**J2EE :** est l'acronyme de Java Entreprise Edition, pour le développement et l'exécution d'applications distribuées, il regroupe un ensemble d’API permettant le développement d’application d’entreprise, tel que Servlet, JDBC, RMI...

**Architecture général d’une application :**

L'architecture d'une application se découpe idéalement en trois tiers :

La partie cliente : c'est la partie qui permet le dialogue avec l'utilisateur, c’est l’Interface Homme machine IHM

La partie métier : c'est la partie qui encapsule les traitements (dans des EJB ou des JavaBeans)

La partie donnée : c'est la partie qui stocke les données

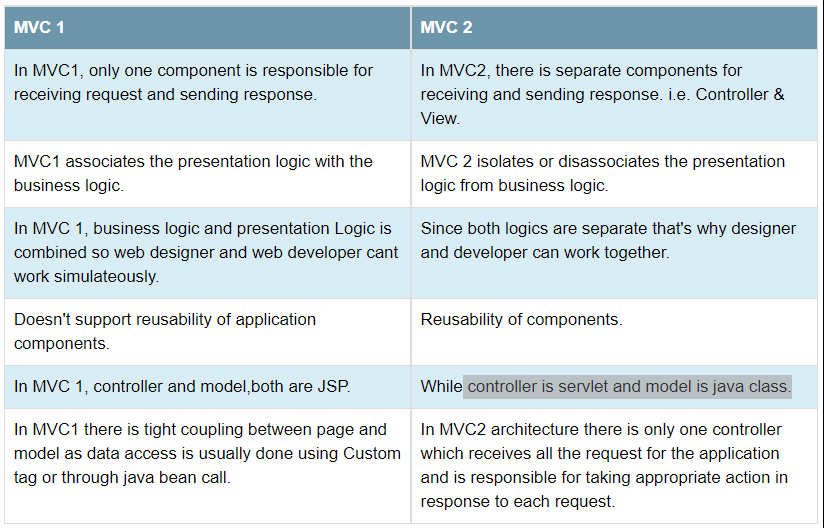
Le [MVC](https://fr.wikipedia.org/wiki/Mod%C3%A8le-vue-contr%C3%B4leur) (Modèle Vue Contrôleur) est un modèle de conception qui repose sur la volonté de séparer les données, les traitements et la présentation. Ainsi l’application se retrouve segmentée en trois composants essentiels :

Le modèle

La vue

Le contrôleur

MVC2. C’est exactement le même modèle de conception à la différence qu’il n’y a plus qu’un seul contrôleur qui se charge de rediriger la requête vers le bon traitement.



***Développement des application web en java :***

Servlet : ce sont des classe java qui hérite de la classe HttpServlet, a comme rôle de recevoir des requêtes venant du serveur Web Tomcat, de l’ai traité et de renvoyer leur réponse au client (réponse peut être json, xml, html…), contient 5 méthodes principales á redéfinir :

init() 🡪 appeler automatiquement après l’instanciation de la servlet par Tomcat, utiliser pour initialiser les objet que la servlet utilise, (crée l’objet metier pour l’utiliser dans la servlet après).

doGet() 🡪 la méthode automatiquement appeler lorsqu’on utilise un formulaire avec la méthode get.

doPost() 🡪 la méthode automatiquement appeler lorsqu’on utilise un formulaire avec la méthode post.

Service() 🡪 appeler automatiquement quel que soit la méthode d’envoi.

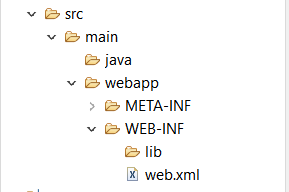
destroy() 🡪 appeler avant la destruction de la servlet .

Fichier web.xml (le descripteur de déploiement) : se trouve au serveur web, permet de mapper une servlet par un URL, depuis servlet 3.0 on peut utiliser des annotations plutôt que web.xml (@WebServlet(‘abc’))

Tomcat : conteneur Web, qui support le développement avec des servlets et des JSP.

Type de projet sous eclispse : ***Dynamic Web Project.***

Les différents dossiers du projet.



Le dossier WEB-APP doit contenir les page html, et que le dossier src/main/java doit contenir les classes java (servlets).

**public** **class** AddServlet **extends** HttpServlet{} 🡪 Création de la servlet.

Pour récupérer les données de la requête dans une servlet, il faut utiliser la méthode service :

Lorsqu’on redéfinie que ça soit la méthode doGet, doPost, ou service Tomcat nous passe 2 objets comme paramètres, HttpServletRequest et HttpServletResponse.

HttpServletRequest req: Contient toutes les informations sur la requête envoyée par le client, les variables d’entête, les cookies, URL, méthode ….

HttpServletResponse resp : c’est la réponse qu’on utilise pour répondre au client, on se serve d’elle lors d’une redirection ou bien d’enregistrer des cookies sur la machine cliente.

