reader, Object, byte[], InputStream, PongMessage
      - cd. la javadoc
    - optionnel : plusieurs strings annotées avec @PathParam
    - optionnel : objet de type Session

```
@OnMessage  
public String handleMessage(String message){  
    return "SERVER : "+message;  
}
```

# API Java EE

- `@OnClose` - paramètres optionnels de la méthode
  - paramètre de type `Session`
  - paramètre de type `CloseReason`
  - plusieurs strings annotées avec `@PathParam`
- `@OnError` - paramètres optionnels de la méthode
  - paramètre de type `Session`
  - paramètre de type `Throwable`
  - plusieurs strings annotées avec `@PathParam`

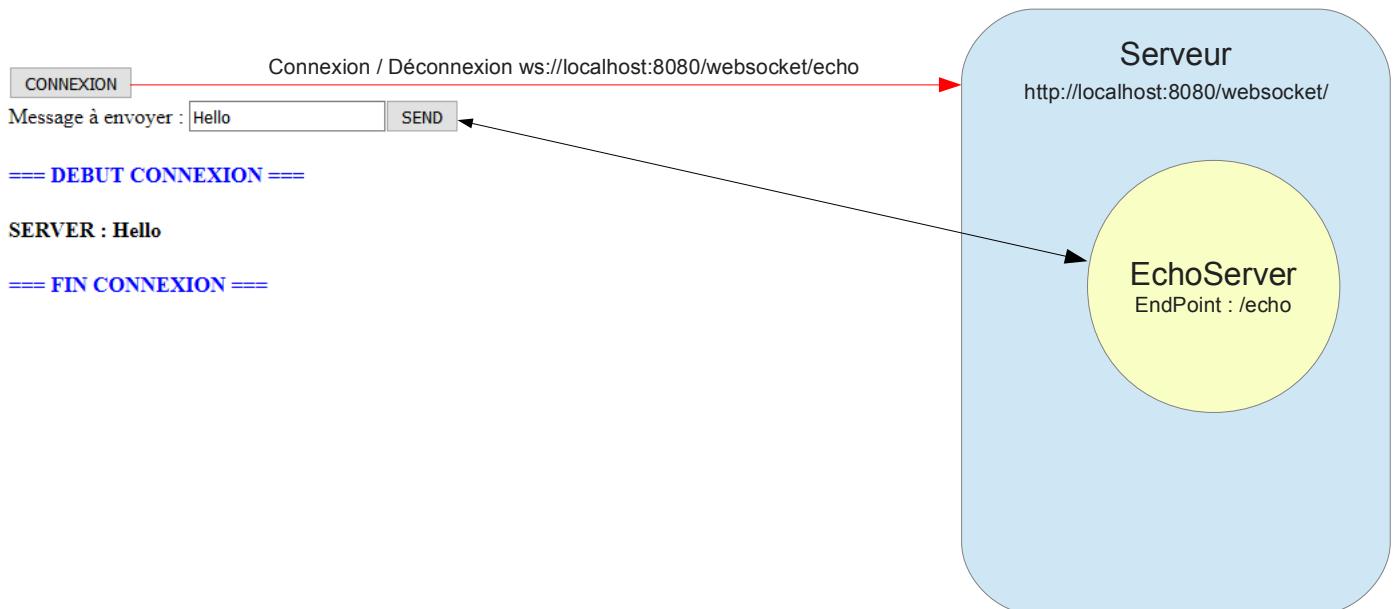
## Étapes de création de l'application

projet Eclipse : WebSocket - 01

- L'application "echo" présentée permet :
  - de se connecter et se déconnecter du serveur
  - d'envoyer un message
    - le serveur renvoie le même message
  - d'afficher les messages reçus par le serveur
  - de suivre les événements dans la console du navigateur
- L'application comprend :
  - une partie "client" : HTML + JavaScript
    - le HTML est généré par une page JSP
  - une partie "serveur" : le code Java du serveur "echo"

# Étapes de création de l'application

projet Eclipse : WebSocket - 01



# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- Page JSP

```
<html>
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>WebSocket - 01</title>
    <script type="text/javascript" src="js/websocket.js"></script>
  </head>
  <body>
    <input type="button" id="connectionButton" value="CONNEXION"/><br/>
    Message à envoyer : <input type="text" id="message">
    <input type="button" id="sendButton" value="SEND"/>
    <div id="container"></div>
  </body>
</html>
```

