

Langages et Environnements Évolués

TP1 dockerisé : Pizzaria sous JEE*

26 septembre 2018

Il s'agit ici de faire la même application que dans le TP1, mais dans des conditions plus proche des environnement pro : nous incluons une partie des outils permettant de travailler en équipe. Ainsi, on utilisera Git, Docker, ainsi qu'une Bdd non-embarquée. Vous pouvez faire ce TP en équipes ou tout seul, nous allons essayer de fournir une correction de chaque phases accessible via les différentes versions d'un dépôt git. Si vous êtes en équipe, je vous demanderais par contre de tous être sous linux (pas forcément le même) ou tous sous windows, c'est pour une question de syntaxe des chemins d'accès ; le projet de départ (et les futures corrections) sont écrit pour un linux, il y aura probablement des changement à faire pour les autres.

Changements :

- Chaque phase est enregistré comme une nouvelle version sur un dépôt git (par défaut hébergé par le gitlab mipn ou par github),
- on exécute tous le programme via un conteneur Docker contenant le serveur (tout le monde a ainsi la même config),
- on n'utilise plus Glassfish, mais Wildfire qui est mieux Dockerisé (pas besoin de le télécharger à la main, Docker s'en chargera),
- on n'utilise plus une BDD derby-embeded, mais un postgre dockerisé (une Bdd est un microservice facile à dockeriser),
- on doit garder un oeuil sur tous les chemins d'accès et les remplacer par des variables de chemins (d'un ordinateur à l'autre, ces chemins changeront),
- Wildfly étant moins bien intégrer à intelliJ, il faut faire quelques manipulations techniques préalables, d'où le projet préinitialisé à télécharger.

Attention : La version Dockerisé de Wildfly ne supporte pas encore java 9 et 10 ; il vous faudra donc un JDK 8 pour lancer le projet...

*Jambon, Émental, Épinards, ou encore Java Edition Enerprise

Exercice 1 (Importer l'initialisation).

1. Allez dans `File` → `New` → `Project from version control` → `Git`
2. Entrez l'URL `https://gitlab.mipn.fr/flavien_breuvart/TP1_docker.git`, testez, entrez votre nom d'utilisateur et password.
3. Si le test est réussi, créez le projet.
4. dans la fenêtre `Projet`, s'il n'y a pas de fichier `src`, créez-le (`TP1_Docker` → `click-droit` → `New` → `Directory`)
5. Faites `CtrlAlt+Maj+S+`, dans `projet Sdk` sélectionnez un `Jdk-8` et cliquez sur `Apply`.
6. Dans `Modules`, sélectionnez `JPA` et fixez l'erreur en bas à droite, téléchargez la librairie requise, acceptez.
7. Faites, dans la barre de menu, `Build` → `Build Artifacts...` → `Pizzeria` → `Build`. Cela permet de créer le `.war`.
8. Vérifiez qu'il y a maintenant un `pizzeria.war` dans `docker_dir`.

On va maintenant lancer le conteneur Docker.

9. Si vous ne l'avez pas déjà fait, téléchargez les images docker de `wildfly` et `postgres` en tapant dans le terminal :

```
sudo docker pull jboss/wildfly
sudo docker pull postgres
```