***req.getParameter("x");*** 🡪 permet de récupérer la variable ‘’x’’ qui se trouve soit dans le corps soit dans l’entête de la requête http.

Quand vous tapez sur submit dans un formulaire, Tomcat recherche dans web.xml sur la servlet qui traite l’action spécifiée au formulaire ***<form action = "add" >*** (l’action ici c’est add), s’elle la trouve, il va la solliciter, sinon il s’agit donc d’une action non autorisée, ou bien d’une action n’est pas encore mappée au niveau du fichier web.xml.

**Déploiement d’une servlet á l’aide de web.xml:**

L’opération d’association d’un URL á une servlet se fait par 2 balises XML, ***servlet*** et ***servlet-mapping***, les 2 continent une balise ***servlet-name*** dont le nom doit être le même (raison d’association), alors que la 1ere contient une balise ***servlet-class*** permet de spécifier la classe de la servlet, et que la 2eme continent la balise ***url-pattern*** permet de spécifier l’URL d’accéder á cette servlet.

<servlet>

<servlet-name>add</servlet-name>

<servlet-class>com.telesku.AddServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>add</servlet-name>

<url-pattern>/add</url-pattern>

</servlet-mapping>

maintenant l’url <http://www.notreSite.com/add> va être traiter par la servlet ***AddServlet.***

Pour écrire directement de contenu dans la réponse HTTP du client á l’intérieur d’une servlet il faut utiliser l’objet ***PrintWriter p = res.getWritter(),*** et écrire dedans en utilisant ***p.println(‘’ helloWorld’’***),

***RequestDispatcher :*** passer localement la requête d’une servlet á une autre (aussi passer la requête vers la vue JSP).

req.getRequestDispatcher("/sq").forward(req, res) ou bien :

**getServletContext().getRequestDispatcher("/sq").forward(req, res)** 🡪 l’objet **RequestDisptcher** permet d’appeler une servlet á partir d’une autre, tout en lui passant la requête et la réponse, pour le client il ne sait pas que le traitement et faite par la 2eme servlet, l’URL reste la même, on peut même passer des variables dedans á l’aide req.setAttribute("k","5") , alors que pour la récupérer on utilise de (String)req.getAttribute("k");

Remarque : ***getAttribute*** n’est pas la même chose que ***getParameter*** (paramètre pour récupérer une variable á partir de la requête, mais attribut pour récupérer le model en général ).

Problème : La servlet cible appartient á un autre site Web, ce qui rend l’utilisation de ***RequestDispatcher*** inutilisable, donc pour résoudre ce problème on fait recours á une autre notion s’appelle la redirection dans laquelle le serveur dit au client j’ai pas le service que vous voulez, mais vous pouvez le consulter en lui proposant de reformuler une nouvelle requête vers un autre domaine.

Pour rediriger le client vers un autre domaine on fait inclure le nouveau URL á la réponse á l’aide de la méthode : ***res.sendRedirect("http://www.google.com");***

Supposons qu’on a fait une redirection vers une servlet interne (ce qui est possible), par conséquent les données de la première requête sont plus valable (á cause de la redirection car elle génère une nouvelle requête indépendante de la première, qui possède ses propre variables) existe-t-elle une moyenne de garder les paramètres d’une requête durant toute la session de l’utilisateur ?

**Les sessions et cookies.**

Pour utiliser les session il faut instancier l’objet session á partir de la requête : ***HttpSession s = req.getSession()***;

***s.setAttribute("k", 3)*** 🡪 permet de crée une variable de session.

***int k = (int)sess.getAttribute("k")****;* 🡪 récupérer l’objet de la session.

**sess.removeAttribute("k")** 🡪 pour supprimer la variable k de la session.

Pour les cookies se sont des informations stockées par le serveur sur le disque dur de client, le navigateur lorsqu’il sollicite un site web il vérifie d’abord s’il possède des cookies, si oui, il les inclut dans la requête, du coup pour sauvegarder les cookies on doit les mettre dans la réponse et de l’envoyer au client, alors que pour les avoir dans le serveur il faut les récupérer depuis la requête du client.

***Cookie cookie = new Cookie("k", "5"); 🡪*** Crée une cookies.

***cookie.setMaxAge(60)*** 🡪 le cookie sera détruit après une minute.

***res.addCookie(cookie)***; 🡪 passer le cookie dans la réponse, pour que le navigateur client l’enregistre.

On peut pas récupérer une cookie particulier, mais on peut récupérer tous les cookies á la fois, On note que la méthode getName() contient le nom de cookie, et que la méthode getValue() contient sa valeur.

