

Jakarta EE

M2i

Alan Piron-Lafleur

Découverte



Découverte

JEE est devenu la plateforme de choix des grandes entreprises pour développer notamment des sites webs solides, robustes et bien structurées.

Très utilisé en finance notamment de par sa maturité professionnelle.

JEE est basé sur ... Java ! Auquel on a ajouté un ensemble de bibliothèques qui dote Java d'un ensemble de fonctionnalités.

JEE est conçu comme un langage. On peut le comparer à Django (Python), ASP.NET ou Rails (Ruby)

Echange client / serveur

Tout part du client qui fait une **requête HTTP://** vers notre serveur en tapant une url dans sa barre de navigation. On appelle cela une architecture n-tiers “Client-serveur”.



Serveur

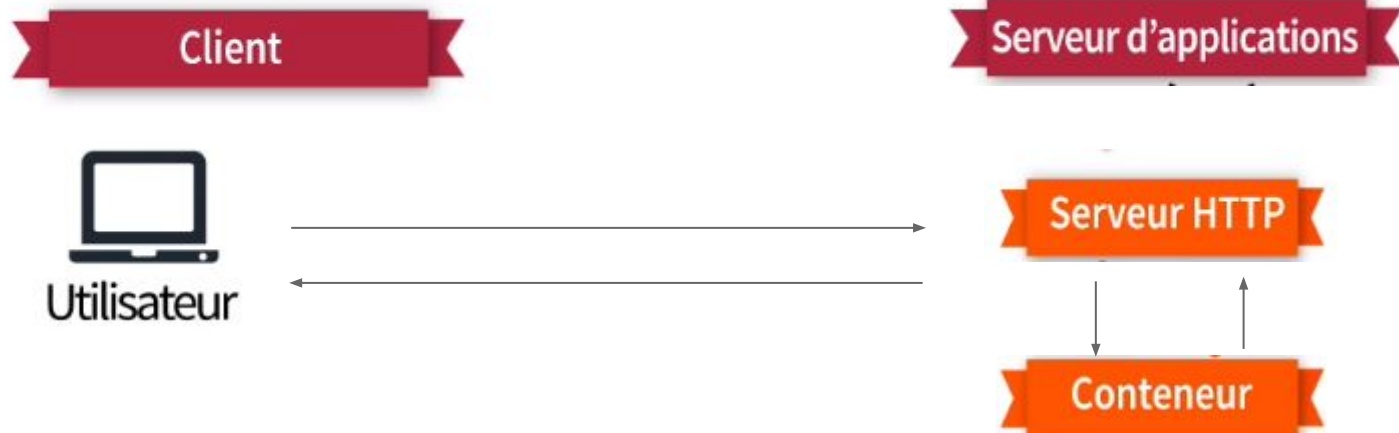
Puis le serveur fait sa tambouille interne (ce que vous allez coder en faite) pour retourner une **réponse HTTP** au client (le site).

Découverte

Dans le détail, notre client va faire une requête HTTP au serveur.

Puis le serveur va lire la requête et l'analyser puis l'envoyer au conteneur.

Le rôle du conteneur, c'est d'exécuter votre code JEE.



Cet ensemble (le serveur d'applications), il en existe plusieurs. Nous, nous utiliserons Tomcat.

Modèle MVC

Modèle MVC

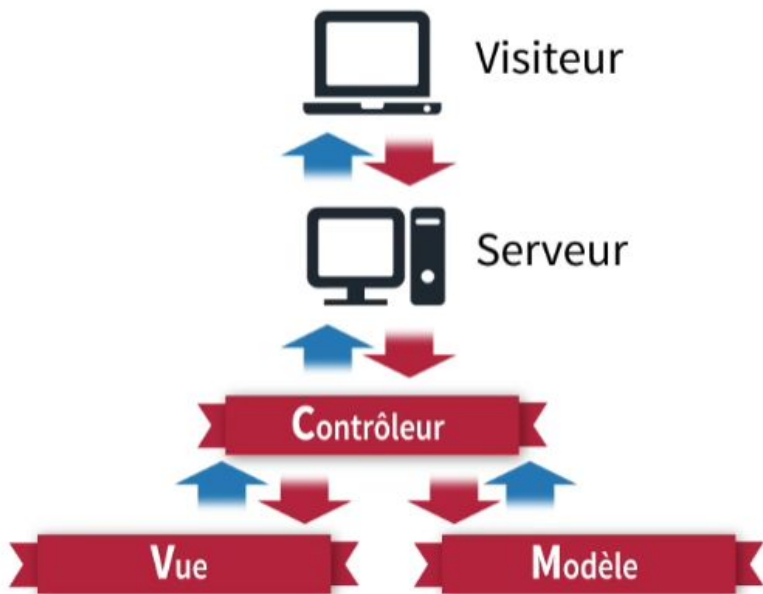
JEE n'impose aucun rangement pour notre code. Autrement dit, nous pourrions faire ce que nous voulons avec nos fichiers et nos appels dans notre code.

Pour pallier à ce problème majeur que poserait un tri anarchique, les développeurs du monde entier utilisent un design pattern bien connu : le système MVC.

Le Modèle Vue Controller est une manière de découper son code proprement.

Modèle MVC

En une infographie, ça donne ça :



Modèle MVC

Dans JEE, chacun des composants du MVC porte un nom spécifique.

Contrôleur = Servlet (s'occupe également du routage)

Modèle = Objets Java (Java BEAN)

Vue = pages JSP (utilisera du code HTML et du code Java)

Modèle MVC

Pour nous aider à développer avec une archi MVC, il existe plusieurs frameworks déjà conçus en MVC. On retrouvera :



Struts



Mais nous, dans ce cours là, nous utiliserons JEE pur, pour comprendre toutes les bases. Si vous voulez ensuite aller plus loin, libre à vous d'apprendre un framework.



Installer son environnement de dev

Un IDE

- Rdv sur eclipse.org pour récupérer l'IDE.
- Lancez l'installateur et choisissez Eclipse IDE for Enterprise Java and Web Developers
- Choisissez la JRE la plus récente lors de l'installation
- Optionnellement, choisissez un dossier pour l'installation
- Installez
- Faites "Launch"
- Choisissez un workspace (le dossier où sera situé votre code); N'oubliez pas où vous l'avez placé.



