



FORMATION
JEE AVANCÉE

Formateur



Pr. O. EL MIDAoui

Professor & Senior JAVA Software Engineer

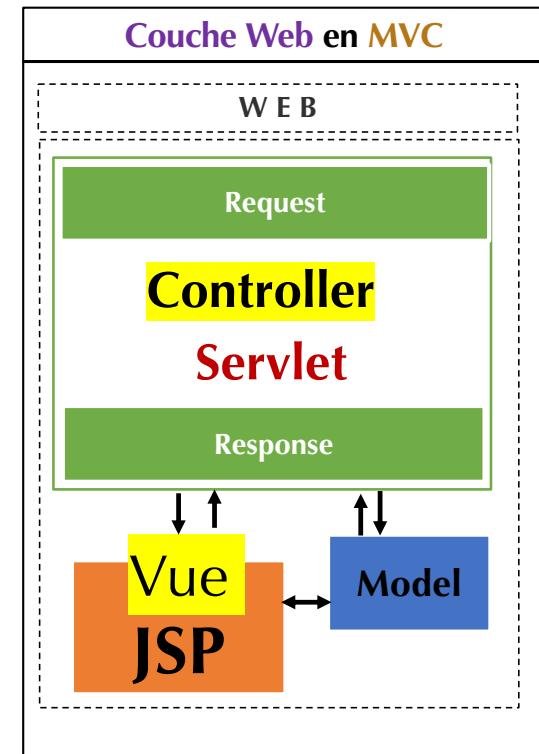
PhD in Data Science - Web Geographic Information Retrieval GIR

Plateforme
Java EE  La technologie JSP

Modèle MVC : vue [JSP]

Pages (JSP): Des Servlets avec vue

Le modèle **MVC** nous conseille de placer tout ce qui touche à l'affichage final (texte, mise en forme, etc.) dans une couche à part : la **vue**, et cela en utilisant la technologie **JSP**.



Modèle MVC : vue [JSP]

Pages (JSP): Des Servlets avec vue

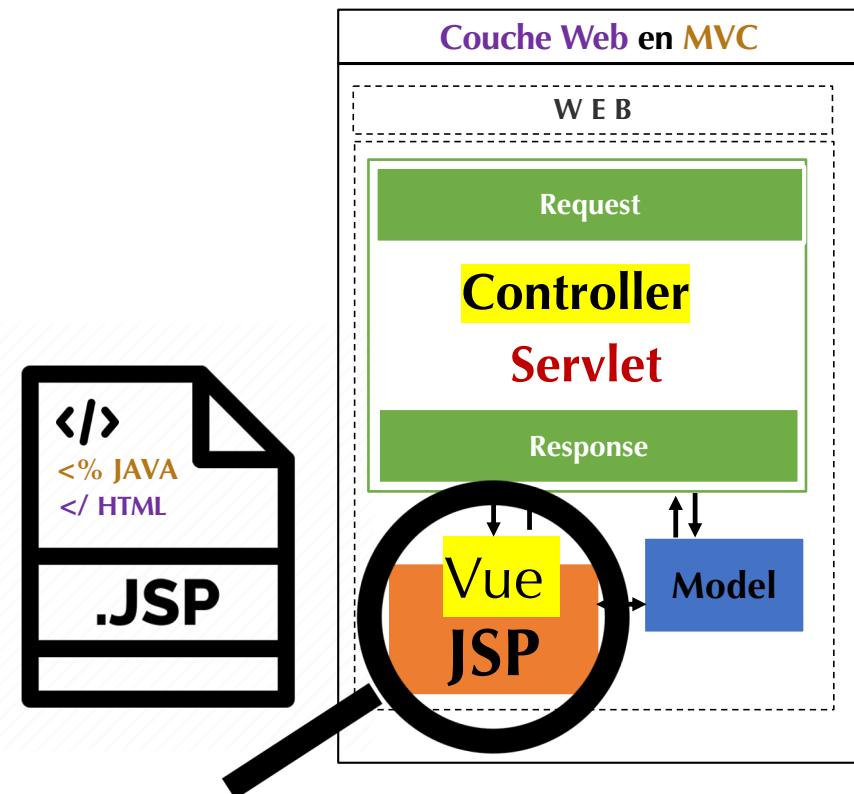
À quoi ressemble une page JSP ?

C'est une document qui ressemble beaucoup à une page **HTML**, mais qui en réalité en diffère par plusieurs aspects :

l'extension d'une telle page devient **.jsp** et non plus **.html**

une telle page peut contenir des balises **HTML**, mais également des balises **JSP** qui appellent de manière transparente du code Java

contrairement à une **page HTML statique** directement renvoyée au client, une **page JSP** est **exécutée côté serveur**, et génère alors une page renvoyée au client.

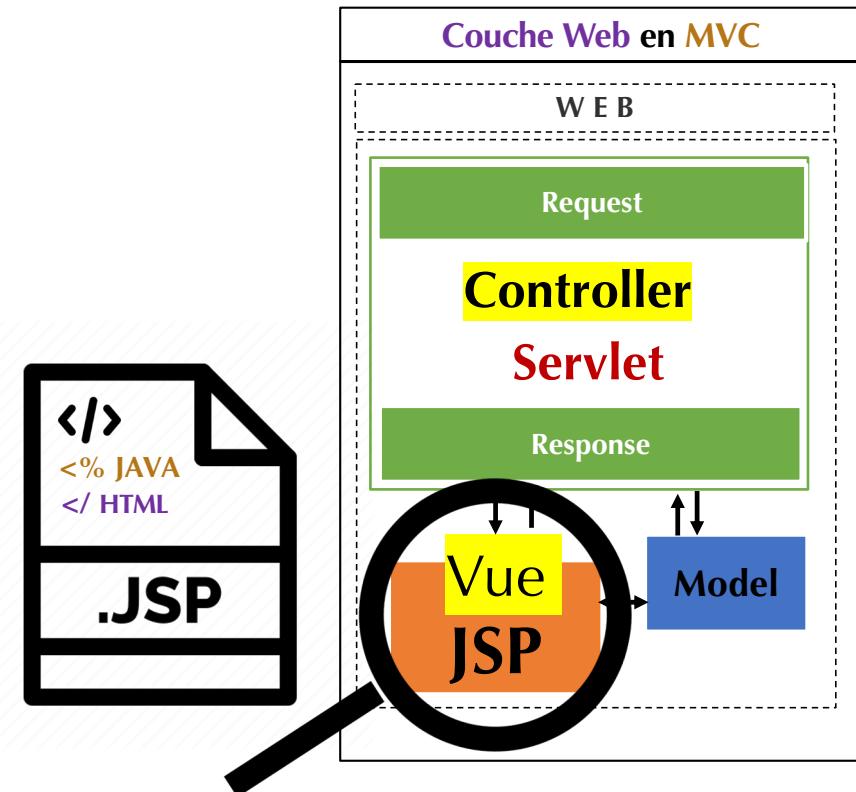


Modèle MVC : vue [JSP]

Pages (JSP): Des Servlets avec vue

Objectif

Rendre possible la création de **pages dynamiques** :
puisque'il y a une étape de génération sur le serveur, il devient possible de faire varier l'affichage et d'interagir avec l'utilisateur, en fonction notamment de la requête et des données reçues !



Modèle MVC : vue [JSP]

Pages (JSP): Des Servlets avec vue

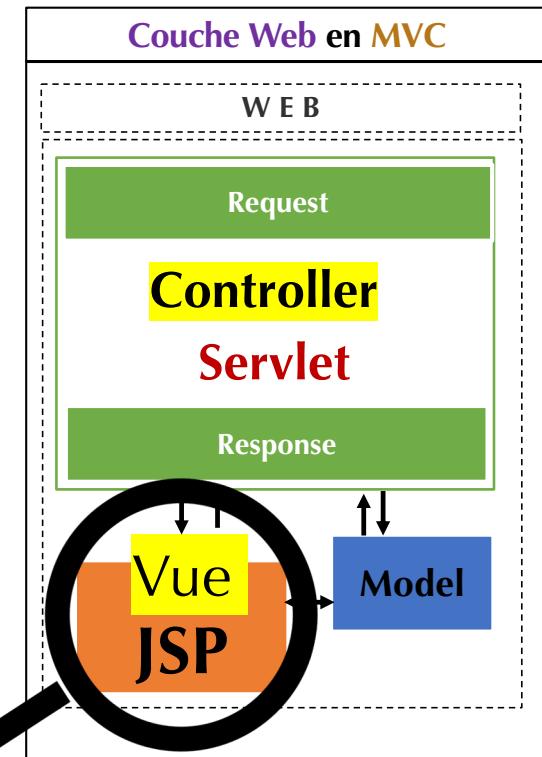


Quoi ?

Les pages JSP sont une des technologies de la plate-forme Java-EE les plus puissantes, simples à utiliser et à mettre en place.

Elles se présentent sous la forme d'un simple fichier au format texte, contenant des balises respectant une syntaxe à part entière.

Le langage JSP combine à la fois les technologies **HTML**, **XML**, **servlet** et **JavaBeans** en une seule solution permettant aux développeurs de créer des vues dynamiques.



Modèle MVC : vue [JSP]

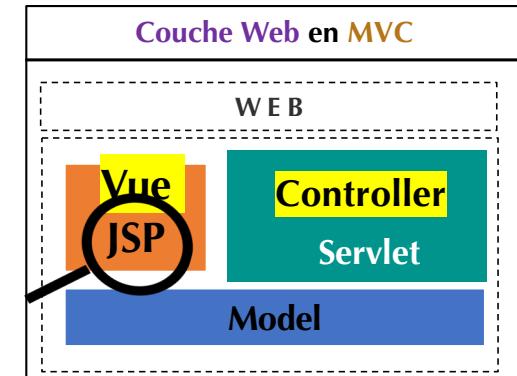
Pages (JSP): Des Servlets avec vue



Pourquoi ?