Cookie [] cookies = req.getCookies(); String k = "";

**for**(Cookie cook: cookies) {

**if**(cook.getName() == "k") { //

k = cook.getValue();

}

}

**Utiliser les annotations pour mapper les servlets :**

Pour mapper une servlet il faut annoter la classe servlet avec : @WebServlet("/sq")

**public** **class** SqServlet **extends** HttpServlet{ }

@WebServlet("/add")

**public** **class** AddServlet **extends** HttpServlet{}

Comme ça l’url /add va invoquer la servlet AddServlet, et /sq va invoquer la servlet SqServlet.

JSP : c’est l’acronyme de **Java Server Page**, ce sont des fichiers de type. jsp, permet d’écrire du code java au sein du code HTML, et pour y faire on se servi des balise très spéciales, les balise <% %>

***<% /\*du code JAVA ici\*/ %>*** ; code de cette balise signifie quel sera exécuté au côté server, et que son résultat va être inclut comme code html statique dans la réponse de le client.

<form action = *"add.jsp"* > 🡪 appeler la page jsp directement á partir du formulaire.

Les pages JSP se convertie en servlet pour être traiter par Tomcat.

Du coup un fichier jsp nommé demo.jsp, Tomcat le converti en une servlet demo\_jsp qui extends du HttpServlet, le code java de la page jsp sera insérer dans la méthode service de la servlet, et que le code html sera insérer ligne par linge á l’aide de l’objet out de res.getWritter().

<%! //méthodes et attributs %> cette balise permet de crée des méthodes et attributs dans une page jsp.

<%@page import= *‘’java.util.Date* [,d’autre package]’’ %> importer des classes ou des package au sein d’une page JSP.

<%= variableName %> 🡪 remplace <% out.println("variableName");%>

<%@include file=*"header.jsp"*%> 🡪 inclure une page jsp dans á l’emplacement où elle est indiquée

En vérité il y’a 3 balise de directives: @page, @include, @tag lib.

***Les directives de @page :***

***<%@page attribut1 = ‘’value1’’ attribut2 = ‘’value2’’ $>*** 🡪 utiliser pour différents raisons selon l’attribut utilisé.

Liste d’attributs utilisées :

-extends 🡪 étendre la jsp d’une classe donnée.

-import 🡪 permet d’importer un package donnée.

-session 🡪 (true|false), pour utiliser les sessions.

-language 🡪 Language de script á utiliser, par défaut il est en java.

-errorPage = ‘error.jsp’ 🡪 en cas d’exception la page error.jsp sera automatiquement appelée.

-contentType 🡪 permet de spécifier le contenu des pages, par défaut c’est text/html.

-isErrorPage = ‘true’🡪 pour indiquer que c’est la page qui traite les exceptions, comme ça vous aurez le droit d’utiliser l’objet exception pour afficher les messages d’ereur.

d’autre attributs :, isELignored, isThreadSafe, info.

<%@taglib %> 🡪 pour inclure des balises des autres bibliothèques, par exemple inclure JSTL <%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %> les balise JSTL il faut les préfixer par <jstl:nomDeBalise> comme ça on arrive á les différencier avec les balise html.

***Les objets instanciés par Tomcat peuvent être utiliser au sein des pages jsp :***

request(HttpServletRequest), response (HttpServletResponse), pageContext (PageContext), out (JspWritter) ~ PrintWritter, session (HttpSession), application (ServletContext), config (ServletConfig).

Remarque : C’est objets son utiliser exclusivement au sein des page jsp, par contre au servlets il faut les instancier.

Les variables de configuration ***ServletConfig*** propre á une servlet donnée, par contre les variables de ***ServletContext*** sont accessible dans toutes les servlets de l’application.

En utilisant **pageContext.setAttribute(‘’k’’,15)** on crée une variable qui est accessible que dans la page ou il était défini.

JDBC: (Java Data Base Connectivity):

D’abord Il faut télécharger le connecteur jdbc.mysql et le mettre sous le dossier web-content/web-inf/lib.

1 **import** java.sql.Connection;

2 charger le driver 🡪 Class.*forName*("com.mysql.cj.jdbc.Driver");

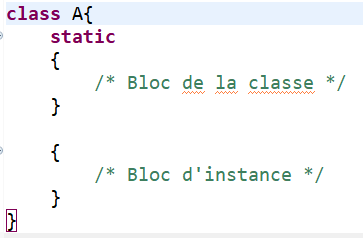
3 créer la connexion 🡪 ***connection* = DriverManager.*getConnection*(path,"root","");**

4 créer peparedStatemnt 🡪 PreparedStatement ps = connection.prepareStatement(“requete ?,?) + passer les paramètres de la requête á l’aide de ps.setString(1, ‘’toto’’) s’il s’agit d’une chaine de caractères, setInt, setDouble, setLong, s’il s’agit de d’autre type, 1 ici remplace le premier ?, pour le 2éme ? il faut utiliser setString(2, ‘’coco’’), et ainsi de suite.