Un IDE

Rdv dans le menu :

sur mac : Eclipse -> préférences

sur windows : window -> préférences

Tapez “Encoding” dans la barre de recherche puis aller sur CSS Files.

Assurez vous que l’encoding soit bien en ISO/UTF8 par défaut.

Idem pour HTML Files et JSP Files.

Un serveur d'application

Rdv sur le site Apache Tomcat : <https://tomcat.apache.org/>

Sur le côté, dans download, prendre la dernière version stable (pas alpha ni beta) de Tomcat.

Sur Windows : dans la section Core, prenez la version ZIP 32 ou 64 bits selon votre OS.

Sur Mac : prenez le tar.gz

- Core:

- [zip \(pgp, sha512\)](#)
 - [tar.gz \(pgp, sha512\)](#)
 - [32-bit Windows zip \(pgp, sha512\)](#)
 - [64-bit Windows zip \(pgp, sha512\)](#)
 - [32-bit/64-bit Windows Service Installer \(pgp, sha512\)](#)
- 
- A red arrow originates from the right side of the slide and points diagonally down and to the left, specifically towards the '32-bit Windows zip (pgp, sha512)' link in the list.

Un serveur d'application

Sur MAC uniquement, lancez les commandes suivantes :

```
sudo mkdir -p /usr/local  
sudo mv ~/Downloads/apache-tomcat-X.X.X /usr/local  
sudo rm -f /Library/Tomcat  
sudo ln -s /usr/local/apache-tomcat-X.X.X/ /Library/Tomcat  
sudo chown -R VOTREIDENTIFIANT /Library/Tomcat  
sudo chmod +x /Library/Tomcat/bin/*.sh
```

Sur Windows : dézippez le .zip et placez le dans un endroit sûr sur votre ordinateur.



Notre première application

Notre première application

Nous commencerons doucement avec une petite page web.

Dans Eclipse : File -> new -> project -> Dynamic web project

En Project Name, mettons : cours_je.

- Cliquez ensuite sur New Server Runtime Environment
- Déroulez la liste “Apache” pour choisir Tomcat. Prenez la même version que celle téléchargée (pas forcément la dernière de la liste).
- Cochez “Create a new local server”
- Cliquez sur “next”

Notre première application

- Cliquez ensuite sur “Browse”
- Sélectionnez le dossier Tomcat (Là où vous l’avez dézippé pour Windows; dans /Library/Tomcat pour Mac)
- Cliquez sur Finish
- Cliquez de nouveau sur Finish

Arborescence

Dans cours_jee/Java Resources se trouve un dossier src.
C'est ici que nous placerons toutes nos classes Java.

Dans src/main/webapp viendront se placer nos fichier html, javascript, JSP.

Vous trouverez le dossier WEB-INF (incontournable dans JEE) dans lequel nous placerons des fichiers de configuration JEE.

Dans WEB-INF/lib viendront se placer diverses bibliothèques externes.

Première page

Créons un premier fichier HTML :

- clic droit sur webapp -> new -> HTML file
- la nommer index.html
- finish

Changez le title et ajoutez un `<p>Ceci est une page HTML</p>` à votre body.

! Les mots en français ne sont pas reconnus par Eclipse et ce “spell checking” peut être fatigant à la longue. Désactivez le :

Préférences -> “spell checking” -> décochez “Enable”

Première page

Il est temps d'exécuter notre page en lançant le serveur.

- Faites un clic droit sur votre fichier (où on voit le code) -> Run As ... -> Run on Server
- Choisissez le serveur Tomcat et cochez la case pour toujours utiliser celui-ci
- Faites Finish

Voici votre page HTML !

Créer un Servlet



Créer un Servlet

C'est parti pour le C de notre modèle MVC.

Dans Java Resources -> src : faire un clic droit -> new -> Servlet

Java package, spécifions : `fr.formation.servlets`

Class name, spécifions : `Test`

Finish

Créer un Servlet

Regarder le fichier et enlever quelque commentaires inutiles

```
package fr.formation.servlets;

import jakarta.servlet.ServletException;

public class Test extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public Test() {
        super();
        // TODO Auto-generated constructor stub
    }

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        // TODO Auto-generated method stub
        response.getWriter().append("Served at: ").append(request.getContextPath())
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        // TODO Auto-generated method stub
        doGet(request, response);
    }
}
```


Créer un Servlet

Un Servlet est une classe Java qui hérite de `HttpServlet`.

Deux méthodes sont autogénérées : `doGet()` et `doPost()`

`doGet()` : lorsque le visiteur requête une page, ce sera une requête `get()` : charger une page pour la visualiser.

`doPost()` : lorsque le visiteur envoie des données (via un formulaire par exemple), il effectue un post.

Créer un fichier de configuration

Notre servlet n'est pas encore rempli mais nous allons la configurer pour qu'elle soit prête à être utilisée.

En créant notre servlet, un fichier web.xml a été généré dans WEB-INF; allez le voir.

Lorsque vous l'ouvrez, Eclipse vous propose une vue Design par défaut. Vous pouvez également afficher le fichier source grâce à l'onglet "source".

Je vous explique les différentes parties du fichier.

Créer un fichier de configuration

Servlet :

name = le nom pour nous du servlet

servlet-class = le nom de la classe (avec son package)

Servlet mapping :

servlet-name = fais le lien avec le servlet que l'on configure

url-pattern = la route de notre servlet. Dès qu'un visiteur ira sur l'URL /Test, notre Servlet Test sera chargé, et en particulier sa méthode doGet()

→ Modifier /Test par /bonjour

Lancer un test

Retour sur le fichier Test.java.

- Cliquez sur l'icone verte en forme de play (ou utilisez le raccourci clavier F11).
- Dans la fenêtre qui s'ouvre, laissez Restart server et cochez "Remember my decision"
- La page s'ouvre dans le navigateur (bon, le HTML n'est pas folichon).

Changez l'url dans le navigateur : au lieu de /test à la fin de l'url, mettez /aurevoir..

Une 404 est générée car il n'existe aucun Servlet lié à la route /aurevoir (logique)

Notion de Realm

Notions de Realm

En installant Tomcat, un fichier de configuration de notre serveur a été créé automatiquement.