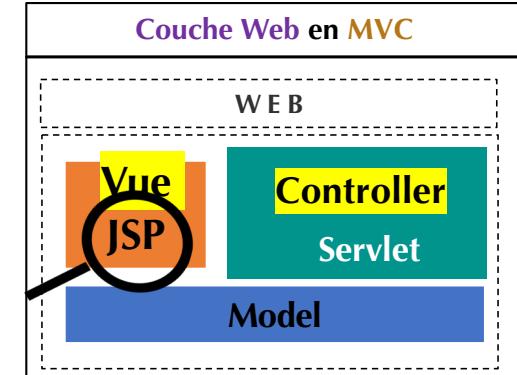
les raisons de l'existence de cette technologie

- La technologie **servlet** est trop difficile d'accès et ne convient pas à la génération du code de présentation
 - Écrire une page web en langage Java est horriblement pénible.
 - les **pages JSP** sont en quelque sorte une **abstraction "haut niveau"** de la technologie **servlet**. (**JSP** : simplification de l'API **servlet**)
- Le modèle **MVC** recommande une séparation nette entre le code de **contrôle** et la **présentation**.
 - Il est théoriquement envisageable d'utiliser certaines servlets pour le contrôle, et d'autres pour effectuer l'affichage, mais la servlet n'est pas adaptée à la prise en charge de l'affichage...
- Le modèle MVC recommande aussi une séparation nette entre le code **métier** et la **présentation**



Modèle MVC : vue [JSP]

Pages (JSP): Des Servlets avec vue



Comment ?

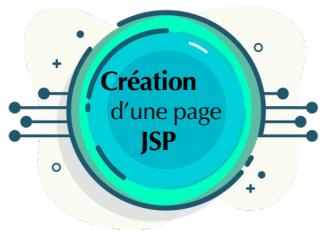
La technologie **JSP** est une technologie offrant les capacités dynamiques des **servlets** tout en permettant une approche naturelle pour la création de contenus statiques. Ceci est rendu possible par :

- **Un langage dédié** : les pages **JSP** sont des documents au format texte, à l'opposé des classes Java que sont les **servlets**, qui décrivent indirectement comment traiter une requête et construire une réponse. Elles contiennent des balises qui combinent à la fois simplicité et puissance, via une syntaxe simple, semblable au **HTML**.
- **L'accès aux objets Java** : des balises du langage **jsp** rendent l'utilisation directe d'objets au sein d'une page très aisée ;
- des mécanismes permettant **l'extension du langage** utilisé au sein des pages **JSP** : il est possible de mettre en place des balises qui n'existent pas dans le langage **JSP**, afin d'augmenter les fonctionnalités accessibles. (**JSTL**)



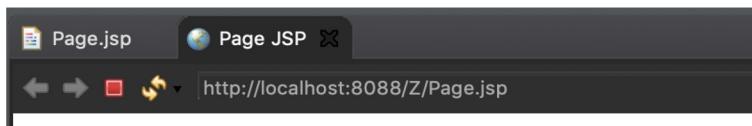
Modèle MVC : vue [JSP]

Pages (JSP): Mise en place



Une page JSP par défaut est alors générée par Eclipse

```
Page.jsp
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title> Page JSP </title>
5   </head>
6 
7   <body>
8     <h4>
9       Ceci est une page HTML
10      générée à partir d'un page JSP
11    </h4>
12  </body>
13 </html>
```

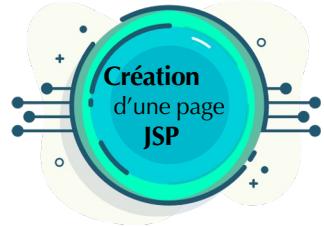


Ceci est une page HTML générée à partir d'un page JSP



Modèle MVC : vue [JSP]

Pages (JSP): Mise en place



```
Page.jsp
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title> Page JSP </title>
5   </head>
6   <body>
7     <h4>
8       Ceci est une page HTML
9       générée à partir d'un page JSP
10    </h4>
11   </body>
12 </html>
```

```
1 out.write("<!DOCTYPE html>\r\n");
2 out.write("<html>\r\n");
3 out.write("  <head>\r\n");
4 out.write("    <meta charset=\"utf-8\" />\r\n");
5 out.write("    <title>Test</title>\r\n");
6 out.write("  </head>\r\n");
7 out.write("  <body>\r\n");
8 out.write("    <p>Ceci est une page générée depuis une JSP.</p>\r\n");
9 out.write("  </body>\r\n");
10 out.write("</html>");
```



Login.jsp

```
<!DOCTYPE html>
<html>
<body>

    <h1> Welcome
    <%
        String login = request.getParameter("login");
        String pass = request.getParameter("pass");

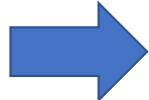
        response.setContentType("text/html");

        out.println(login);

    %>

    ^_ ^ </h1>

</body>
</html>
```



LoginJSP.java

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

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<!DOCTYPE html>");
    out.println("<html>");
    out.println("<body>");
    out.println("<h1> Welcome");

    String login = request.getParameter("login");
    String pass = request.getParameter("pass");

    response.setContentType("text/html");

    out.println(login);

    out.println(" ^_ ^ </h1>");
    out.println("</body>");
    out.println("</html>");

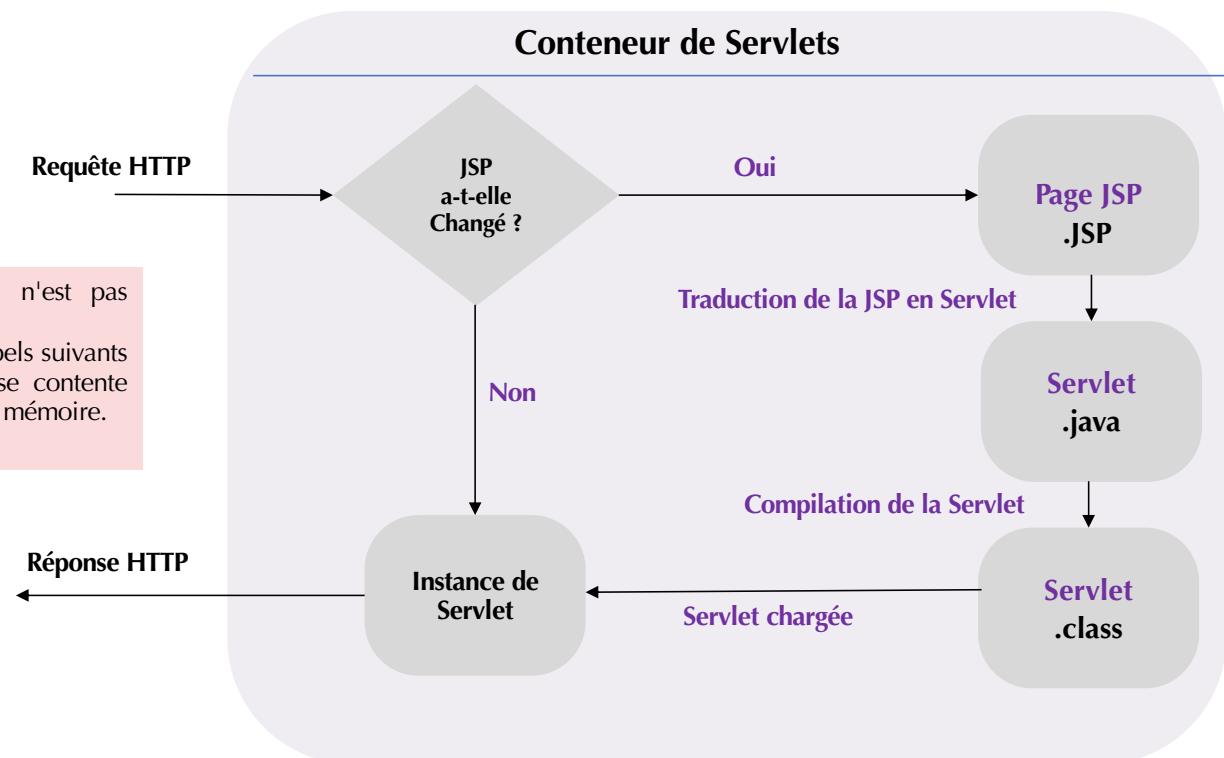
}
```

Modèle MVC : vue [JSP]

Pages (JSP): Cycle de vie

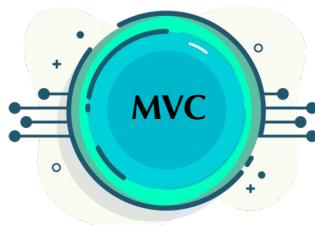
Le processus de **vérification/traduction/compilation** n'est pas effectué à chaque appel !

La servlet générée et compilée étant sauvegardée, les appels suivants à la **JSP** sont beaucoup plus rapides : le conteneur se contente d'exécuter directement l'instance de la servlet stockée en mémoire.



Modèle MVC : vue [JSP]

Pages (JSP): Mise en relation avec notre servlet



Le modèle de conception **MVC** nous recommande en effet la mise en place d'un **contrôleur**, dont nous allons donc tâcher de toujours associer à une **vue**.



Mais on viens de montrer qu'une **JSP** était de toute façon traduite en **servlet**...
Quel est l'intérêt de mettre en place une autre **servlet** ?

les servlets résultant de la traduction des JSP dans une application n'ont pour rôle que de permettre la **manipulation** des **requêtes et réponses HTTP**.

En aucun cas elles n'interviennent dans la couche de contrôle, elles agissent de manière transparente et font bien partie de la vue : ce sont simplement des traductions en un langage que comprend le serveur (le Java !) des vues présentes dans votre application (de simples fichiers textes contenant de la syntaxe **JSP**).



Systématiquement on va créer une **servlet** lorsque nous créerons une page **JSP**.
C'est une bonne pratique à prendre pour garder le contrôle, en s'assurant qu'une vue ne sera jamais appelée par le client sans être passée à travers une servlet.
La servlet est le point d'entrée de notre application
Pour cela on va déplacer nos pages **JSP** dans le répertoire **/WEB-INF**.



Modèle MVC : vue [JSP]

Pages (JSP): Mise en relation avec notre servlet

Cette opération est réalisée depuis la **servlet**, ce qui est logique puisque c'est elle qui décide d'appeler la **vue**