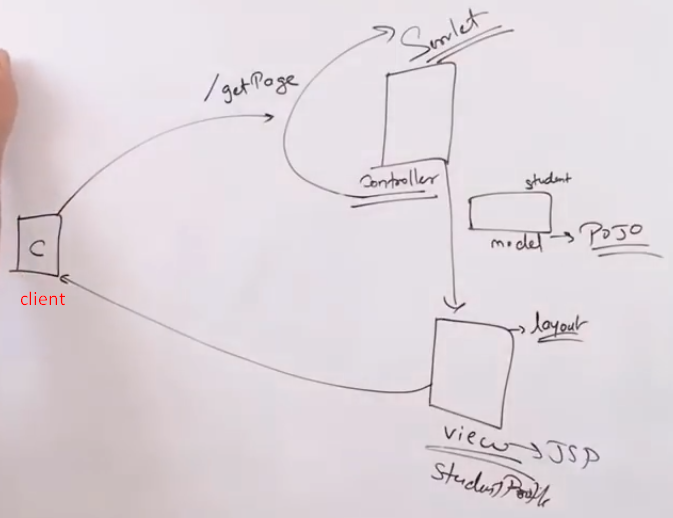
5 exécuter la requête 🡪 s’il s’agit d’un langage DML on utilise la méthode ps.executeUpdate() et en ferme la connexion et le statement, en cas d’une requête select on utilise ps.executeQuery() qui retourne le résultat de la requête dans un objet ResultSet rs qui est un objet similaire á une table sql, rs pointe au départ sur les colonnes de la table, rs.next() permet avec chaque appelle de parcourir la table ligne par ligne, pour récupérer les donnée on utilise rs.getString(‘nomDeColonne’), getDouble, ou bientgetInt selon le type de données.

Le bloc static d’une classe est exécuter une seul fois á la 1ere création d’une instance de classe, mais cette opération entraine aussi l’exécution du bloc d’instance, donc pour exécuter que le bloc de classe on utilise Class.forName(‘’)

Class.forName(‘className’) permet de charger la classe dans la mémoire, et d’exécuter le bloc static de la classe.



***MVC avec JSP et servlet :***



Contrôleur (servlet) récupère les données á partir de la couche DOA, Il les met ensuit dans le model (classe Java) qui va être passer á son tour vers la vue pour qu’il soit affiché.

JSTL : utiliser pour ne pas écrire du code Java au sein des page JSP, il facilite la communication avec le model (pas besoin de créer une variable á l’intérieur de jsp pour récupérer le model, car il est passé automatiquement) il se base sur des balises introduit par

<c:out value = *"*${k}*"* /> 🡪 pour afficher la valeur de k;

<c:forEach items = *"*${liste}*"* var = *"l"*>

<c:out value = *"${l}"* />

</c:forEach> 🡪 parcourir une liste des éléments équivalent en Java á for(Object l : liste) syso(l) ;

<c:if test=*"*${ 50 > 10 }*"* var=*"x"*>

C'est vrai !

</c:if> 🡪 si la variable x suffit la condition de l’attribut test, message C’est vrai sera afficher.

<c:set var=*"pseudo"* value=*"Mateo21"* scope=*"page"* /> 🡪permet de créer une variable pseudo avec le contenu Mateo21.

Filters : ce sont des classes java qui implémentent l’interface ***Filter*** utiliser pour intercepter les requêtes dirigées vers les servlets afin de les filtrer et de décider de les laisser passer ou pas, @WebFilter(‘’/lien’’) permet d’intercepter les requêtes passantes vers la servlet mapper par le lien indiqué.

L’exemple ci-dessous traite le cas de test si une note est valide ou pas :

@WebFilter("/add")

**public** **class** NoteFilter **implements** Filter {

**public** **void** doFilter(ServletRequest request,ServletResponse response ,FilterChain chain) **throws** IOException, ServletException{

HttpServletRequest req = (HttpServletRequest) request;

**int** note = Integer.*parseInt*(req.getParameter("note"));

**if**(note > 20 || note < 0) {

PrintWriter out = ((HttpServletResponse) response).getWriter();

out.println("note invalide");

}**else** chain.doFilter(request, response);

}

}

Lorsque le filter est appeler par l’url /add , la méthode doFilter est exécutée automatiquement, alors pour tester les variables d’une requête http, il faut d’abord faire le casting de ServletRequest vers HttServletRequest, puis de récupérer la variable et d’appliquer le test, si le test est positif alors il faut appeler le filter suivant, ici parce qu’on a pas de filter suivant la servlet mapper sur l’URL /add sera appeler á la place.

response.setContentType(‘’text/html’’) 🡪 permet de dire au client que le contenu de la réponse est une page HTML.