Allez le chercher pour l'ouvrir dans Eclipse. Il se trouve dans le dossier Tomcat (là où vous l'avez placé) : tomcat/conf/server.xml

On peut voir notamment que le port 8080 qui est ouvert lorsque l'on fait un "run server" n'est pas magique, il est spécifié par la balise <Connector>

Egalement, on peut voir un Realm de configuré. Il spécifie simplement que pour se connecter à la base de donnée dont l'host est "localhost", nous utiliserons les utilisateurs de la table User.DatabaseRealm.

Pour nous, rien de précis à faire, étant donné que notre database est accessible par localhost... server.xml est déjà bien configuré !

Associer une vue à un Servlet

L'objet request, l'objet response

Voyons notre méthode doGet().

On voit en paramètre un objet request et un objet response.

request : contient tous les paramètres qu'a pu envoyer le visiteur. Info sur son navigateur, données envoyées en paramètre de l'url, etc

reponse : contient tous les paramètres du serveur. Notamment la réponse HTML.



Un simple println

Faisons un affichage simple, je vous montre.

Un simple println

<!-- Pensez bien à importer PrintWriter.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
  
    response.setContentType("text/html");  
  
    response.setCharacterEncoding("UTF-8");  
  
    PrintWriter out = response.getWriter();  
  
    out.println("Bonjour !");  
  
}
```

Une page HTML

Pour que notre servlet puisse envoyer une page HTML, nous devons créer une nouvelle JSP.

WEB-INF : click droit -> new -> JSP File
Nommez la bonjour.jsp

Renseignez le title et remettre le même paragraphe dans le body que ce que nous avons fait sur index.html précédemment

Une page HTML

Retour sur le servlet, c'est `doGet()` qui va gérer le chargement et l'affichage de `bonjour.jsp`

Je vous montre.

Une page HTML

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

    this.getServletContext().getRequestDispatcher("/WEB-INF/bonjour.js
p").forward(request, response);

}
```

Ce que fait ce code : il ne fait que passer la requete et la reponse à notre jsp.
On peut lancer la page pour voir le site.

Une page HTML

Retour sur le servlet, c'est `doGet()` qui va gérer le chargement et l'affichage de `bonjour.jsp`

Je vous montre.

Présentation des JSP

Présentation des JSP

Il est possible d'écrire du code Java dans nos pages HTML, et c'est là tout l'intérêt.

Retour dans la servlet pour écrire du code Java à transmettre à notre JSP.

Présentation des JSP

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

```
+   String message = "Au revoir !";  
+   request.setAttribute("variable", message);  
  
           this.getServletContext().getRequestDispatcher("/WEB-INF/bonjour.js  
p").forward(request, response);  
}
```

Présentation des JSP

Dans bonjour.jsp

<p>

<%

```
String variable = (String) request.getAttribute("variable");
```

```
out.println(variable);
```

%>

</p>

Refresh la page pour voir.

Présentation des JSP

Dans une jsp, on peut écrire du code JAVA :

```
<p>  
  
    <%  
  
        for(int i = 0; i< 20; i++){  
  
            out.println("Une nouvelle ligne ! <br/>");  
  
        }  
  
    %>  
  
</p>
```

Refresh la page pour voir.

Présentation des JSP

Ou encore des conditions. Modifier le servlet :

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

    +   request.setAttribute("heure", "jour");

           this.getRequestDispatcher("/WEB-INF/bonjour.js
p").forward(request, response);

}
```

Présentation des JSP

Modifier la vue, à la place de ce qu'on avait avant :

<p>

<%

```
String heure = (String) request.getAttribute("heure");
```

```
if(heure.equals("jour")){
```

```
    out.println("Bonjour");
```

```
}else{
```

```
    out.println("Bonsoir");
```

```
}
```

%>

</p>

Les inclusions de JSP

Les inclusions de JSP

Imaginez un menu que vous voulez sur toutes les pages du site.

On ne va évidemment pas mettre le menu en dur dans chaque page.

Je vous montre les inclusions.

Les inclusions de JSP

Dans WEB-INF : on crée un fichier menu.jsp

On enlève tout le contenu sauf les deux premières lignes :

Short :

- alt + ctrl + fleche haut pour dupliquer une ligne
- ctrl + shift + F pour l'auto indent

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
```

```
    pageEncoding="UTF-8"%>
```

```
<nav>
```

```
    <ul>
```

```
        <li>Page 1</li>
```

```
        <li>Page 2</li>
```

```
        <li>Page 3</li>
```


Les inclusions de JSP

Dans menu.jsp, virer tout ce qu'il y avait dans le <body> et faire :

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

(ne pas oublier le @ car la directive est comme ça pour le include)

Une deuxième page

👉 A vous de jouer :

Nous voulons une seconde page : accueil, accessible à l'url /accueil.

Créez le servlet, la configuration dans web.xml (il suffira de changer Accueil par accueil) et une jsp qui inclura le menu et un paragraphe : Bienvenue sur mon site

Rajoutons des liens au menu

Des liens au menu pour naviguer :

```
<li><a href="/cours_je/">Accueil</a></li>  
<li><a href="/cours_je/bonjour">Bonjour</a></li>
```

**Faire transiter des
données entre les
pages : l'url**

Récupérer les paramètres GET

Voyons ensemble comment nous pouvons récupérer des paramètres placés dans l'URL.

Rdv sur notre navigateur à l'url : http://localhost:8080/cours_je/bonjour et ajoutons : `http://localhost:8080/cours_je/bonjour?name=VotrePrenom`

Voici le code qui permet de récupérer le name dans l'URL :

```
String name = request.getParameter("name");
```

👉 Sur la page bonjour, affichez sous le menu le texte : Bonjour <VotrePrenom>

Récupérer les paramètres GET

Le servlet :

```
String name = request.getParameter("name");  
  
request.setAttribute("name", name);
```

La vue :

<p>

Bonjour

<%

```
String name = (String) request.getAttribute("name");  
  
out.println(name);
```

%>

</p>

Expression Language

Expression Language

Dans nos JSP, nous avons mélangé du HTML et du code JAVA.
C'est une mauvaise pratique sur le long terme qui risque de complexifier nos projets.