10. Téléchargez le driver `PostgreSQL 42.2.5` et mettez-le dans le dossier `Docker_dir`.¹

11. Créez un réseau Docker avec la commande :

```
docker network create mon_reseau
```

12. Dans `docker_dir/Dockerfile`, à gauche de `FROM`, vous devriez voir une double flèche verte. Cliquez dessus et sélectionnez `Run on 'Docker'`, acceptez.
13. En haut à droite, vous avez maintenant un onglet `docker_dir/Dockerfile`, cliquez dessus et allez dans `Edit_configurations`.
14. Dans `image tag` et `container name`, entrez `pizzeria` (cela permet de distinguer les images et containers docker créés des autres).
15. Dans `Bind port`, cliquez sur le fichier, ajoutez un binder avec `18080` comme `Host port`, `8080` comme `container port` et `0.0.0.0` comme `Host IP`, cela redirigera le port `localhost:8080` à l'intérieur du container vers le `localhost:18080` sur votre machine (vous pouvez changer l'ip et le port hôte comme bon vous semble).
16. Dans `Commande line option`, ajoutez l'option `--net=mon_reseau`.
17. Tout en bas, dans `Before launch [...]`, il faut ajouter le build de l'artefact : `+ → build artifact → pizzeria`. Acceptez.
18. Faites de même avec `docker_dir/dockerfile_postgre`, en l'appelant `pizzadb` et branchant le port `5432` sur lui même (pas besoin d'ajouter le build de l'artefact par contre).
19. Dans `dockerfile_postgres`, à gauche de `FROM`, vous devriez voir une double flèche verte. Cliquez dessus et sélectionnez `Run 'Docker'`.
20. Faites de même avec `Dockerfile`.

Cela ouvre une nouvelle fenêtre appelée `Docker`. Vous y trouvez la liste des images construites (normalement `jboss/wildfly:latest`, `postgres:latest`, `pizzadb:latest` et `pizzeria:latest`) et la liste des conteneurs actifs (normalement `pizzadb` et `pizzeria`) et arrêtés dans la colonne de gauche. Vous y trouvez aussi différentes sous fenêtres à droite que nous allons explorer.

21. Dans la sous fenêtre `Deploy log`, vous avez les logs liés au déploiement du conteneur, vérifiez-y que l'application a été déployée avec succès.
22. Dans la fenêtre `Attached console`, vous avez les log du serveur `wildfly` lancé.² Vérifiez qu'il n'y a pas d'erreur.
23. Allez sur `http://localhost:18080`, vous devriez y trouver la page d'accueil du serveur `wildfly`.
24. Allez sur `http://localhost:18080/pizzeria/`, vous devriez y trouver l'application lancée.
25. Modifiez `index.jsp` comme s'il s'agissait d'une page `html` pour afficher "Ouverture Imminente".
26. Relancer la création d'image/conteneur et actualisez la page web.

1. Celui-ci sera inclut dans le serveur au lancement via le `dockerfile`, le `standalone.xml` et le `module.xml` afin de permettre la connexion à une `BdD Postgre`.

2. La fenêtre log contiens la même chose plus d'autre logs s'il y a d'autres logiciels lancés.

Exercice 2 (BDD).

27. Ouvrez la fenêtre Database dans View → Tool Windows → Project.
28. Dans la fenêtre Database, cliquez sur le + et sélectionnez Datasource → PostgreSQL.
29. Appellez-la BddPizza et remplissez les champs :
 - Host : localhost (c'est l'ip host du lien créé vers le conteneur de la Bdd),
 - Port : 5432 (c'est le port host du lien créé vers le conteneur de la Bdd),
 - Database : pizzaDB (c'est le nom donné à la base de donnée créé dans dockerfile_postgre),
 - User : postgres (c'est l'utilisateur par défaut, on aurait pu en créer un autre dans dockerfile_postgre),
 - Password : par défaut il n'y en a aucun (on aurait pu en ajouter un dans dockerfile_postgre),
 - URL : se remplit automatiquement. Attention, cette URL n'en est pas vraiment une, elle utilise le JDBC.
30. En bas, si l'icône ⚠ peut s'afficher, dans ce cas cliquez sur Download missing driver files
31. Testez la connexion est continuez.
32. en allant sur pizzaDB@localhost → databases → pizzaDB → schemas → public vous trouverez la table pizza déjà créée; en effet, on l'a ajouter à la création de la base de donnée grace au fichier psql_dum.sql.
33. Ajoutez une table StockPizza avec trois champs :
 - un int appelé ID, qui est la clef principal et qui est incrémentée automatiquement,
 - un varchar(20) appelé pizza,
 - un int appelé quantité;pour lier la colonne pizza à la table pizza, allez dans Foreign Keys → + entrez pizza comme Target table, puis ajouter le lien en cliquant sur le + de gauche et en liant pizza à nom. Copiez le sql généré avant d'accepter.
34. Pour pouvoir détruire et reconstruire l'image docker sans perdre la structure de table, ajouter ces changements à docker_dir/psql_dum.sql.
35. Entrez quelques valeurs : double-cliquez sur la table Pizza, cela ouvre une fenêtre avec le contenu de la table, cliquez sur le +, cela crée une ligne défaut, puis double-cliquez sur les <null> pour les remplir, une fois fini, cliquez sur la flèche vers le haut verte avec un petit "DB" au dessus, cela envera la requette correspondante à la BDD. Faites de même avec la table stockpizza.