```
protected void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
    this.getServletContext()
        .getRequestDispatcher( "/WEB-INF/Page.jsp" )
        .forward( request, response );
}
```

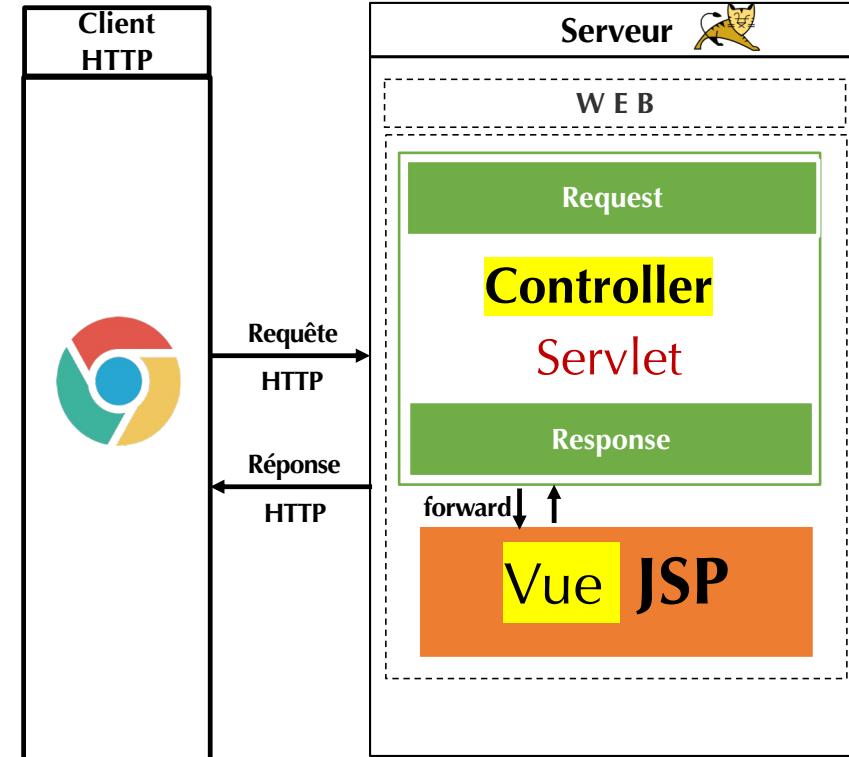
Depuis la servlet (**this**), nous appelons la méthode **getServletContext()**. Celle-ci nous retourne alors un objet **ServletContext**, qui fait référence au contexte commun à toute l'application : qui offre un ensemble de méthodes permettant à une servlet de communiquer avec le conteneur de servlet ;

getRequestDispatcher(), retourne un objet **RequestDispatcher**, qui agit ici comme une **enveloppe** autour de notre page JSP.

Utilisé pour que la servlet soit capable de faire suivre nos objets requête et réponse à une vue.

Il est impératif d'y préciser le chemin complet vers la JSP, en commençant obligatoirement par un /

nous utilisons enfin ce dispatcher pour réexpédier la paire **requête/réponse HTTP** vers notre page JSP via sa méthode **forward()**.



l'objet **HttpServletRequest**, contient lui aussi une méthode **getRequestDispatcher()** qui peut prendre en argument un chemin relatif, alors que sa grande sœur n'accepte qu'un chemin complet.



Modèle MVC : vue [JSP]

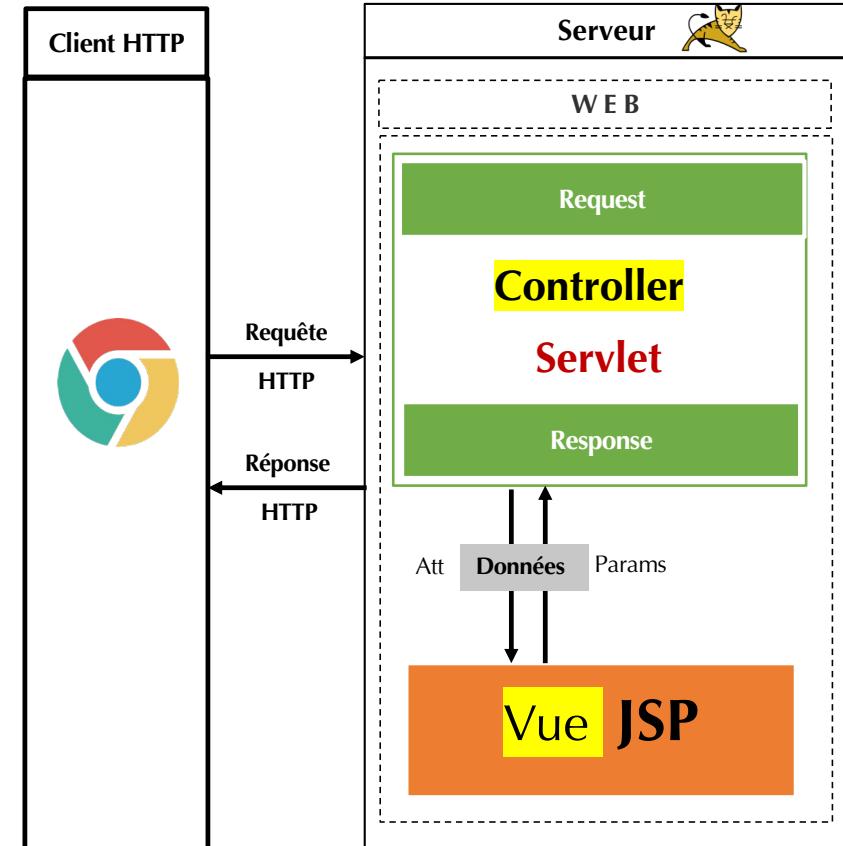
JSP → Servlet: Transmission de données

Notre contenu est présent dans une **page JSP** à laquelle nous avons associé une servlet.

Pour ajouter du dynamisme à notre projet. Il est temps d'apprendre à faire communiquer entre nos composants web les différents données constituant notre application.

Il y a deux types de données à transmettre entre nos composants WEB :

- **Données issues du serveur : les attributs**
- **Données issues du client : les paramètres**



Modèle MVC : vue [JSP]

JSP ↔ Servlet: Données issues du Serveur : les attributs

Transmettre des variables de la servlet à la JSP :

Jusqu'à présent nous n'avons pas fait grand-chose avec notre **requête HTTP**, (l'objet **HttpServletRequest**) nous nous sommes contentés de le transmettre à la JSP.

Puisque notre **requête HTTP** passe maintenant au travers de la **servlet** avant d'être transmise à la vue, profitons-en pour y apporter quelques modifications ! Utilisons donc notre **servlet** pour mettre en place un semblant de dynamisme dans notre application :

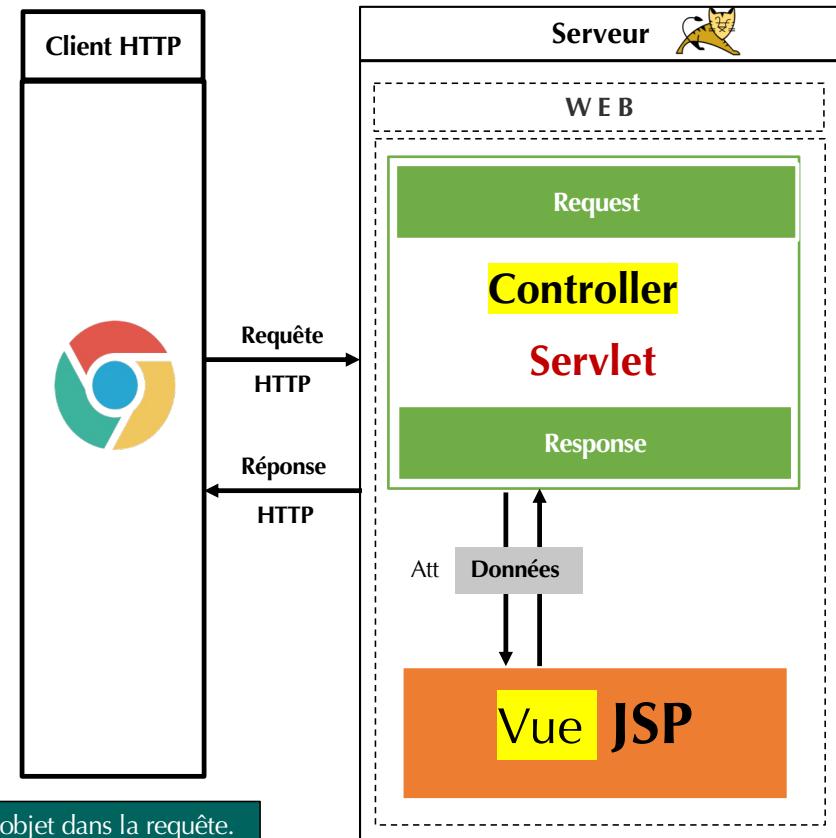
```
protected void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {

    String message = "Transmission de variables réussie : ^_^ !";

    request.setAttribute( "MSG", message );

    this.getServletContext()
        .getRequestDispatcher( "/WEB-INF/Page.jsp" )
        .forward( request, response );
}
```

Mon objet se nomme **message** mais j'ai nommé par la suite **MSG** l'attribut qui contient cet objet dans la requête. Côté vue, c'est par ce nom d'attribut que vous pourrez accéder à votre objet !



Modèle MVC : vue [JSP]

JSP ↔ Servlet: Données issues du Serveur : les attributs

Côté page JSP :

la technologie **JSP**, permet d'inclure du code Java dans notre code html en entourant ce code des balises `<%` et `%>`. Ce sont des marqueurs qui délimitent les portions contenant du code Java du reste de la page.

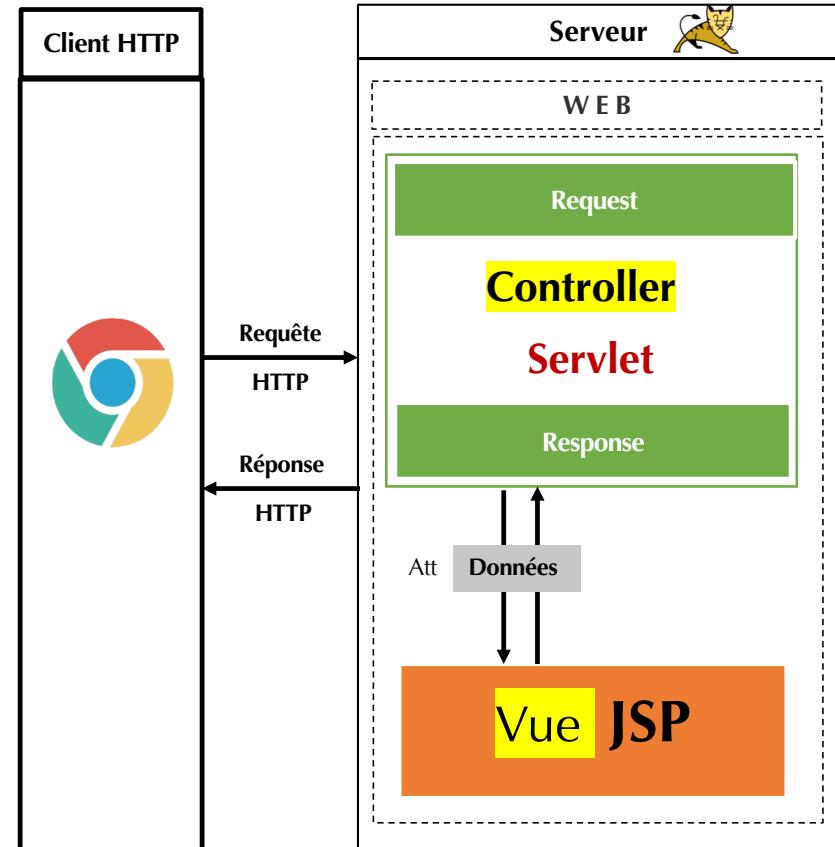
La seule différence réside dans le fait que depuis une **JSP**, il n'est plus nécessaire de récupérer l'objet **PrintWriter**, comme nous l'avions fait depuis notre servlet. Ceci est rendu possible grâce à l'existence d'**objets implicites**, sur lesquels nous allons revenir très bientôt !

Pour récupération de l'attribut depuis la requête, par analogie avec la création de l'attribut, il suffit d'appeler la méthode **getAttribute()** pour le récupérer un depuis une JSP !