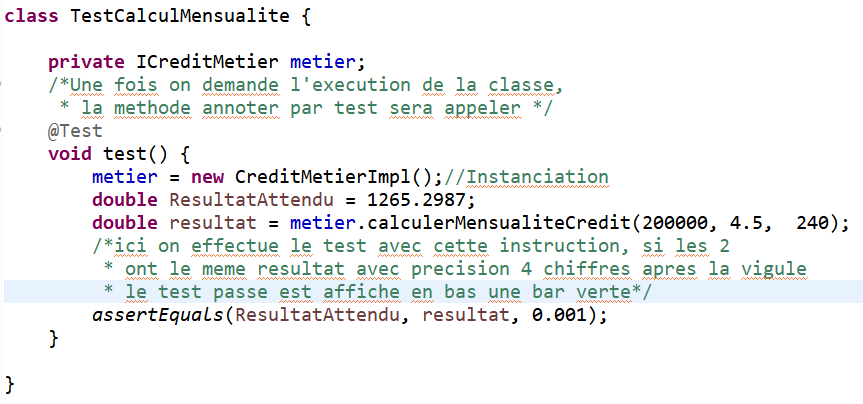
**Différence entre JSP et Servlet :** servlet c’est une classe java qu’on peut á l’intérieur introduire du code HTML, par contre JSP est une page HTML qu’on peut á l’intérieur écrire du code java, du coup pour afficher c’est plus pratique d’utiliser des JSP, mais pour les traitements c’est préférable d’utiliser les servlets. 🡪 Ce qu’on peut faire avec une servlet on peut le faire également avec une jsp (Tomcat traduit jsp en servlet, car il ne sait travailler qu’avec des servlets), mais c’est préférable d’utiliser servlet comme contrôleur et jsp comme vue.

On peut appeler directement une page JSP sans passer par une servlet tout en introduisant son nom dans l’URL, en faisant ça c’est comme si en dit que jsp c’est un contrôleur.

**Le model :** c’est une classe qui contient tous les paramètres á afficher dans une vue, (classe POJO) passer par le contrôleur après qu’on mettre les données de base de données dedans.

L’interface et écrite par le concepteur, mais son implémentation est fait par le développeur.

**JUnit :** Framework java utiliser pour effectuer des tests unitaires, généralement chaque interface ou bien classe on l’associe avec une classe JUnit, comme ça á la fin en peut tester tout le projet á l’aide d’un outil s’appelle maven, qui exécute tous les tests unitaires de projet á la fois.

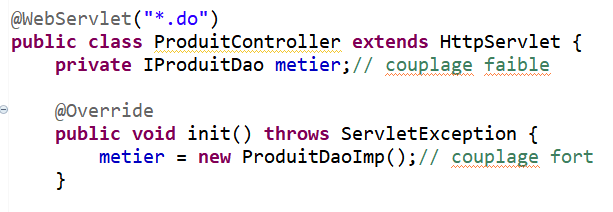


assertEquals fait la comparaison entre les 2 premiers paramètres avec précision qui est la 3éme paramètres.

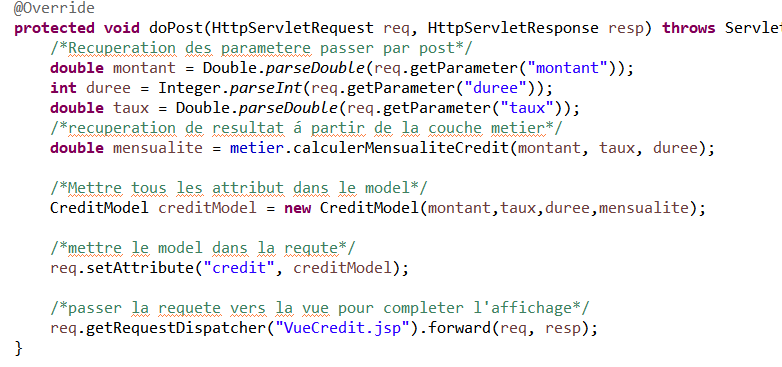
*assertTrue*(ResultatAttendu == resultat) 🡪 accepte boolean, ici si les 2 résultats sont égaux par conséquent le test est validé.

***Faire associer le contrôleur avec la couche metier ou bien couche DAO :***

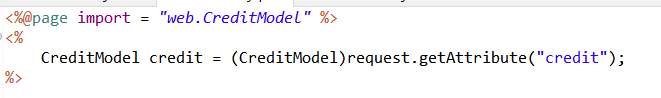
Si nous somme dans le contrôleur et qu’on veut utiliser une interface ou bien classe de couche metier (), on l’ajoute comme attribut dans la servlet (dépendance), puis on l’instancier dans la méthode init() ;



Comme ça on aura un seul objet instancier durant tout le cycle de vie de la servlet, par contre si on instancie l’objet metier dans doGet ou doPost, du coup á chaque fois qu’on a une requête avec l’une de ses méthodes, on procède alors á une instanciation de l’objet metier ce qui consomme les ressources de la machine.