Aujourd'hui, nous utilisons EL (Expression Language).

Exemple d'écriture (testez le) : `${ 6 * 3 }` ou encore `$ { name }`

On peut voir que `${ name }` =

```
String name = (String) request.getAttribute("name");  
out.println(name);
```

Et `${ name }` ... c'est plus pratique !

Tester un paramètre

Avec EL, nous pouvons tester un paramètre.

Enlevez le code Java et EL en dessous du menu et mettez :

<p>

```
Bonjour ${ empty name }
```

</p>

Que remarquez vous ?

👉 Si le paramètre est renseigné, empty renvoie false, sinon true

Tester un paramètre


Ainsi, nous pouvons faire une ternaire en EL :

```
Bonjour ${ empty name ? '' : name }
```

et même mieux :

```
Bonjour ${ !empty name ? name : '' }
```

Travailler sur des tableaux

 A vous de jouer : créez un tableau prenomms (il contiendra 3 prenomms) dans votre servlet Test.java, passez le à votre JSP et affichez la première case du tableau avec EL.

Solution :

1 - Dans le servlet :

```
String[] prenomms = {"Alan", "Mathieu", "Bob"};
```

```
request.setAttribute("prenomms", prenomms);
```

2 - Dans la jsp :

```
Moi c'est ${ prenomms[0] }
```

Manipuler les Java Beans

Manipuler les Java Beans



Dans le modèle MVC (Modèle, Vue, Controller) ...
La Vue, ce sont nos ... ?

JSP ! 🥰

Le Controller, c'est les ... ?

Servlet ! 😎

Et le modèle, c'est ... ?

Pas vraiment les Java Beans, le modèle, on le verra à la fin :-)
Toutefois le modèle utilisera des Java Beans.

Manipuler les Java Beans

Les Beans, ce sont des classes Java publiques dont les attributs sont privés et qui possèdent des méthodes publiques.

On parle aussi de POJO : Plain Old Java Object.

Autrement dit, une classe JAVA qui ne dépend pas d'une interface ni d'une classe abstraite.

Pour vous présenter les Beans, nous allons créer une classe Auteur basique.

Manipuler les Java Beans

Clic droit sur src de Java Resources -> new -> Class

Package : fr.formation.beans

Name : Auteur (Toujours au singulier, commence par une majuscule)

Finish

—

Remplir la classe :

```
private String nom;  
private String prenom;  
private boolean actif;
```

Puis clic droit dans le fichier -> Generates getters and setters

Manipuler les Java Beans

Rdv dans la Servlet pour créer un Auteur à partir de notre Beans.

Supprimez tout de la méthode doGet(), excepté la ligne `this.getServletContext()`

Je vous montre comment créer un Auteur.

Manipuler les Java Beans

```
Auteur auteur = new Auteur();
```

```
auteur.setPrenom("Alan");
```

```
auteur.setNom("PironLafleur");
```

```
auteur.setActif(true);
```

```
request.setAttribute("auteur", auteur);
```

Manipuler les Java Beans

Rdv dans la JSP.

```
<p>
```

```
Bonjour ${ auteur.prenom }
```

```
</p>
```

EL appelle automatiquement getPrenom()

Manipuler les Java Beans

Ajouter :

<p>

Bonjour \${ auteur.prenom }

Vous êtes \${ auteur.actif ? "actif" : "inactif" }

</p>

Quizz 1

TP 1 : les chats, évidemment

**Qu'est-ce que la
JSTL ?**

Qu'est-ce que la JSTL ?

La JSTL (Java Standard Tag Library) c'est un ensemble de balise qui ressemble à XML que nous pourrions insérer dans nos JSP.

Par exemple, une boucle JSTL Foreach :

```
<c:forEach var="i" begin="0" end="20" step="1">
```

```
    Une ligne !<br/>
```

```
</c:forEach>
```

(Pour l'instant, vous ne devriez voir qu'une seule ligne affichée).

Les avantages de la JSTL

Facile à lire : basé sur le principe XML, tout est intuitif

Facile à réutiliser : il sera simple de coder des bouts vues JSTL pour les réutiliser ensuite

Facile à maintenir : notamment lorsque l'on travaille dans des gros projets.

JSTL, c'est une librairie que l'on va utiliser et qui contient 5 composantes :

Core : Gère les variables, les conditions, les boucles

Format : Formate les données et fait l'internationalisation du site

XML : Manipule des données XML

Function : Traite des chaînes de caractères

SQL : Permet d'écrire des requêtes, ce qui est une mauvaise pratique MVC, donc à ne pas utiliser !

Mettre en place la JSTL

Il nous faut ajouter la bibliothèque JSTL dans un premier temps.

Je vous donne deux fichiers qui sont la JSTL 2.0 à utiliser avec un serveur TomCat.

Fichiers à mettre dans WEB-INF/lib

Enfin, maintenant que nous disposons de la librairie, nous pouvons la linker dans nos jsp, avant le doctype :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
```

Et tester : `<c:out value="Bonjour !" />`

Mettre en place la JSTL

Configurons notre JSTL pour ne pas avoir à l'inclure à chaque fois qu'on créera un fichier .jsp

- Créer un fichier dans /WEB-INF/partials/taglibs.jsp
- Mettre dans notre taglibs.jsp le code qui est au début d'accueil.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

- Configurer le web.xml :

```
<jsp-config>

    <jsp-property-group>

        <description>Toutes les pages</description>

        <url-pattern>*.jsp</url-pattern>

        <include-prelude>/WEB-INF/partials/taglibs.jsp</include-prelude>
```

Mettre en place la JSTL

Info : dans web.xml, tout fichier qui se trouve dans `<include-prelude></include-prelude>`
Sera chargé en début de page.

JSTL et variables

`<c:out value="Bonjour !" />` : vous connaissez.

Elle a une particularité : elle échappe les caractères.

Pratique si vous affichez un texte qui a au préalable été saisi par un utilisateur : si celui-ci, malveillant, rentre du code (html, js, etc), il ne sera pas exécuté.

Autrement dit, c:out protège contre la célèbre faille XSS.

Dans la pratique, on écrira jamais en dur dans c:out (autant utiliser du HTML de base).

Nous écrirons essentiellement du Expression Language :

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

JSTL et variables

Cout a d'autres paramètres que value.

Vous pouvez par exemple spécifier un paramètre default au cas où ce qui se trouve dans value n'existe pas.