```
<!DOCTYPE html>
<html>
    <head>
        <title> Page JSP </title>
    </head>

    <body>
        <h4>
            Ceci est une page HTML
            générée à partir d'un page JSP
        </h4>

        <strong>
            <%
                String attribut = (String) request.getAttribute("MSG");
                out.println( attribut );
            %>
        </strong>
    </body>
</html>
```



Modèle MVC : vue [JSP]

JSP ↔ Servlet: Données issues du Serveur : les attributs

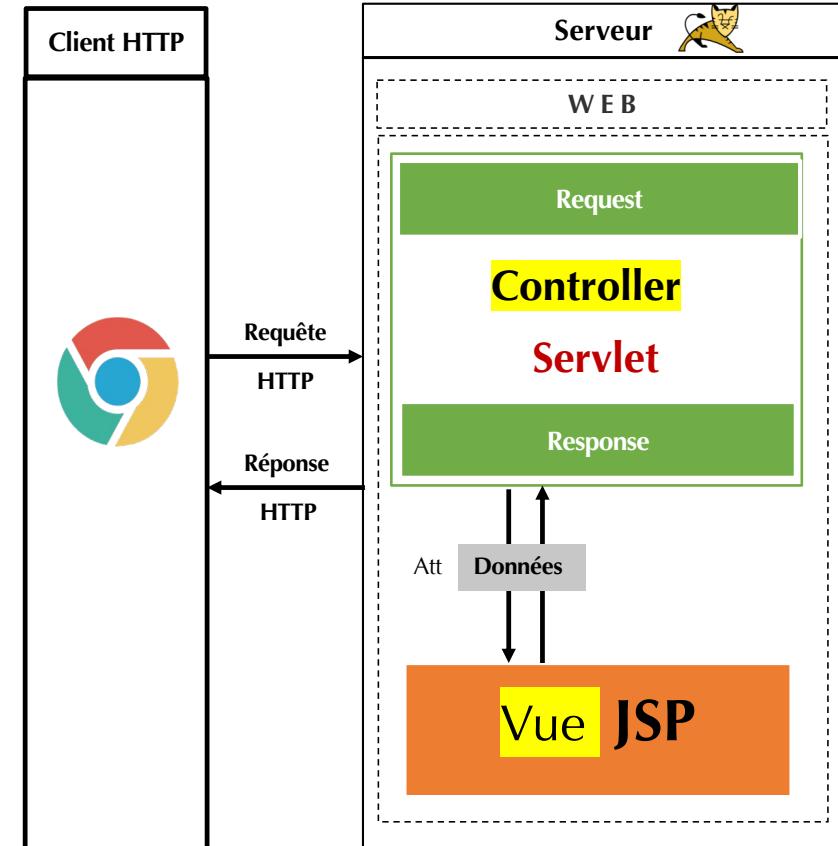
```
<strong>  
    <%  
        String attribut = (String) request.getAttribute("MSG");  
        out.println( attribut );  
    %>  
</strong>  
</body>  
</html>
```

???

Écrire du Java dans une JSP n'a aucun sens : l'intérêt même de ces pages est de s'affranchir du langage Java

Mais cela confirme qu'en Java EE, rien n'impose au développeur de bien travailler comme il veut, il est possible de coder n'importe comment sans que cela n'impacte le fonctionnement de l'application.

On a utiliser du code Java ici, parce que nous n'avons pas encore découvert le langage **JSP**. D'ailleurs, à partir du chapitre suivant, nous allons tout mettre en œuvre pour ne plus jamais écrire de Java directement dans une JSP !



Modèle MVC : vue [JSP]

JSP → Servlet: Données issues du Client : les paramètres

Paramètre de requête :

la méthode **GET** du protocole **HTTP** permet au client de transmettre des données au serveur en les incluant directement dans l'**URL**, dans ce qui s'appelle les **paramètres** ou **query strings** en anglais.

<!-- URL sans paramètres -->

/page.jsp

<!-- URL avec un paramètre nommé 'lg' et ayant pour valeur 'java' -->

/page.jsp?lg=java

<!-- URL avec deux paramètres nommés 'lang' et 'admin', et ayant pour valeur respectivement 'fr' et 'true'-->

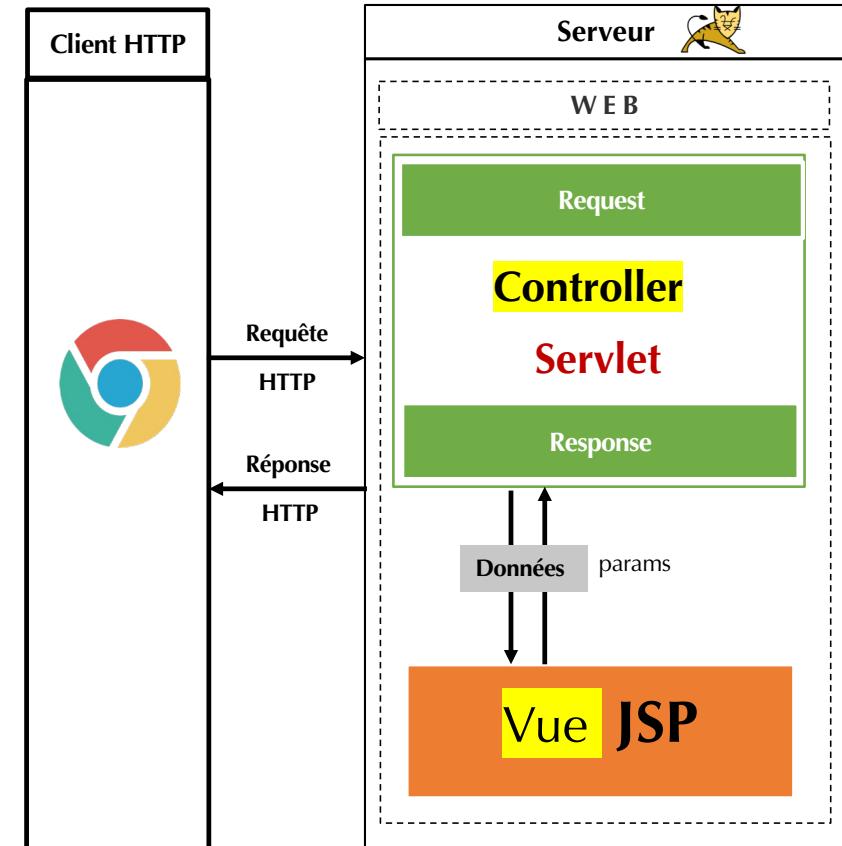
/page.jsp?lang=fr&admin=true

la taille d'une URL étant limitée, la taille des données qu'il est ainsi possible d'envoyer est limitée également !

Autrement dit si votre URL est si longue qu'elle contient plus de 2 000 caractères par exemple, ce navigateur ne saura pas gérer cette URL !

Préférez dans ce cas la méthode **POST**

Si vous envoyez des données ayant un impact sur la ressource demandée, il est, là encore, préférable de passer par la méthode **POST** du protocole, plutôt que par la méthode **GET**.



Modèle MVC : vue [JSP]

JSP → Servlet: Données issues du Client : les paramètres

Avoir un impact sur la ressource :

Cela veut dire "**entraîner une modification sur la ressource**", mais en fin de compte tout dépend de ce que vous faites de ces données dans votre code.

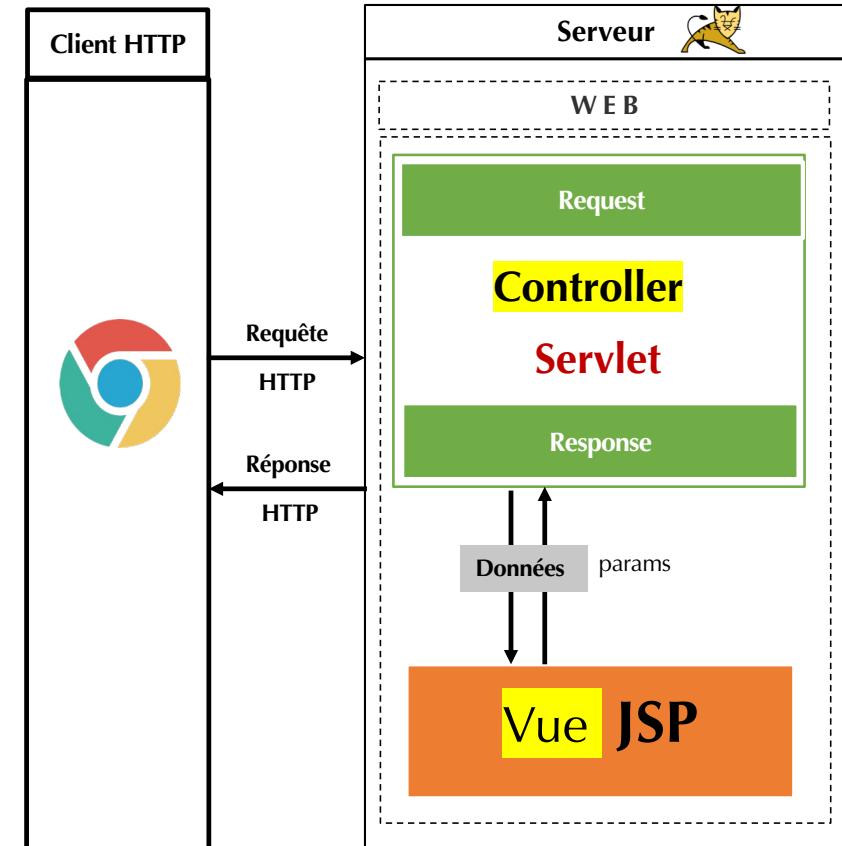
Exemple :

Imaginons une application proposant une page **compte.jsp** qui autoriserait des actions diverses sur le compte en banque de l'utilisateur.

L'envoi d'un paramètre **mois** pour afficher la liste des E/S d'argent du compte, i.e. **compte.jsp?mois=avril**, n'aura pas d'impact sur la ressource. Même si on renvoie 10 fois la requête au serveur, notre code ne fera que réafficher les mêmes données sans les modifier.