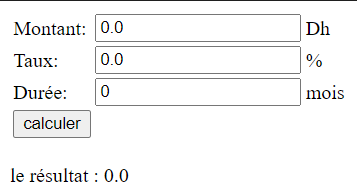
La récupération de l’objet model dans la jsp se fait par le code suivant :

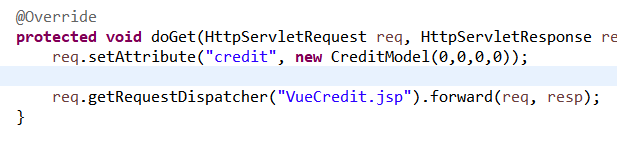


Ce code permettant la consommation de l’objet model á l’intérieur de la page jsp :

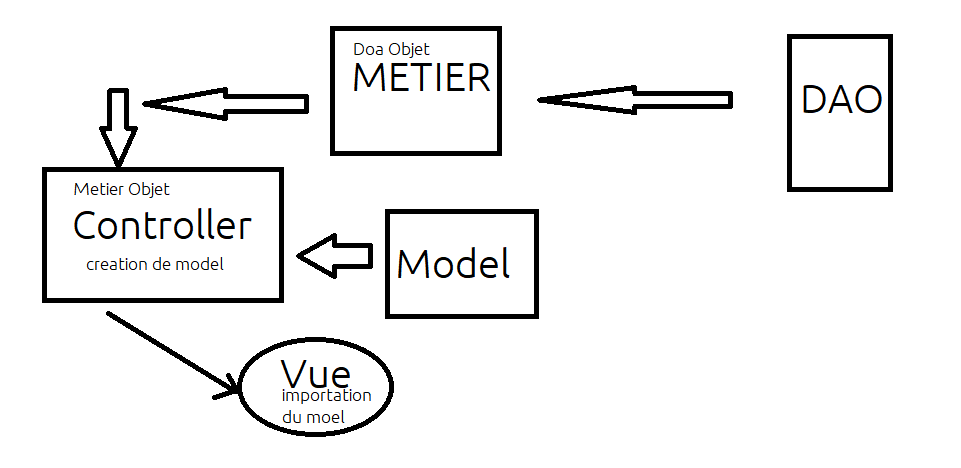


Si on part á la vue avec la méthode get on se trouvera avec l’exception nullPointer, car l’objet model est disponible seulement avec la méthode post, donc pour résoudre ce problème on crée une instance du model remplie par des valeur null (chaine vide pour String, 0 pour des nombre, null pour des liste) á l’intérieur de la méthode doGet.





***Structure Général de MVC basée sur JEE :***



Pour faire des appels á la base de donnée á partir de couche métiers on a besoin des objets (attributs) des interface DAO, et pour effectuer les traitements dans le contrôleur on a besoin aussi d’instanciation des objet metier au sein de contrôleur, toute l’application doit avoir un seul contrôleur, et que chaque page jsp doit avoir un modèle qui lui correspond.

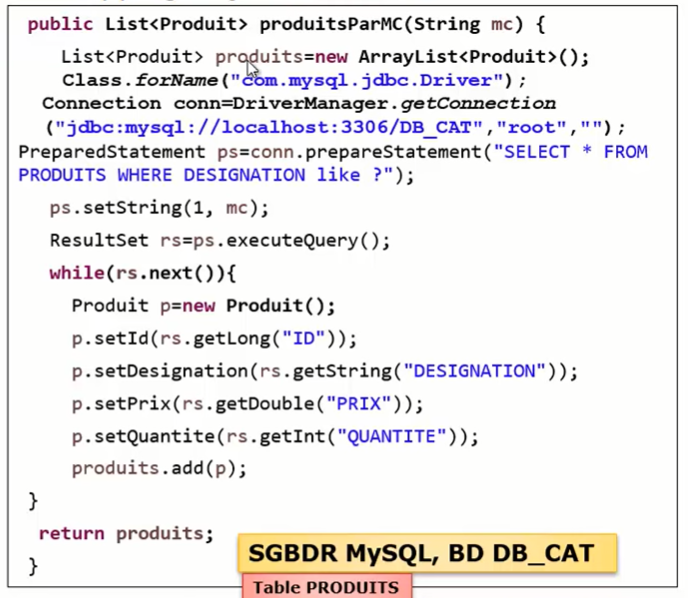
Toutes les actions á traiter dans une servlet on les met á l’intérieur de la méthode doGet, et pour séparer le traitement de chaque action d’une autre on se base sur la récupération de contexte d’action avec la méthode String path = req.getServletPath(), ensuite la valeur de variable path nous indique le traitement qu’on veut faire, il suffit donc de faire des if else, pour séparer les traitements.