Exercice 3 (Entités).

36. Ouvrez la fenêtre de persistance `View` → `Tool Windows` → `Persistence`.
37. Normalement il y a déjà une unité de persistance vide. Si vous double-cliquez sur `peristence.xml`, vous verrez la ligne

```
<jta-data-source>java:jboss/datasources/PizzeriaDS</jta-data-source>+
```

celle-ci signifie que l'on utilise une base de donnée déjà paramétré dans le serveur (dans `docker_dir/standalone.xml`).

38. Dans la fenêtre `Persistence`, faites `Pizzeria` → `PersistenceUnit` → clique droit → `Generate Persistence Mapping` → `By Database Schema`.
39. Choisissez `BddPizzeria` comme `datasource` et entrez `entities` comme package³.
40. Sélectionnez toutes les tables et colonnes sauf la colonne `StockPizza(pizza)`.
41. Faites `STOCKPIZZA` → `PIZZA` → clique droit → `Add Relationship`.
42. Acceptez, acceptez, acceptez.
43. Vérifiez dans `peristence.xml` que des liens vers les entités ont été créés, profitez-en pour elever les deux propriétés `hibernate.connection.url` et `hibernate.connection.driver_class`.⁴
44. Cherchez `PizzaEntity` et `StockPizzaEntity` dans la fenêtre `Project` (en haut à droite) et analysez les.
45. Faites `Pizzeria` → `PersistenceUnit` → clique droit → `Diagram ER` dans la fenêtre de persistance, manipulez le diagramme obtenu.
46. Ajoutez des constructeurs aux entités. Attention : il faut toujours qu'il y ai un constructeur vide.⁵

3. En fait, vous pouvez entrer n'importe quoi ici, c'est le nom de dossier où vont être mise les entités

4. Cest précisions sont nécessaire pour glassfish, mais firefly fonctionne avec le `jta-data-source`...

5. Car le module de persistance l'importe depuis la BD.

Exercice 4 (JSP 1).

Attention, il s'agit ici d'un premier exercice, où l'on met beaucoup de code dans le JSP et où l'on utilise des balises qui sont *deprecated*; nous allons voir par la suite que ce n'est pas une bonne habitude et nous allons créer d'autres classes pour éviter ça.

Pour l'instant, nous voyons 4 types de balise pour insérer du code :⁶

- `<%@ page import= ... %>` permet de faire des import de bibliothèques,
- `<#! ... %>` permet d'initialiser une nouvelle variable,
- `<%= ... %>` permet d'injecter une valeur,
- `<% ... %>` tout autre code java (en particulier les boucles).

Il est possible, en JSP, d'introduire du code java :

47. Importez des librairies java dans votre jsp en écrivant au début de fichier :

```
<%@ page import= "  
    javax.persistence.EntityManager,  
    javax.persistence.EntityManagerFactory,  
    javax.persistence.Persistence  
" %>
```

48. Créez les initialisations du module de persistance⁷

```
<#! EntityManagerFactory emf =  
    Persistence.createEntityManagerFactory("persistenceUnit"); %>  
<#! EntityManager em = emf.createEntityManager(); %>
```

Nous allons principalement manipuler l'EntityManager em dont la doc se trouve ici.

49. Si vous voulez afficher le prix de trois margarite, par exemple, il suffit alors d'écrire :

```
<#! PizzaEntity margarita = em.find(PizzaEntity.class, "Margarita"); %>  
<p> Le prix de trois margarite est de <%= 3 * margarita.getPrix() %> </p>
```

A ce moment, IntelliJ doit s'apercevoir que PizzaEntity n'a pas été importé. En mettant la souris dessus, IntelliJ devrait vous proposer de cliquer sur Ctrl-Enter pour résoudre ça, faites le...