Par contre l'envoi de paramètres précisant des informations nécessaires pour faire un transfert d'argent, i.e. **compte.jsp?montant=100&destinataire=01K87B612**, aura clairement un impact sur la ressource : en effet, si nous renvoyons 10 fois une telle requête, notre code va effectuer 10 fois le transfert !

nous pouvons utiliser une requête **GET** pour le premier cas, et devons utiliser une requête **POST** pour le second.



Modèle MVC : vue [JSP]

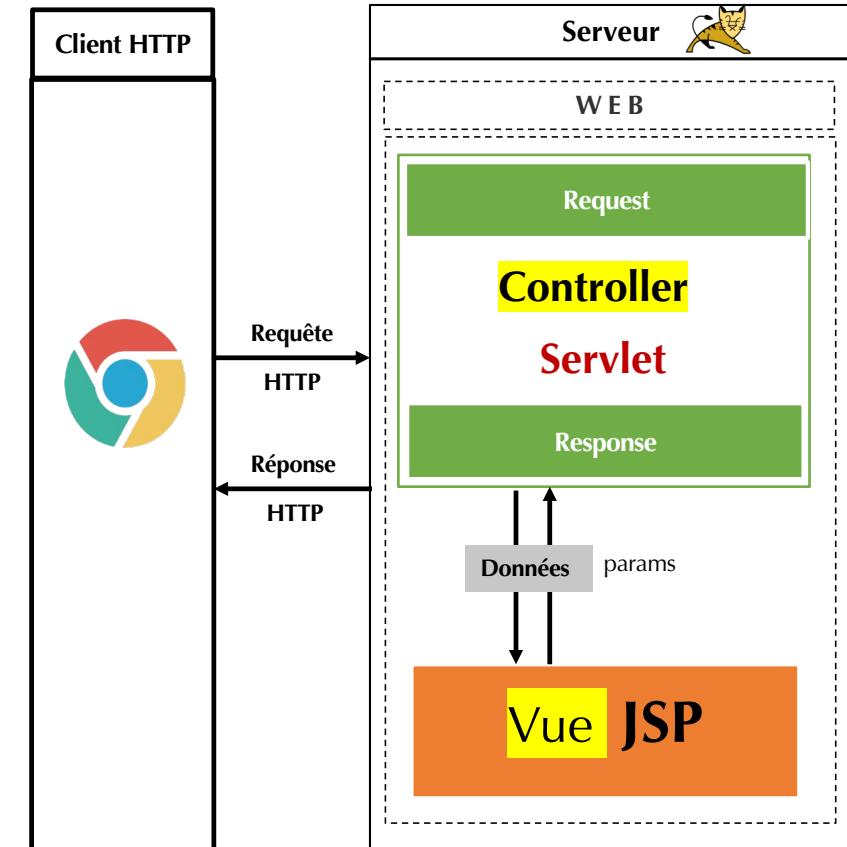
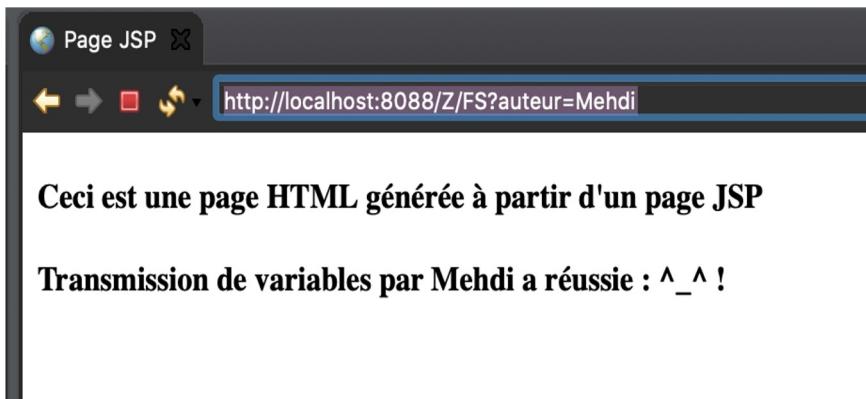
JSP → Servlet: Données issues du Client : les paramètres

Récupération des paramètres par le serveur :

Si vous appelez à nouveau votre **servlet** depuis votre navigateur, en ajoutant un paramètre nommé **auteur** à l'URL, par exemple :

`http://localhost:8080/test/FS?auteur=Mehdi`

Alors vous observerez que le message affiché dans le navigateur contient bien la valeur du paramètre précisé dans l'URL



Modèle MVC : vue [JSP]

JSP → Servlet: Données issues du Client : les paramètres

Récupération des paramètres par le serveur :

Nous pouvons très bien y accéder depuis notre page **JSP** sans passer par la **servlet**, de la manière suivante :

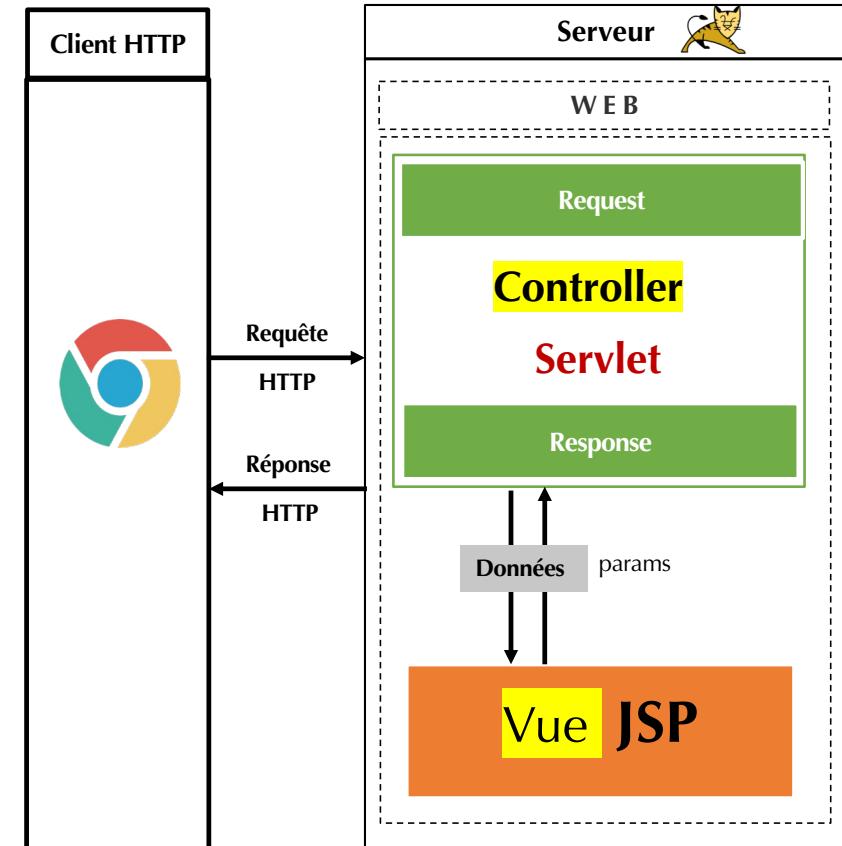
`http://localhost:8080/test/Page.jsp?auteur=Mehdi`

```
<strong>
    <%>
        String attribut = (String) request.getAttribute("MSG");
        out.println( attribut );

        String parametre = request.getParameter( "auteur" );
        out.println( parametre );

    </strong>
    %>
```

Mais c'est une bonne pratique de toujours contrôler ce qu'on affiche au client



Technologie JSP : Langage

Les bases du langage de la technologie JSP

1. Les balises
2. Les directives d'une page
 1. Directives d'importation, d'inclusion et de définition
3. La portée ou visibilité des objets dans une page JSP
4. Les actions dites standard de la technologie JSP
5. Les expressions EL



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les Balises de bases

Balises de scriptlet : <% %>

Venant des mots "script" et "servlet", à l'intérieur d'une scriptlet se cache simplement du code Java.

Cette balise que nous avons déjà utilisée sert en effet à inclure du code Java au sein de vos page JSP.

Boucle java pour remplir la liste de choix des jours

The screenshot shows a JSP page with a form. On the left, there is a dropdown menu labeled "Jours :" with "Lundi" selected. Below it is a blue button labeled "Valider". To the right, another dropdown menu is shown with a tooltip containing the following Java code:

```
Lundi
Mardi
Mercredi
Jeudi
Vendredi
Samedi
Dimanche
```

Exemple

```
<%@ page pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head> <title>Test</title> </head>
    <body>

        <form action="#" method="post">
            <label> Jours : </label>
            <select>

                > <% String[] Jours = {"Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"}; %>

                <% for(int i = 0; i< Jours.length; i++) { %>
                    <option> + Jours[i] + </option>
                <% } %>