```
<c:out value="{ $ { name } " default="Valeur par défaut" />
```

ou alors, cela peut s'écrire ainsi (c'est exactement pareil)

```
<c:out value="{ $ { name } " >Valeur par défaut</c:out>
```

Vous pourriez aussi vouloir désactiver la protection XSS dans certains cas précis (si l'utilisateur a le droit de rentrer du BBCode par exemple) :

```
<c:out value="{ $ { name } " escapeXml="false" />
```

JSTL et variables

JSTL nous permet de déclarer nos propres variables directement dans un fichier .jsp :

```
<c:set var="pseudo" value="AlanPL" scope="page" /> OU <c:set var="pseudo" scope="page">AlanPL</c:set>
```

? scope="page" signifie que la variable n'existe que dans la page.

Il existe d'autres scope :

request -> signifie que (lors d'un post), on pourrait retrouver la variable dans la request

session -> signifie que la variable est valable pendant toute la session du visiteur

application -> signifie que la variable sera visible pour tous les utilisateurs dans toute l'application

JSTL et variables

Testons la variable :

```
<p><c:out value="Mon pseudo : ${ pseudo }" /></p>
```

Lorsque l'on travaille avec des beans, il est aussi possible de les modifier :

```
<c:set target="${ auteur }" property="prenom" value="Alan" />
```

Enfin, pour supprimer une variable de la mémoire, on fera un :

```
<c:remove var="pseudo" scope="page" />
```

Les conditions JSTL

Les conditions JSTL

Les conditions sont très, très utilisées pour créer des pages webs.

Condition pour savoir si une variable existe, si un utilisateur est connecté, si le panier n'est pas vide, si le prix de la commande est supérieur à 500€, etc etc.

Avec la JSTL, on l'écrit comme ça :

```
<c:if test="$ { 50 > 10 }">
```

```
    C'est vrai !
```

```
</c:if>
```

Les conditions JSTL

Il est possible d'enregistrer le résultat du test (true ou false) dans une variable :

```
<c:if test="{ 50 > 10 }" var="testCalcul"></c:if>  
<p><c:out value="{ testCalcul }" /></p>
```

L'intérêt étant de ne pas avoir à refaire ce test plus tard.

Par défaut, la variable a une portée définie à la page, mais on pourrait faire :

```
<c:if test="{ 50 > 10 }" var="testCalcul" scope="application">
```

Test multiples JSTL

Avec un IF, on ne peut pas faire de test multiples. Pour cela, il faut utiliser choose :

```
<c:choose>
```

```
  <c:when test="$ { auteur.prenom.equals('Bob') }">Salut Bob</c:when>
```

```
  <c:when test="$ { auteur.prenom.equals('Alan') }">Salut Alan</c:when>
```

```
</c:choose>
```

Et pour ajouter un “else” : une condition si aucun when n’est respecté :

```
<c:otherwise>Salut inconnu</c:otherwise>
```

Les boucles JSTL

Les boucles JSTL

Les boucles nous permettent de répéter une opération plusieurs fois. Elles sont utilisées essentiellement pour parcourir des tableaux : pour lister tous les clients d'un site, tous les produits, toutes les commandes, etc etc

Elles s'écrivent ainsi :

```
<c:forEach var="i" begin="0" end="10" step="1">
```

```
    <p>Un message n°<c:out value="{ i }" />!/</p>
```

```
</c:forEach>
```

! La boucle répète l'opération jusqu'au "end", inclus !

Les boucles JSTL

Grâce aux boucles, nous pouvons parcourir des tableaux. Créons en un dans notre servlet :

```
String[] titres = {"Nouvel incendie !", "Une ile a été découverte", "Chute du dollar"};  
  
request.setAttribute("titres", titres);
```

Et créons notre boucle :

```
<c:forEach items="{ titres }" var="titre">  
  
    <p><c:out value="{ titre }" /> !</p>  
  
</c:forEach>
```

! Nous pourrions mettre “begin” et “end” à notre forEach pour ne boucler qu’un nombre précis de fois.

Les boucles JSTL

Dans notre `forEach`, nous pouvons récupérer des informations sur la boucle :

```
<c:forEach items="${ titres }" var="titre" varStatus="status">
```

Et les utiliser : (exemple avec `status.count`)

```
<p>N°<c:out value="${ status.count }"></c:out> : <c:out value="${ titre }" /> !</p>
```

Existent aussi :

`status.index` : pour un compteur qui commence à 0

`status.current` : pour retrouver l'élément courant que nous sommes en train de parcourir

`status.first` : un booléen à `true` si on est dans le premier élément de la liste

`status.last` : un booléen à `true` si on est dans le dernier élément de la liste

Les boucles JSTL

Les boucles pour parcourir les objets fonctionnent exactement comme pour des String.

 A vous de jouer :

- Créez trois auteurs dans le servlet
- Mettez les auteurs dans un tableau
- Affichez :
 - 1 - Prénom Nom : est actif/inactif
 - 2 - Prénom Nom : est actif/inactif
 - etc

Les boucles JSTL

Servlet :

```
Auteur auteur3 = new Auteur();  
auteur3.setNom("Rebiere");  
auteur3.setPrenom("Vincent");  
auteur3.setActif(false);  
  
Auteur auteurs[] = {auteur, auteur2, auteur3};
```

JSP :

```
<c:forEach items="${ auteurs }" var="auteur" varStatus="status">  
    <p><c:out value="${ status.count }"></c:out> - <c:out value="${ auteur.prenom } ${  
    auteur.nom } : est ${ auteur.actif ? 'actif' : 'inactif' }" /> </p>  
</c:forEach>
```

Les boucles JSTL

Enfin, une dernière boucle dédiée aux string permet de couper une phrase selon un délimiteur :

```
<c:forTokens var="morceau" items="Un élément/Encore un autre élément/Un dernier pour la route" delims="/">
```