50. S'il n'y a pas de Margarita, cela risque de crasher, c'est pourquoi on utilise la syntaxe absconse suivante :⁸

```
<#! PizzaEntity margarita = em.find(PizzaEntity.class, "Margarita"); %>  
<% if (margarita != null) { %>  
    <p> Le prix de trois margarite est de <%= 3 * margarita.getPrix() %> </p>  
<% } else { %>  
    <p> Cette pizzeria ne fait pas de Margarita (est-ce vraiment une pizzeria ?) </p>  
<% } %>
```

51. Si vous voulez faire la liste des pizzas, par contre, il va malheureusement falloir écrire vous-même la requête, qui doit être écrite en JPQL, un cousin du SQL qui réfère aux entités plutôt qu'aux tables :⁹

```
<#! Collection<PizzaEntity> listPizze =  
    em.createQuery("SELECT p FROM PizzaEntity p").getResultList() %>  
<% for (PizzaEntity pizza : listPizze ) { %>  
    <p> Voulez-vous une <%= pizza.getNom %> k<%= pizza.getPrix() %> euros ? </p>  
<% } %>
```

Résolvez les erreurs.

52. Il y a aussi moyen de modifier la base de données, mais il s'agit d'une opération trop complexe pour être réalisée dans un JSP en toute sécurité.

53. Écrivez un menu et compilez-le. Il y aura probablement des bugs. Essayez de les corriger avec l'aide d'un chargé de TP.

6. En fait ces constructions sont maintenant déconseillées, il est préférable (mais plus complexe) de disperser le code java ailleurs.

7. Si vous ne comprenez pas ces lignes, n'hésitez pas à demander au chargé de TP de vous les expliquer.


8. Là encore, ce n'est pas la solution recommandée, on ferait mieux d'utiliser des EJBs.

9. Encore une fois, n'hésitez pas à solliciter une explication.

Exercice 5 (EJB).

Les EJBs (*Enterprise Java Beans*) vont constituer la couche métier de votre *backend*. Il y a plusieurs types d'EJBs (*stateless*, *statefull*, *singleton* et *messages*), pour l'instant on ne voit que le premier, qui correspond à une classe dont les objets sont propres à une session mais ne comportent pas d'état (en cas de problème ils peuvent donc être écrasés et recréés sans que le client soit au courant).

Ici, on crée une simple façade pour l'entité *pizza*. Il s'agit d'un *proxy* qui évite d'avoir à gérer l'*EntityManager* dans le JSP.

54. Allez dans `File` → `Project Structure...` puis dans `Modules` → `Pizzeria` → cliquez droit → `Add` → `EJB`.
55. En bas, tant que l'icône  s'affiche, résolvez les problèmes comme proposés.
56. En haut à droite, cliquez sur `+` → `ejb-jar.xml` puis validez.
57. Ouvrez la fenêtre `EJB View` → `Tool Windows` → `EJB`.
58. Dans la fenêtre `EJB`, faites `EJB (in Pizzeria)` → cliquez droit → `New` → `Stateless Session Bean`.¹⁰
59. Pour l'instant, nous allons simplement créer un `PizzaService` qui va gérer `PizzaEntity`. Donnez, comme `<ejb-name >` et comme `EJB class` le même nom `PizzaService`.¹¹ Ajouter EJB comme package.
60. Dans la fenêtre `EJB`, Vous avez un nouvel onglet `PizzaServiceEJB` et un sous onglet `PizzaService`, double-cliquez sur ce dernier, cela vous ouvrira la classe en question.
61. Ajoutez un `EntityManager` : dans le corps de la classe, faites
cliquez droit → `Generate` → `@PersistenceContext/Unit Reference`
puis sélectionnez `PersistenceContext`, `NewPersistenceUnit` et `entityManager`. Cet `EntityManager` devrait être implicitement créé dans chaque constructeur ; s'il y a des bugs à cause de ça, créez le explicitement.
Remarque : le `@PersistenceContext` est une injection : l'entity manager n'a pas besoin d'être initialisé, il le sera par JEE de façon à ce qu'il n'y en ai qu'un seul par session (voir cours sur l'inversion de contrôle).
Troubleshooting : Il semble que Derby Embeded ne réponde pas toujours bien aux injections. Dans le cadre du TP elles ne sont pas essentielles, vous pouvez les remplacer par des constructeurs classiques si besoin (mais n'oubliez pas que dans le monde réel il faut faire ces injections).
62. Dans la classe `PizzaService`, faites "clic droit" → `Generate...` → `Delegate Methodes...` → `entityManager` → `remove` et spécifiez la méthode générée pour des `PizzaEntity`, faites de même avec les méthodes `persist` et `find`.
63. Dans le JSP, créez une instance EJB de `PizzaService` :

```
<%! @EJB PizzaService service; %>
```