            </select>
        </form>
        <input type="submit" value="Valider" />

    </body>
</html>
```



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les Balises de bases

Balises de déclaration : <%! %>

Scriptlet permettant de déclarer une variable ou une méthode.

Il est possible d'effectuer plusieurs déclarations au sein d'un même bloc

Déclaration d'une chaîne et d'une méthode

Test d'appel à la méthode Inverser

"L'inverse de EMSI est ISME"

.JSP

Exemple

```
<%! String Message = "E M S I";  
  
public String Inverser(String S) {  
    String resultat = "";  
    for(int i = (S.length()-1) ; i >= 0 ; i--)  
        resultat += S.charAt(i);  
    return resultat;  
}  
  
>%>  
  
<span>    <% out.print(" L'inverse de " + Message +  
        " est " + Inverser(Message)); %>  
</span>
```



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les Balises de bases

Balises d'expression : <%= %>

C'est un raccourci de la scriptlet suivante :

Elle retourne le contenu d'une chaîne :

```
<%  out.println( "Bonjour ^_^" ); %>
```

```
<%= "Bonjour ^_^" %>
```

Balises de commentaire : <%-- --%>

```
<%-- Ceci est un commentaire JSP, non visible dans la page HTML finale. --%>
<%= "Bonjour ^_^" %>
```



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Les directives contrôlent comment le conteneur de servlets va gérer votre JSP.

Il en existe trois : **taglib**, **page** et **include**.

À travers les quelles on va pouvoir :

- Importer un package ;
- Inclure d'autres pages JSP ;
- Inclure des bibliothèques de balises (nous y reviendrons dans un prochain chapitre) ;
- Définir des propriétés et informations relatives à une page JSP.

Les directives sont comprises entre les balises `<%@` et `%>`, et hormis la directive d'**inclusion** de page qui peut être placée n'importe où, **elles sont à placer en tête** de page JSP.



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Directive taglib : `<%@ taglib %>`

Utilisé pour inclure et utiliser une bibliothèque personnalisée dans vos page JSP

Exemple d'inclusion d'une bibliothèque personnalisée nommée **myLib** avec un préfix **ml**

```
<%@ taglib uri="myLib.tld" prefix="ml" %>
```

Directive page : `<%@ page %>`

Définit des informations relatives à la page JSP.

Exemple d'importation de deux classes Java :

```
<%@ page import="java.util.List, java.util.Date" %>
```

Cette fonctionnalité n'est utile que si vous mettez en place du code Java dans votre JSP puisque notre objectif est de faire disparaître le Java de nos vues, nous allons très vite apprendre à nous en passer !



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Directive page : <%@ page %>

Il existe encore plus un ensemble des propriétés accessibles via cette directive **page** :

```
<%@ page
    language      = "           "
    extends       = "           "
    import        = "           "
    session       = "   true || false   "
    isThreadSafe  = "   true || false   "
    isELIgnored   = "   true || false   "
    info          = "           "
    errorPage     = "           "
    contentType   = "           "
    pageEncoding  = "           "
    isErrorPage   = "   true || false   "
%>
```

Exemple du **pageEncoding**. Qui s'ajoute automatiquement pour spécifier l'encodage « **UTF-8** »

```
<%@page pageEncoding="UTF-8"%>
```



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Directive page : <%@ page errorPage %> et <%@ page isErrorPage %>

Page.jsp

```
<%@ page errorPage="error.jsp" %>
<!DOCTYPE html>
<html>
  <head>
    <title>Test Error Tag</title>
  </head>

  <body>
    <h1> Welcome </h1>
    <div>
      <%= 1/0 %>    <%-- L'erreur affichera la page error.jsp --%>
    </div>
  </body>
</html>
```

error.jsp

```
<%@ page isErrorPage="true" %>
<div style="text-align: center; font-family: Optima;">
  <h1 style="color: white; background-color: red; font-size: 50px;">
    Error
  </h1>
  <h2 style="color: red; background-color: yellow;">
    <%= exception.getMessage() %>
    <%-- le fait de faire true en haut nous permet
       d'utiliser un objet exception
    --%>
  </h2>
</div>
```

Error

/ by zero



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Directive include : <%@ include %>

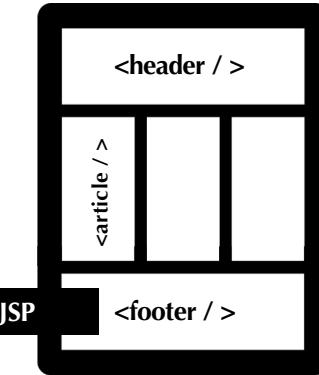
Nos vues correspondent rarement à une **JSP** constituée d'un **seul bloc**.

Cela permet notamment de pouvoir réutiliser certains blocs dans plusieurs vues différentes (ex. Les menus...)

Pour permettre un tel découpage, la technologie **JSP** met à votre disposition une balise qui inclut le contenu d'un autre fichier dans le fichier courant.

Via le code suivant par exemple, on peut inclure une feuille de style **css** interne dans notre page **JSP**. (mais cela pourrait très bien être une page **HTML** ou **JSP** ou autre)

```
<style>
    %@ include file="css/bootstrap.min.css" %
</style>
```



En pratique, il est très courant de découper littéralement une page web en plusieurs fragments, qui sont ensuite rassemblés dans la page finale à destination de l'utilisateur.



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

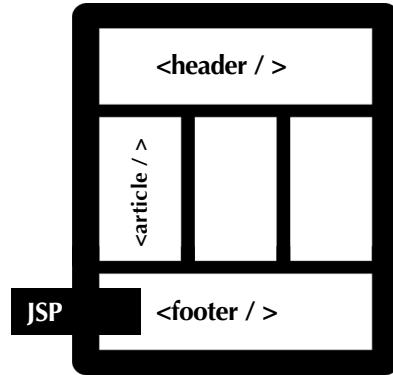
Directive include : <%@ include %>

La subtilité à retenir, c'est que cette directive ne doit être utilisée que pour inclure du contenu "statique" dans votre page : i.e. **headers** ou le **footers** de la page, très souvent identiques sur l'intégralité des pages du site.

Statique : l'inclusion est réalisée au moment de la **compilation** ;

Alors si le code du fichier est changé par la suite, les changements sur la page l'incluant n'auront lieu qu'après **une nouvelle compilation** !

Cette directive peut être vue comme un simple **copier-coller** d'un fichier dans l'autre

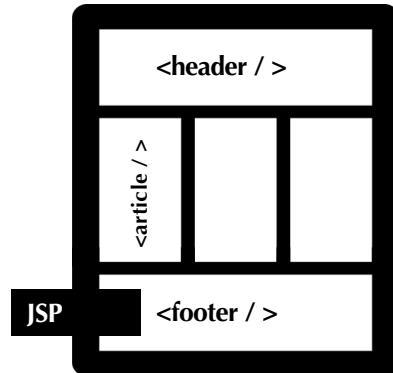
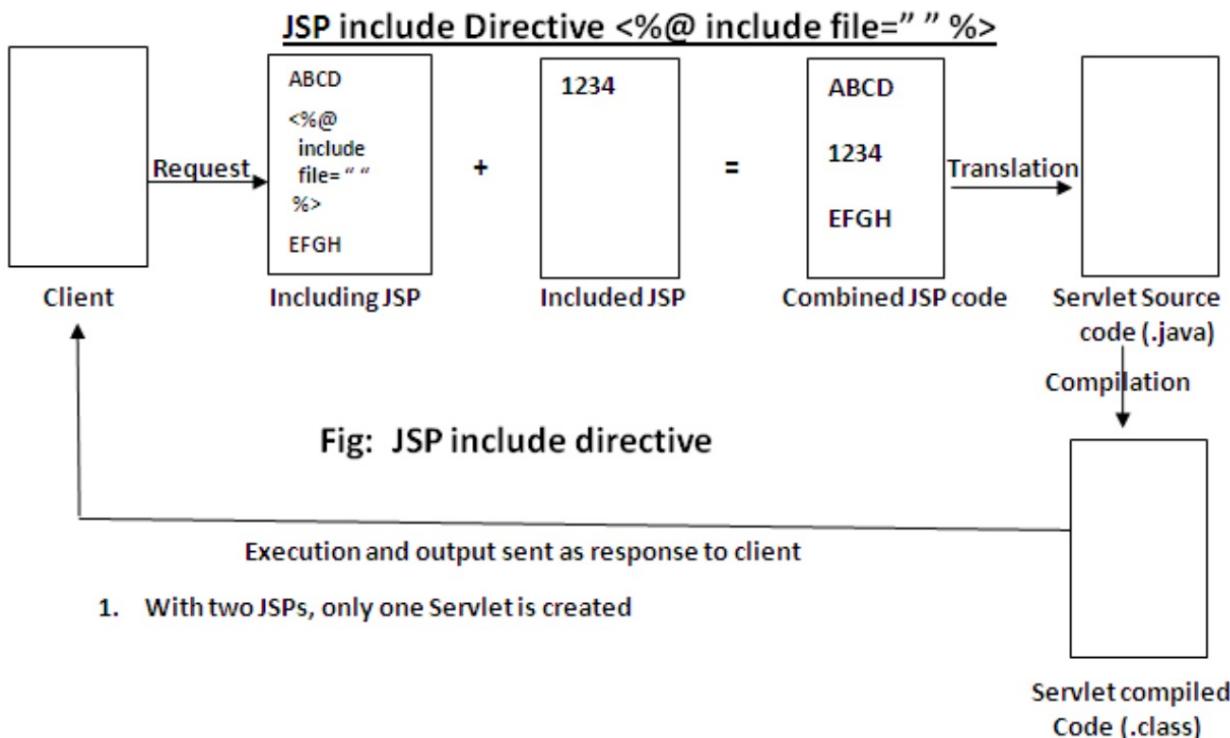


Modèle MVC : vue [JSP]

Technologie JSP : Langage

Les directives JSP :

Directive include : <%@ include %>



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Include dynamique avec Action standard :

Action standard include : <jsp:include page="page.jsp" %>

Une autre balise d'inclusion dite "standard" existe, et permet d'inclure du contenu de manière "dynamique". (**après la compilation**)
Le contenu sera ici chargé à l'**exécution**, et non à la **compilation** comme c'est le cas avec la directive précédente :

```
<%-- L'inclusion dynamique d'une page fonctionne par URL relative : --%>
<jsp:include page="page.jsp" />

<%-- Son équivalent en code Java est : --%>
<% request.getRequestDispatcher( "page.jsp" ).include( request, response ); %>