Pour la méthode doPost, il suffit de passer le traitement directement en appelant la méthode doGet á l’intérieur.

Dans la méthode doGet si on veut conditionner que le traitement ou l’action se fait qu’avec la requête envoyer par post, on ajouter un test qui nous aide á savoir la méthode utilisée côté client : String method = req.getMethod();🡪 renvoi POST out GET en majuscule.

Côte DAO, il faut crée un classe SingletonConnection qui nous offre une objet connection unique utiliser pour toutes les accès á la base de donnée.

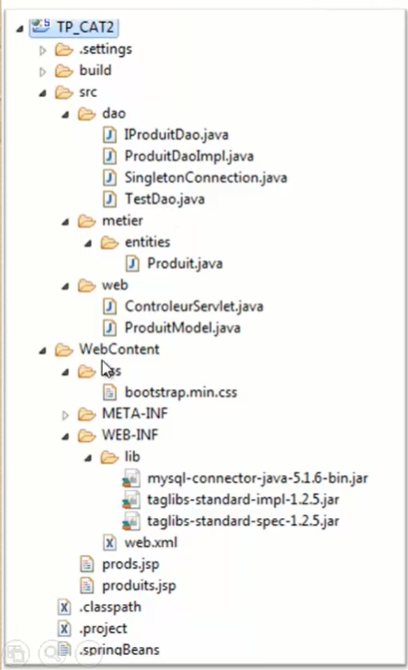
**ORM (Objet Relationnel Mapping) :**

C’est de passer du model objet vers le relationnel, et inversement 

Singleton Connexion : Tous les utilisateurs utilise la même connexion pour arriver á la base de donnée 🡪 assurer par le bloc static de la classe.

Bloc static exécuter automatiquement dès que la classe est chargée dans la mémoire.

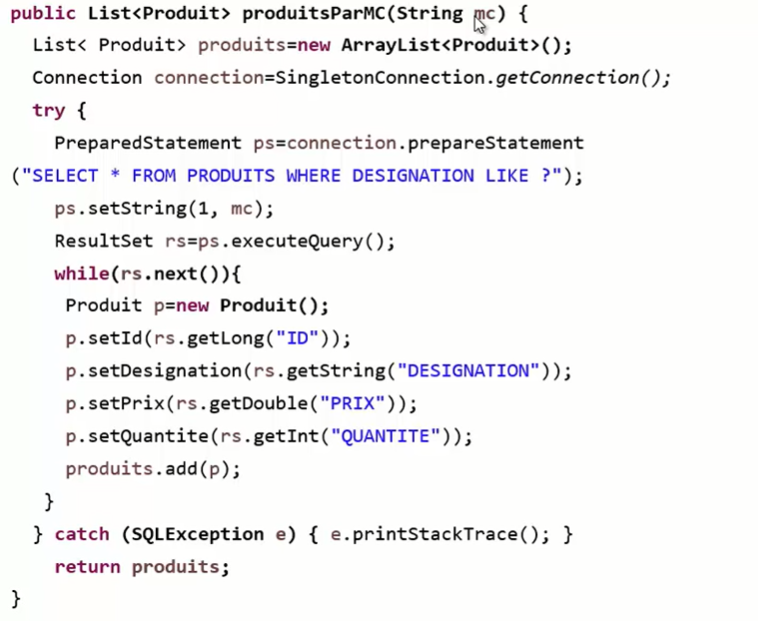




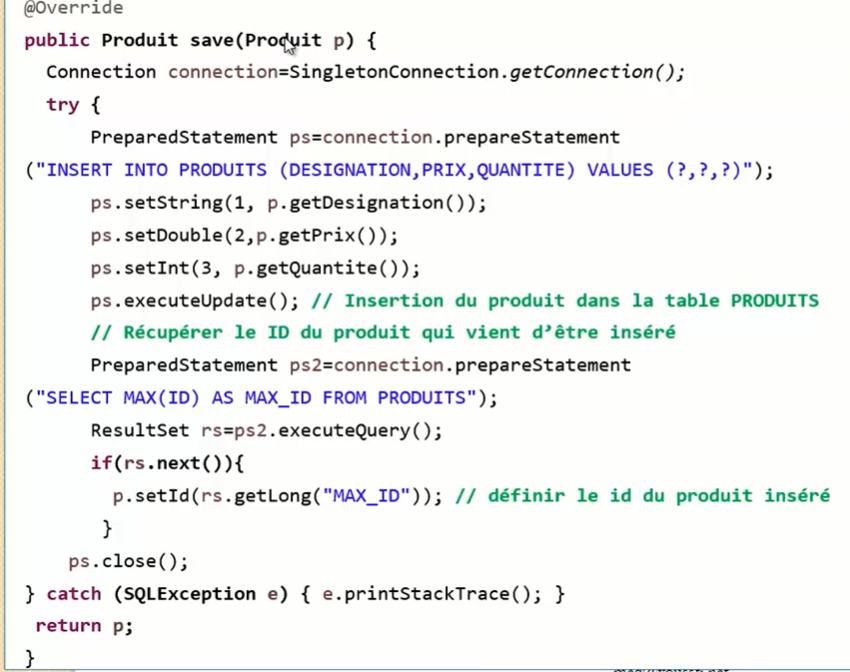
Serializable : c’est une sérialisation binaire (il y’a aussi la sérialisation XML), c’est la conversion d’un objet en un tableau d’octet pour le transférer ensuite dans le réseau ou bien de l’écrire dans un fichier OutPutStream.