```
    <p>${ morceau }</p>
```

```
</c:forTokens>
```

Rappel de P00 : l'héritage

Rappel de POO : l'héritage

Parce qu'avec JEE, l'héritage de Java Beans est fondamental, faisons ensemble un rappel sur ce concept.

L'héritage, c'est le fait de regrouper les points communs d'au moins deux Java Beans dans une seule.

Nous avons déjà une bean Auteur. Imaginons une bean Editeur (nom, prenom, salaire). Créons la ensemble avec son constructeur.

Rappel de POO : l'héritage

Ces beans ont des points communs : un auteur a un nom et un prenom, tout comme un éditeur.

Ainsi, on créera un bean Contributeur qui va regrouper ces attributs communs.

Créons ensemble Contributeur et ses champs protected.

Modifions notre servlet car nous devons maintenant construire un auteur à partir de son constructeur.

Créons également un editeur.

Enfin, il est possible de les regrouper dans un tableau de contributeur.

Quizz 2

TP 2

Le radar Mitch 4.0

Travailler avec JDBC et une BDD

Qu'est-ce JBDC ?

JBDC est une bibliothèque qui va nous permettre de nous connecter à n'importe quelle base de données.

Nous allons utiliser une base Mysql pour ce cours.

Première étape : installer Mysql sur l'ordinateur.

- Téléchargez Xampp sur votre machine, il s'agit d'un logiciel qui inclut mysql et phpmyadmin pour voir les données graphiquement
- Installez Xampp en laissant la configuration par défaut

Lançons ensemble phpmyadmin et :

- créons une bdd javaee
- créons une table utilisateur(id, nom, prenom)
- insérons des données



Télécharger JBDC

Nous devons maintenant installer JBDC sur notre machine :

- Dev mysql jdbc dans Google.
- Sélectionnez "Platform independent"
- Téléchargez le .zip
- Dézippez le et copiez/coller le .jar dans WEB-INF/lib

Communiquer avec la BDD

Créons une classe Java nommée UtilisateurDB pour communiquer avec notre BDD, dans le package `fr.formation.bdd`

Puis créons une fonction pour charger la BDD. Je vous montre.

Communiquer avec la BDD

```
private Connection connexion;  
  
private void loadDatabase() {  
  
    // Chargement du driver  
  
    try {  
  
        Class.forName("com.mysql.cj.jdbc.Driver");  
  
    } catch (ClassNotFoundException e) {  
  
    }  
  
    try {  
  
        connexion = DriverManager.getConnection("jdbc:mysql://localhost:3306/javaee",  
"root", "");  
  
    } catch (SQLException e) {  
  
        e.printStackTrace();  
  
    }  
}
```



Création de notre bean

Nous aurons besoin d'une Java Bean qui va représenter nos utilisateurs.

Créez la Bean Utilisateur (id, nom et prenom), son constructeur et ses getters & setters.

Récupération des utilisateurs

Nous voulons réaliser une fonction `recupererUtilisateurs()` qui ira chercher tous les utilisateurs en BDD.

Dans quel fichier la placeriez-vous ? 😊

Avant de répondre, pensons MVC :

-> le controller, c'est le servlet

-> la vue, c'est la jsp

-> le modèle (celui qui fait l'interface entre notre app et la BDD), c'est ...

Les fichiers qui se trouvent dans le package `bdd` !

La Java Bean ne sert "que" de classe pour décrire les attributs de nos objets.

Récupération des utilisateurs

```
public List<Utilisateur> recupererUtilisateurs() {  
    // les imports se font dans le package java.sql...  
  
    List<Utilisateur> utilisateurs = new ArrayList<Utilisateur>();  
  
    Statement statement = null;  
  
    ResultSet resultat = null;  
  
    loadDatabase();  
  
    try {  
  
        statement = connexion.createStatement();  
  
        // Exécution de la requête  
  
        resultat = statement.executeQuery("SELECT * FROM utilisateur;");  
  
        // Récupération des données  
  
        while (resultat.next()) {  
  
            int id = resultat.getInt("id");
```

Et ensuite ?

Nous avons notre Bean qui décrit ce qu'est un utilisateur.

Nous avons notre table en BDD qui stock les utilisateurs.

Nous avons une classe qui nous permet de récupérer les données en base et de construire un objet Utilisateur.

Quelle est l'étape suivante ?

Modèle : ☒

Vue : X

Controller : X

Configurons notre servlet !

Le servlet

Dans doGet() :

```
UtilisateurDB utilisateurDB = new UtilisateurDB();  
request.setAttribute("utilisateurs", utilisateurDB.recupererUtilisateurs());
```

La vue

👉 A vous de jouer : terminez la vue pour avoir la liste des utilisateurs sous le form.
(le form n'ajoute pas encore des utilisateurs à la base, on est bien d'accord).

Nom :

Prénom :

Envoyer

- Alan Piron-Lafleur
- Bob Le Bricoleur

```
<ul>
```

```
    <c:forEach items="${ utilisateurs }">
```

```
    var="utilisateur">
```

```
        <li>${ utilisateur.prenom } ${  
utilisateur.nom }</li>
```

```
    </c:forEach>
```

```
</ul>
```

Ajouter des utilisateurs

Ajouter des utilisateurs en BDD

Pour cette partie je vais vous faire travailler seul.
Allons y étape par étape :-)

✍ Etape 1 :

Quand le formulaire est posté, `doPost()` crée un Utilisateur en renseignant le nom et le prénom.

! Pour l'ID, mettez "0", il sera automatiquement généré lors de l'ajout en BDD.

Etape 2 : créer une fonction `ajouterUtilisateur()` dans le modèle.
Faisons le ensemble.