<%-- Et il est impossible d'inclure une page externe comme ci-dessous : --%>
<jsp:include page="http://www.emsi.ma" />
```

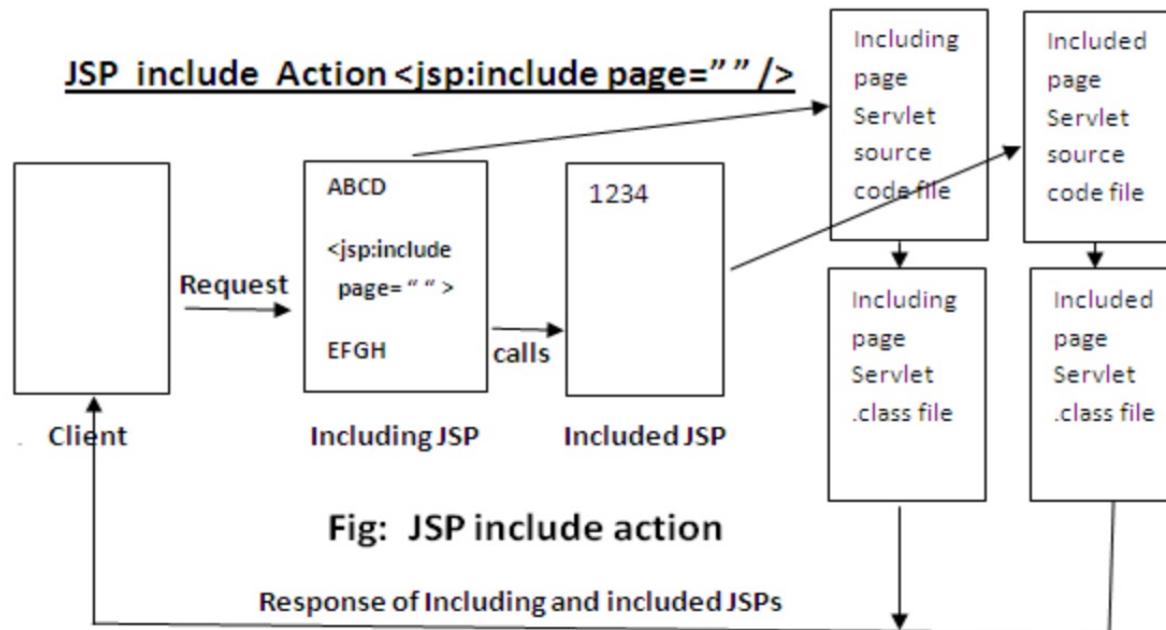


Modèle MVC : vue [JSP]

Technologie JSP : Langage

Include dynamique avec Action standard :

Action standard include : <jsp:include page=" " %>



1. With two JSPs, two Servlets are created



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Include dynamique avec Action standard :

Action standard include : <jsp:include page "%>

Inconvénient : Ce type d'inclusion ne prend pas en compte les imports et inclusions faits dans la page réceptrice.

List.jsp

```
<%  
    ArrayList<String> Days = new ArrayList<String>();  
  
    Days.add( "Lundi" );  
    Days.add( "Mardi" );  
    Days.add( "Mercredi" );  
    Days.add( "Jeudi" );  
    Days.add( "Vendredi" );  
    Days.add( "Samedi" );  
    Days.add( "Dimanche" );  
  
%>  
  
<h2>  
    <%  
        for( int i = 0 ; i < Days.size() ; i++ )  
            out.println("<li>" + Days.get( i ) + "</li>");  
    %>  
</h2>
```

Test.jsp

```
<%@page import="java.util.ArrayList"%>  
<!DOCTYPE html>  
<html>  
    <head>  
        <title>Test d'inclusion</title>  
    </head>  
  
    <body>  
        <%@ include file="List.jsp" %>  
    </body>  
</html>
```

output

- Lundi
- Mardi
- Mercredi
- Jeudi
- Vendredi
- Samedi
- Dimanche

Avec la directive **include** ça marche très bien



Modèle MVC : vue [JSP]

Technologie JSP : Langage

Include dynamique avec Action standard :

Action standard include : <jsp:include page %>

Inconvénient : Ce type d'inclusion ne prend pas en compte les imports et inclusions faits dans la page réceptrice.

List.jsp

```
<%
    ArrayList<String> Days = new ArrayList<String>();
    Days.add( "Lundi" );
    Days.add( "Mardi" );
    Days.add( "Mercredi" );
    Days.add( "Jeudi" );
    Days.add( "Vendredi" );
    Days.add( "Samedi" );
    Days.add( "Dimanche" );

%>
<h2>
    <%
        for( int i = 0 ; i < Days.size() ; i++ )
            out.println("<li>" + Days.get( i ) + "</li>");
    %>
</h2>
```

Test.jsp

```
<%@page import="java.util.ArrayList"%>
<!DOCTYPE html>
<html>
    <head>
        <title>Test d'inclusion</title>
    </head>
    <body>
        <jsp:include page="List.jsp"/>
    </body>
</html>
```

output

```
java.lang.NullPointerException
```

Avec l'action standard **include** ça tient pas compte des l'import

- Les pages incluses via la balise `<jsp:include ... />` doivent en quelque sorte être "**indépendantes**" ; elles ne peuvent pas dépendre les unes des autres et doivent pouvoir être compilées séparément.
- Ce n'est pas le cas des pages incluses via la directive `<%@ include ... %>`



Modèle MVC : vue [JSP]

Technologie JSP : Langage

La gestion des objets par la technologie JSP

L'un des concepts les plus importants qui intervient dans la gestion des objets par la technologie JSP est **la portée des objets ou visibilité, ou scope** en anglais, qui définit tout simplement leur **durée de vie**.

Lors de la **transmission de données** de la servlet vers nos pages JSP, on a utilisé les attributs de requête. Tels objets sont accessibles via l'objet **HttpServletRequest**, et ne sont visibles que durant le traitement d'une même requête. Ils sont créés par le conteneur lors de la réception d'une **requête HTTP**, et disparaissent dès lors que le traitement de la requête est terminé.

Du coup, on a donc créé, sans le savoir, des objets ayant pour portée la requête !



Technologie JSP : Langage

La gestion des objets par la technologie JSP

La portée des objets

Il existe au total **quatre portées** différentes dans une application JEE:

- **page (JSP seulement)** : les objets dans cette portée sont uniquement accessibles dans la page JSP en question ;
 - **JSP seulement** veut dire qu'il n'est possible de créer et manipuler des objets de portée **page** que depuis une page JSP, ce n'est pas possible via une servlet.
 - Alors qu'il est possible de créer et manipuler des objets de portées **requête**, **session** ou **application** depuis une page **JSP** ou depuis une servlet.
- **requête** : les objets dans cette portée sont uniquement accessibles durant l'existence de la requête en cours ;
- **session** : les objets dans cette portée sont accessibles durant l'existence de la session en cours ;
 - Une session est l'objet associé à un utilisateur, elle existe tant qu'il utilise l'application, elle peut expirer lorsqu'il ferme son navigateur, reste inactif trop longtemps, ou encore lorsqu'il se déconnecte.
 - une session correspond en réalité à un navigateur particulier, plutôt qu'à un utilisateur.
- **application** : les objets dans cette portée sont accessibles durant toute l'existence de l'application.

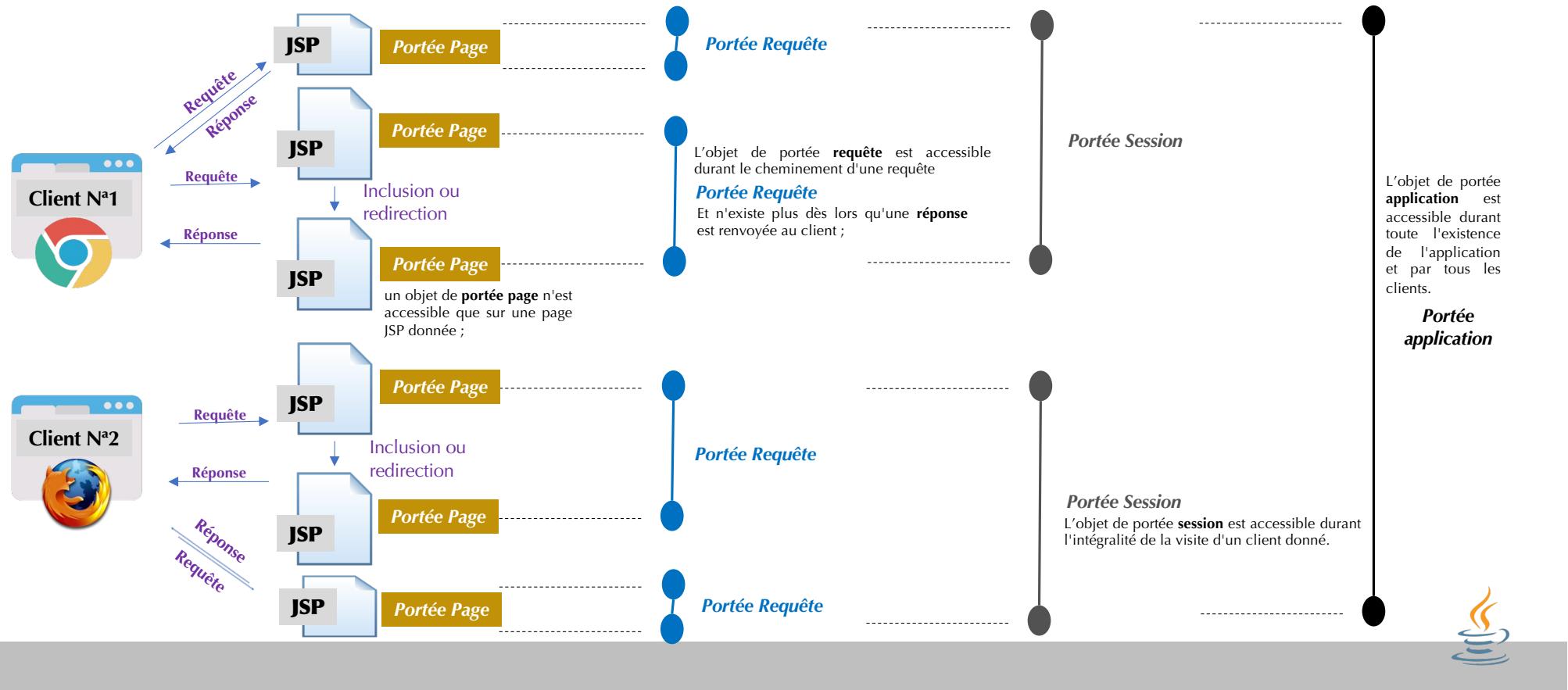


Modèle MVC : vue [JSP]

Technologie JSP : Langage

La portée des objets

Pour visualiser bien le principe, voici un schéma regroupant les différentes portées existantes :



Modèle MVC : vue [JSP]

La portée des objets

Exemple pratique

Login.jsp

On envoi le string **login** comme paramètre de **requête** vers la servlet TestServlet