Java bean 🡪 grain du café, ce sont des objet Java qui travers toutes les couches de l’application.

Relationnel vers l’objet



L’objet vers le relationnel : généralement on insert pas l’id, on suppose qu’il s’incrément automatiquement, c’est pour cela on utiliser ci-dessous 2 objets prepareStatment :

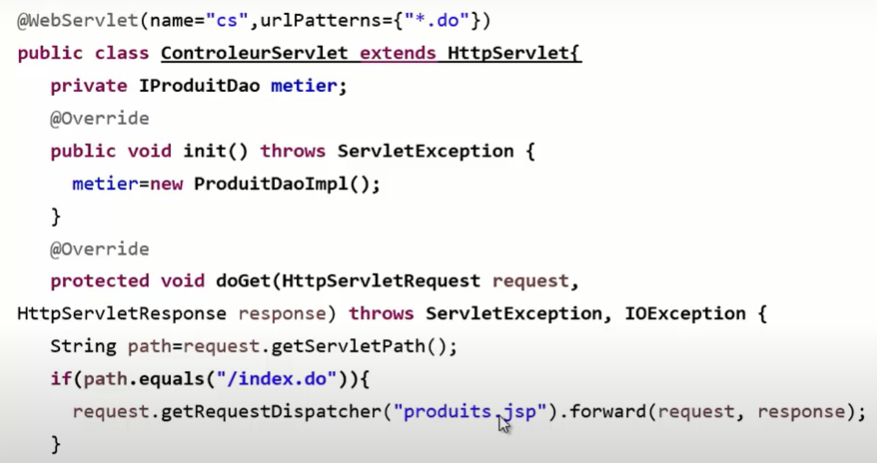


Dans la page jsp on doit afficher la liste des produits et le mot clé, par conséquent on doit créer une classe model qui contient ces 2 objets :



Mot clés doit être vide au départ, et la liste aussi.

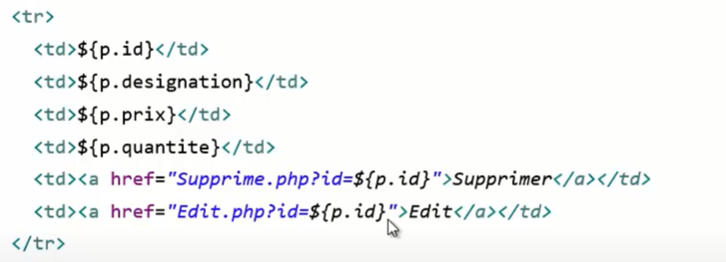
Pour utiliser qu’une seul servlet dans l’application, on doit tout d’abord utiliser une annotations générale qui redirige tous les requêtes vers la servlet @WebSerlvet(‘’\*.do’’), ensuite on récupère l’action voulue á partir de l’url de la requête par path = request.getServletPath() auquel on applique des tests imbriquer pour séparer les traitements :





Pour raison de bonne pratique, il faut mettre tous les actions dans doGet et dans doPost on fait appel á doGet, pour exiger la méthode doPost dans une action donnée on ajoute dans le test if 🡪 && request.getMethod() == ‘’post’’ ;





Jpa et Hibernate : Hibernate implémente les spécifications de Jpa, lors de développement en utilise Jpa, mais lors d’exécution c’est Hibernate qui va exécuter notre code.

persistance.xml est le fichier de configuration de Jpa et Hibernate, qui contient les informations de la base de donnée tel que (connexion :dbname, user, password).

APPLICATION :

1. EnitityManager entityManagerFactory = Persistance.createEntityManagerFactory(‘’nomDePersistanceUnit’’) ;
2. EntityManager entityManager = entitManagerFactoyr.createEntityManager()
3. entityManager.persist(Object) 🡪 enregistrer un objet dans la BD.
4. Query q = entityManager.createQuery(‘’requeteHql’’) 🡪 q.setParameter(‘’x’’, parametre ) 🡪 q.getResultList() ou bien q.getSingleResult().
5. ObjectClass o = entityManager.find(ObjectClass.class, id).
6. entityManager.merge(Object);
7. entityManager.remove(Object);