```
public void ajouterUtilisateur(Utilisateur utilisateur) {  
  
    loadDatabase();  
  
    try {  
  
        PreparedStatement preparedStatement = connexion.prepareStatement("INSERT INTO  
utilisateur(nom, prenom) VALUES(?, ?);");  
  
        preparedStatement.setString(1, utilisateur.getNom());  
  
        preparedStatement.setString(2, utilisateur.getPrenom());  
  
        preparedStatement.executeUpdate();  
  
    } catch (SQLException e) {  
  
        e.printStackTrace();  
  
    }  
  
}
```

Ajouter des utilisateurs en BDD

 Etape 3 : Rendre possible l'ajout de l'utilisateur en base de données.

doPost()

```
Utilisateur utilisateur = new
Utilisateur(0,request.getParameter("nom"),request.getParameter("prenom"));

UtilisateurDB utilisateurDB = new UtilisateurDB();

utilisateurDB.ajouterUtilisateur(utilisateur);

response.sendRedirect("/cours_je");
```


Ajouter des utilisateurs en BDD

 Etape 3 : Rendre possible l'ajout de l'utilisateur créé dans le servlet.

Ajouter des utilisateurs en BDD

```
Utilisateur utilisateur = new
Utilisateur(0,request.getParameter("nom"),request.getParameter("prenom"));

UtilisateurDB utilisateurDB = new UtilisateurDB();

utilisateurDB.ajouterUtilisateur(utilisateur);

response.sendRedirect("/cours_je");
```

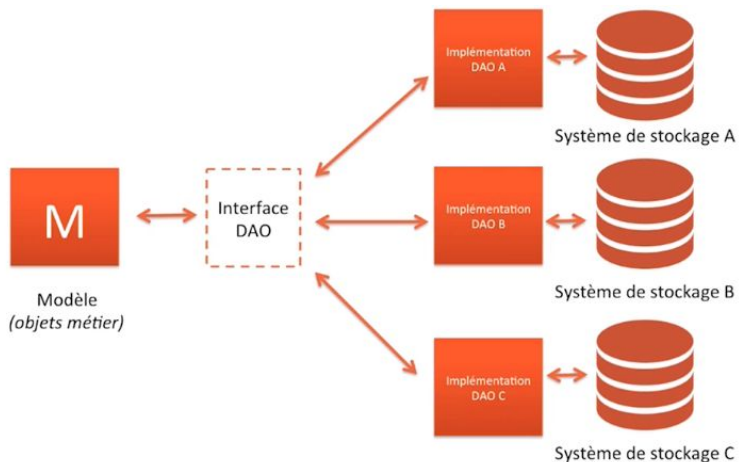
Utiliser le modèle DAO

Utiliser le modèle DAO

Le modèle DAO est une bonne pratique pour écrire ses requêtes SQL dans un code JAVA... plutôt que de taper du SQL en dur.

Le design pattern Data Object Access se place entre notre modèle et notre BDD.

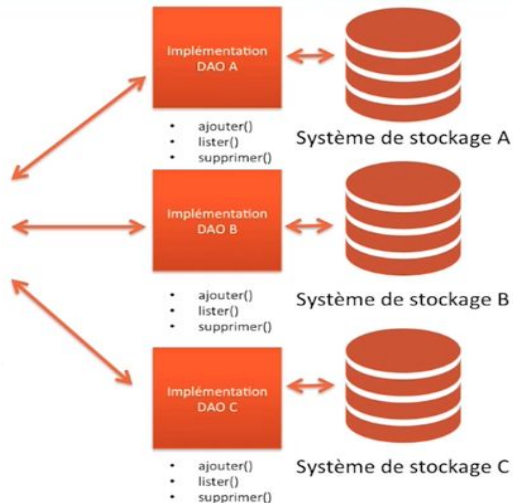
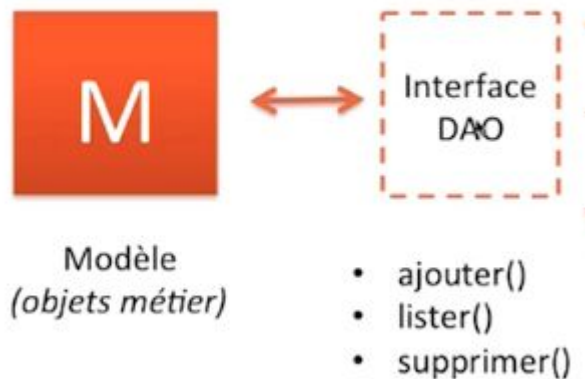
Techniquement, DAO peut gérer plusieurs système de stockage, pas que du Mysql.



Utiliser le modèle DAO

Le principe DAO, c'est que chaque méthode est déclarée dans une interface :

Chaque implémentation de l'interface va devoir, pour communiquer avec la BDD, déclarer les méthodes obligatoires.



Utiliser le modèle DAO

Avantages du système : si je change de système de stockage, je n'aurai rien à changer dans mon code.

Nous n'allons plus utiliser notre package `fr.formation.bdd` mais notre DAO à présent.

👉 Quand on parle d'un DAO, on parle de l'interface qui déclare les méthodes.



La DAO Factory

La DAO Factory permet d'initialiser le DAO en chargeant notamment les drivers nécessaires (ici un driver JDBC MySQL) et se connecte à la base de données.

Voyons cela ensemble en créant DaoFactory dans `fr.formation.dao`



La DAO Factory

```
public class DaoFactory {  
  
    private String url;  
  
    private String username;  
  
    private String password;  
  
    DaoFactory(String url, String username, String password) {  
  
        this.url = url;  
  
        this.username = username;  
  
        this.password = password;  
  
    }  
  
    public static DaoFactory getInstance() {  
  
        try {  
  
            Class.forName("com.mysql.cj.jdbc.Driver");
```


Il est temps de créer notre DAO : UtilisateurDao.
A créer dans fr.formation.dao

```
public interface UtilisateurDao {  
  
    void ajouter( Utilisateur utilisateur );  
  
    List<Utilisateur> lister();  
  
}
```

L'implémentation des méthodes

C'est le fichier UtilisateurDaoImpl qui déclare comment les méthodes de l'interface sont implémentées. A mettre dans fr.formation.dao

```
public class UtilisateurDaoImpl implements UtilisateurDao {  
  
    private DaoFactory daoFactory;  
  
    UtilisateurDaoImpl(DaoFactory daoFactory) {  
  
        this.daoFactory = daoFactory;  
  
    }  
  
}
```

L'implémentation des méthodes

Il ne reste plus qu'à implémenter les méthodes de l'interface pour rendre notre système DAO opérationnel !

Je vous montre.

L'implémentation des méthodes

Créer la méthode :

```
@Override
```

```
public List<Utilisateur> lister(){
```

```
}
```

Mettre à l'intérieur le contenu de la méthode `recupererUtilisateurs()` précédemment écrite dans `UtilisateurDB.java`

L'implémentation des méthodes

Modifier lister() :

ajouter `Connection connexion = null;` sous ResultSet

enlever `loadDatabase()` ;

Au début du try{ :