```
<!DOCTYPE html>
<html>
    <head>
        <title>Login</title>
    </head>

    <body >
        <a href="Home.jsp">Home Page</a>
        <h5>Connected as : <%=session.getAttribute("login")%></h5>

        <form action="ts" method="post">
            <span> UserName : </span>
            <input type="text" name="login"/>
            <button> Connect </button>
        </form>
    </body>
</html>
```

Home Page

Connected as :

UserName : Connect



Modèle MVC : vue [JSP]

La portée des objets Exemple pratique

On crée une session et on stocke le paramètre envoyé dans la session

TestServlet.java

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

    private static final String VUE = "Test.jsp";

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

        String lg = request.getParameter("login");
        HttpSession session = request.getSession();
        session.setAttribute("login", lg);
        request.getRequestDispatcher(VUE).forward(request, response);
    }
}
```



Modèle MVC : vue [JSP]

La portée des objets Exemple pratique

Login.jsp

On affiche la donnée du login stocké dans la session

```
<!DOCTYPE html>
<html>
    <head>
        <title>Login</title>
    </head>

    <body >
        <a href="Home.jsp">Home Page</a>
        <h5>Connected as : <%=session.getAttribute("login")%></h5>

        <form action="ts" method="post">
            <span> UserName : </span>
            <input type="text" name="login"/>
            <button> Connect </button>
        </form>
    </body>
</html>
```

La valeur sauvegardé dans la session reste affiché durant toute la session de l'utilisateur.
Vous pouvez tester en faisant un rafraîchissement de la page

Home Page

Connected as : Admin

UserName :

Connect



Modèle MVC : vue [JSP]

La portée des objets

Exemple pratique

Home.jsp

On peut même tester avec une autre page accessible par cet utilisateur

```
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>

    <body>
        <a href="Login.jsp">Login Page</a>
        <h5>Connected as : <%=session.getAttribute("login")%></h5>

        <h1> Welcome to Home Page </h1>
    </body>
</html>
```

Login Page

Connected as : Admin

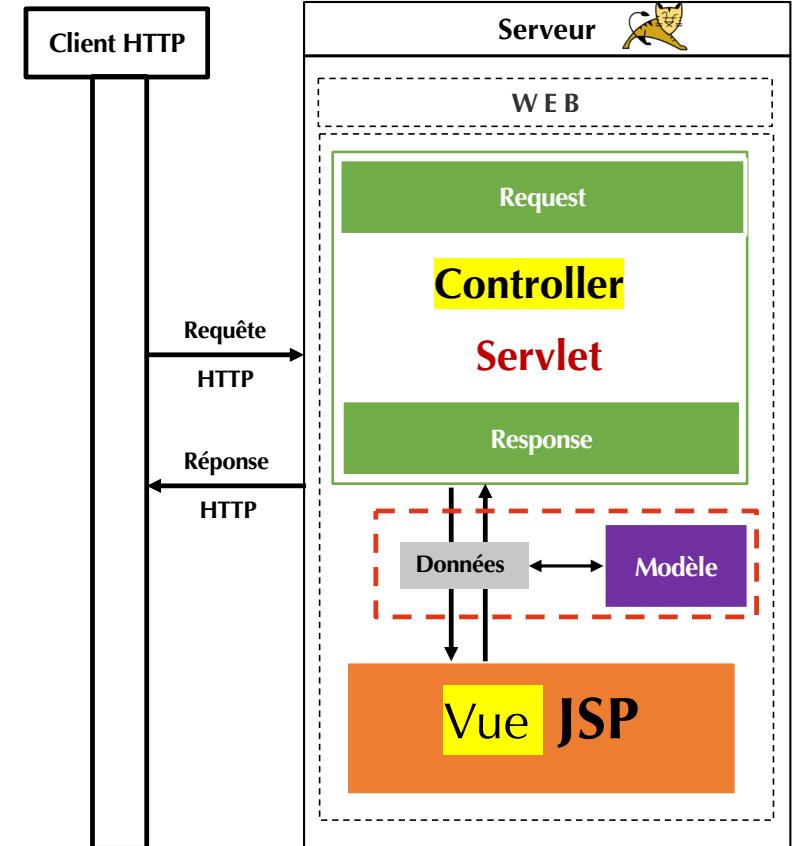
Welcome to Home Page



Modèle MVC : le modèle [JavaBean]

JavaBean : Composant Réutilisable pour représenter les données

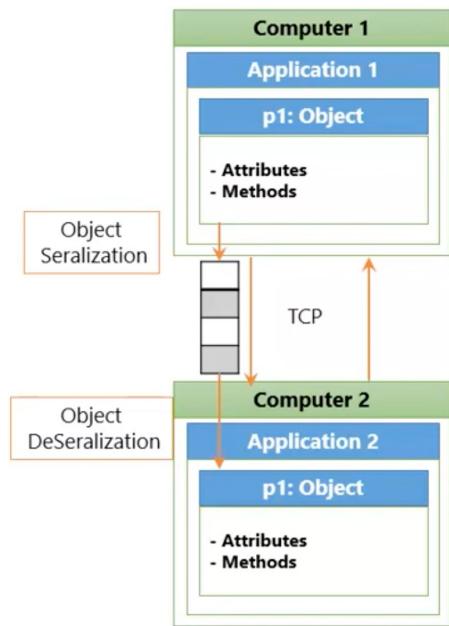
- un **JavaBean** désigne tout simplement un **composant réutilisable**.
- Il est construit selon certains standards des spécifications du langage Java
- un **JavaBean** n'a donc rien de spécifique au Java EE.
- C'est un simple objet Java qui suit **certaines contraintes** :
 - **Paramétrable** : un Bean est conçu pour être **paramétrable** à travers les "**propriétés**" ou champs **non publics** présents dans un Bean (type primitif ou objets)
 - **Persistant** : un Bean est conçu pour être **sérialisable**. La **sérialisation** est un processus qui permet de sauvegarder l'état d'un Bean, et donne ainsi la possibilité de le restaurer par la suite. Ce mécanisme permet une persistance des données, voire de l'application elle-même
 - **Réutilisable** : un Bean est un composant conçu pour être **réutilisable**. Ne contenant que des données ou du code métier, un tel composant n'a en effet pas de lien direct avec la couche de présentation, ni avec la couche DAO, c'est cette indépendance qui lui donne ce caractère réutilisable.
 - **Dynamique** : L'**introspection** est un processus qui permet de connaître le contenu d'un composant (**attributs, méthodes**) de manière **dynamique**, sans disposer de son code source. C'est ce processus, couplé à certaines règles de normalisation, qui rend possible une **découverte** et un **paramétrage dynamique** du Bean !



Modèle MVC : le modèle [JavaBean]

JavaBean : Composant Réutilisable pour représenter les données

Sérialisation dans les application distribué



```
package model;

import java.io.Serializable;

public class Bean implements Serializable{
    private static final long serialVersionUID = 1L;

    /* Les champs de l'objet ne sont pas publics (ce sont donc des propriétés) */
    private String propertyOne;
    private int     propertyTwo;

    /* Un constructeur par défaut public et sans paramètre. */
    public Bean() {}

    /* Getters et Setters */

    public String getPropertyOne() { return this.propertyOne; }
    public int   getPropertyTwo() { return this.propertyTwo; }

    public void setPropertyOne(String pptOne) { this.propertyOne = pptOne; }
    public void setPropertyTwo(int pptTwo)    { this.propertyTwo = pptTwo; }
}
```



Modèle MVC : le modèle [JavaBean]

JavaBean : Composant Réutilisable pour représenter les données

Structure d'un JavaBean

- un JavaBean :
 - doit être une **classe publique** ;
 - doit avoir au moins un constructeur par défaut, **public** et sans paramètres. Java l'ajoutera de lui-même si aucun constructeur n'est explicité ;
 - peut implémenter l'interface **Serializable**, il devient ainsi persistant et son état peut être sauvegardé ;
 - **ne doit pas avoir de champs publics** ;
 - peut définir des propriétés (des champs non publics), qui doivent être accessibles via des méthodes publiques **getter** et **setter**, suivant des règles de nommage.

```
package model;

import java.io.Serializable;

public class Bean implements Serializable{
    private static final long serialVersionUID = 1L;

    /* Les champs de l'objet ne sont pas publics (ce sont donc des propriétés) */
    private String propertyOne;
    private int    propertyTwo;

    /* Un constructeur par défaut public et sans paramètre. */
    public Bean() {}

    /* Getters et Setters */

    public String getPropertyOne() { return this.propertyOne; }
    public int   getPropertyTwo() { return this.propertyTwo; }

    public void setPropertyOne(String pptOne) { this.propertyOne = pptOne; }
    public void setPropertyTwo(int pptTwo)    { this.propertyTwo = pptTwo; }
}
```



Modèle MVC : le modèle [JavaBean]

JavaBean : Composant Réutilisable pour représenter les données

Mise en place des JavaBean

- Afin de rendre vos objets accessibles à votre application, il faut que les classes compilées à partir de vos fichiers sources soient placées dans un dossier "classes", lui-même placé sous le répertoire /WEB-INF.
 - Par défaut **Eclipse**, ne procède pas ainsi et envoie automatiquement vos classes compilées dans un dossier nommé "**build**".
 - Afin de changer ce comportement, il va falloir modifier le **Build Path** de notre application.
-
- C'est ici qu'il faut préciser le chemin vers **WEB-INF/classes** afin que nos classes, lors de leur compilation, soient automatiquement déposées dans le dossier pris en compte par notre serveur d'applications.