bouton gérant la connexion

bouton gérant l'envoi du message

zone d'affichage des messages

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- JavaScript : fonction de gestion du websocket

```
var uri = "ws://127.0.0.1:8080/websocket/echo";
var connected = false;

function echoWebSocket(){
    websocket = new WebSocket(uri);
    websocket.onopen = function(evt){
        connected = true;
        console.log("CONNECTION EFFECTUÉE");
        document.getElementById("container").innerHTML += "<h4 style='color:blue;'>==== DEBUT CONNEXION ====</h4>";
        document.getElementById("connectionButton").value = "DECONNEXION";
    };

    websocket.onclose = function(evt){
        connected = false;
        console.log("CONNECTION TERMINEE");
        document.getElementById("container").innerHTML += "<h4 style='color:blue;'>==== FIN CONNEXION ====</h4>";
        document.getElementById("connectionButton").value = "CONNEXION";
    };

    websocket.onmessage = function(evt){
        document.getElementById("container").innerHTML += "<h4>" + evt.data + "</h4>";
        console.log("RECEIVED : " + evt.data);
    };

    websocket.onerror = function(evt){
        console.log("ERROR : " + evt.data);
        document.getElementById("container").innerHTML += "<h4 style='color:red;'>" + evt.data + "</h4>";
    };
}


```

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- JavaScript : mise en place des événements

```
window.onload = function(){
    document.getElementById("connectionButton").onclick=function(){
        if(!connected){
            echoWebSocket();
        }else{
            websocket.close();
        }
    };

    document.getElementById("sendButton").onclick=function(){
        if(connected){
            var message = document.getElementById("message").value;
            console.log("SENT : " + message);
            websocket.send(message);
        }else{
            alert("Pas de connexion");
        }
    };
};


```

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- Côté serveur - réception des message
  - le serveur ne fait que renvoyer le message reçu

```
@ServerEndpoint("/echo")
public class EchoServer {
    private static Logger LOG = Logger.getLogger(EchoServer.class.getCanonicalName());

    @OnOpen
    public void onOpen(Session session){
        LOG.info(">>> DEBUT DE SESSION - ID : "+session.getId());
    }

    @OnClose
    public void onClose(Session session){
        LOG.info(">>> FIN DE SESSION - ID : "+session.getId());
    }

    @OnMessage
    public String handleMessage(String message){
        return "SERVER : "+message;
    }
}
```

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- L'intérêt du WebSocket est d'envoyer des informations depuis le côté serveur, sans que le client initie cette demande
- Il faut alors garder la liste des sessions connectées
  - ajout de la session dans la méthode @onOpen
  - retrait de la session dans la méthode @onClose

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- Extraits de la classe EchoServer

```
@ServerEndpoint("/echo")
public class EchoServer {
    private static Logger LOG = Logger.getLogger(EchoServer.class.getCanonicalName());
    static private Queue<Session> sessions = new ConcurrentLinkedQueue<>();

    ...

    @OnOpen
    public void onOpen(Session session){
        sessions.add(session);
        LOG.info(">>> DEBUT DE SESSION - ID : "+session.getId());
    }

    @OnClose
    public void onClose(Session session){
        sessions.remove(session);
        LOG.info(">>> FIN DE SESSION - ID : "+session.getId());
    }
    ...
}
```

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- Une méthode statique est ajoutée dans EchoServer pour l'envoi d'un message vers l'ensemble des sessions connectées

```
...
static public void send(String message) throws IOException{
    Date date = new Date();
    for(Session session : sessions){
        if(session.isOpen()){
            session.getBasicRemote().sendText("HEURE SERVEUR "+date+" - MESSAGE "+message);
        }
        else{
            sessions.remove(session);
        }
    }
}
...
```

# Étapes de création de l'application

projet Eclipse : WebSocket - 01

- Le test de l'envoi de messages est confié à un EJB

```
@Singleton  
@Startup  
public class SenderWorker {  
    @Resource private TimerService timerService;  
    private Timer timer;  
    private long delay = 5000;;  
  
    @PostConstruct public void init(){  
        timer = timerService.createTimer(delay , delay, "Timer de SenderWorker");  
    }  
  
    @PreDestroy public void destroy(){  
        timer.cancel();  
    }  
  
    @Timeout public void send(){  
        try {  
            EchoServer.send("Hi");  
        } catch (IOException e) {  
            LOG.log(Level.SEVERE, "Erreur", e);  
        }  
    }  
}
```

## Ressources

- Livres
  - Developping RESTful Services with JAX-RS2.0, WebSockets and JSON
    - éditeur : Packt Publishing
    - auteurs : Masoud Kalali & Bhaktu Mehta
  - Java EE 7 - Essentials
    - éditeur : O'REILLY
    - auteur : Arun Gupta

# Ressources

- Web

- <http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/HomeWebSocket/WebsocketHome.html>
- <http://mgreau.com/posts/2013/11/11/javaee7-websocket-angularjs-wildfly.html>
- <http://www.hascode.com/2013/08/creating-a-chat-application-using-java-ee-7-websockets-and-glassfish-4/>