```
connexion = daoFactory.getConnection();
```

Dans le catch :

```
e.printStackTrace();
```

Enlever le finally

L'implémentation des méthodes

Et pour la méthode ajouter() :

```
@Override  
  
public void ajouter(Utilisateur utilisateur) {  
  
}
```

Copier/coller le contenu de ajouterUtilisateur().

enlever loadDatabase()

Avant le try :

```
Connection connexion = null;  
  
PreparedStatement preparedStatement = null;
```

L'implémentation des méthodes

Modifier le try :

```
try {  
  
    connexion = daoFactory.getConnection();  
  
    preparedStatement = connexion.prepareStatement("INSERT INTO utilisateur(nom, prenom)  
VALUES(?, ?);");  
  
    preparedStatement.setString(1, utilisateur.getNom());  
  
    preparedStatement.setString(2, utilisateur.getPrenom());  
  
    preparedStatement.executeUpdate();  
  
}
```

Lien DAO et DAOFactory

Pour finir, il faut dire à la DAO Factory que notre DAO existe, à mettre sous `getConnection()` :

```
// Récupération du Dao

public UtilisateurDao getUtilisateurDao() {

    return new UtilisateurDaoImpl(this);

}
```

! Si nous avons d'autres DAO, nous les listerions ici.

Point récap

- 👉 Notre Factory contient les informations de connexion.
- 👉 Notre interface liste les méthodes (ajouter, lister, ...)
- 👉 NotreDaoImpl décrit la logique métier : ce qui se passe dans les méthodes.
- ✗ Notre UtilisateurDB ... vous pouvez le supprimer (son package directement)

Occupons nous du servlet

Il ne reste plus qu'à appeler notre DAO dans le servlet, je vous montre.

Occupons nous du servlet

Dans un premier temps, déclarons un attribut, sous serialVersionUID :

```
private UtilisateurDao utilisateurDao;
```

Ensuite, nous voulons, lors de l'initialisation du servlet, pouvoir utiliser notre DAO (notre interface et ses méthodes). Cela se fait avec init() (vous pouvez enlever le constructeur).

```
public void init() throws ServletException {  
  
    DaoFactory daoFactory = DaoFactory.getInstance();  
  
    this.utilisateurDao = daoFactory.getUtilisateurDao();  
  
}
```

Occupons nous du servlet

Notre doGet() devient :

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)

    throws ServletException, IOException {

    request.setAttribute("utilisateurs", utilisateurDao.lister());

    this.getRequestDispatcher("/WEB-INF/accueil.js
p").forward(request, response);

}
```

Occupons nous du servlet

Notre doPost() devient :

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

    Utilisateur utilisateur = new
Utilisateur(0,request.getParameter("nom"),request.getParameter("prenom"));

    utilisateurDao.ajouter(utilisateur);

    response.sendRedirect("/cours_jee");

}
```

Gérer les erreurs avec son DAO

Gérer les erreurs avec son DAO

Il se pourrait que certaines erreurs empêchent votre site de bien fonctionner.

Par exemple, que se passe-t-il si la base de données ne répond plus, si le prénom inséré par l'utilisateur fait un seul caractère ou 10 000, etc ?

Commençons avec la protection de nos données en base. Dès que vous voulez être sûr qu'une donnée respecte une règle, ça se passe dans la Bean.

Prenons par exemple la règle : le nom de famille doit faire plus de 3 caractères et moins de 20.

Créons un fichier qui va gérer nos erreurs : dans le package `fr.formation.beans`, créer un fichier `BeanException.java` :

```
public class BeanException extends Exception {
```

```
    public BeanException(String message) {
```

```
        super(message);
```

```
    }
```

```
}
```


Rdv dans Utilisateur.java :

```
public void setNom(String nom) throws BeanException
{
    if (nom.length() < 3 || nom.length() > 20) {
        throw new BeanException("Le nom doit être compris entre 3 et 20 caractères.");
    }
    else {
        this.nom = nom;
    }
}
```

Gérer les erreurs avec son DAO

Pour l'instant, j'ai créé un fichier pour gérer l'exception et j'ai modifié le setter pour y apposer la règle.

Problème : pour construire un utilisateur dans Accueil.java, je passe par le constructeur (avec `new Utilisateur(...)`).

Donc, je ne passe pas par le setter (et donc pas par ma règle).

Une idée pour la solution ? 🤔

Il faut faire en sorte que le constructeur utilise le setter !

```
public Utilisateur(int id, String nom, String prenom) throws BeanException {  
  
    super();  
  
    this.id = id;  
  
    this.setNom(nom);  
  
    this.prenom = prenom;  
  
}
```

Gérer les erreurs avec son DAO

En Java, pour qu'une exception (une erreur donc), soit levée (soit activée), il faut que le code qui puisse potentiellement contenir l'erreur soit entouré de try ... catch.

Où mettriez vous votre try ... catch ?.. 🖋️ Ne me répondez pas et essayez de le réussir 😁

Il faudra le mettre là où l'on construit l'utilisateur.

Dans doPost() :

```
try {  
  
    Utilisateur utilisateur = new  
Utilisateur(0,request.getParameter("nom"),request.getParameter("prenom"));  
  
    utilisateurDao.ajouter(utilisateur);  
  
    response.sendRedirect("/cours_jee");  
  
} catch (BeanException e) {  
  
    request.setAttribute("erreur", e.getMessage());  
  
        this.getServletContext().getRequestDispatcher("/WEB-INF  
F/accueil.jsp").forward(request, response);  
  
}  
  
}
```

Et aussi dans lister du UtilisateurDaoImpl, à ajouter avec le catch SQLException :

Gérer les erreurs avec son DAO

Il ne reste plus qu'à afficher l'erreur au visiteur, directement sur le site.
Le servlet passe déjà un attribut erreur à la JSP.

Ajoutons donc le code suivant :

```
<c:if test="${ !empty erreur }">
```

```
    <p style="color:red;">${ erreur }</p>
```

```
</c:if>
```

Quizz 4

TP 3 : le crud