Remarque : Il s'agit encore d'une injection, cela permet d'avoir toujours la bonne instance de l'EJB (comme on l'a vu dans le cours de déploiement, votre classe d'EJB est ensuite scindée en 4 classes...)
Troubleshooting : Il semble que l'injection dans un JSP ne soit pas supporté par la norme JEE (bien que souvent implémenté) ; il est en fait conseillé de préprocesser les données du JSP par une servlet¹², ce que l'on fait dans le prochain exercice.
64. Écrivez les méthodes nécessaires pour ne plus avoir à faire appel à `PizzaEntity` ni aucun code SQL dans votre JSP.
65. Modifiez en conséquence votre JSP. N'oubliez pas de supprimer les références à l'`EntityManager`.
66. Déployez.

Remarque : Certains sont perturbés par l'existence d'"Enterprise Java Beans" (EJBs) et de "JavaBeans". Il s'agit de deux interfaces standards en Java, et toutes deux fonctionnent avec du JSP et des Servlet, mais la ressemblance s'arrête là. En fait, les "JavaBeans" sont utilisées dans une architecture MVC (modèle-vue-contrôleur) des appli web java, or JEE utilise une architecture n-tier. Cela veut dire que

- au lieu d'avoir : le triplet `JSP=vue`, `JavaBean=modèle` et `Servlet=Contrôleur` avec le client qui voit la vue, utilise le contrôleur qui lui même modifie le modèle utilisé par la vue ;
- on a plutôt : on a une chaîne de 6 tiers `client-JSP-Servlet-EJB-Entités-BDD`, avec chaque élément qui interagit avec ceux juste à côté (et d'une façon moindre avec les éléments à une distance de 2). En fait, dans une application plus complexe, on va pouvoir rajouter des tiers importants : pour la sécurité, la journalisation, les accès client lourd, etc...

10. C'est l'EJB le plus courant, nous verrons les différents beans et leur utilisation par la suite.

11. Le premier est le nom JNDI qui servira par la suite et le second est le nom de la classe Java.

12. Je ne partage pas forcément cette politique...

Exercice 6 (Servlet et JSP 2¹³).

67. Ouvrez la fenêtre Web View → Tool Windows → Web.
68. Créez y une servlet : Web (in Pizzeria) → clique droit → New → Servlet, que vous nommez PreprocessMenu dans le package `servlets` et qui servira à préprocesser les requêtes utilisées dans le JSP du menu.
69. Si ça n'a pas été fait automatiquement,¹⁴ modifiez le fichier `web/WEB-INF/web.xml` comme suit pour associer la servlet à une URL :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-3.1"
  version="3.1">
  <servlet>
    <servlet-name>PreprocessMenu</servlet-name>
    <servlet-class>servlets.PreprocessMenu</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>PreprocessMenu</servlet-name>
    <url-pattern>/menu</url-pattern>
  </servlet-mapping>
</web-app>
```

Remarquez que j'ai changé l'URL en `/menu`. Lorsque l'on ira sur `Pizzeria/menu`, on sera redirigé sur la servlet qui va faire le préprocessing avant de rediriger vers la page jsp du menu.

70. On n'a pas besoin de faire de différence entre les accès "Post" et "Get", comme souvent en java, on crée alors une troisième méthode qui va être appelée par les deux autres :

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

71. On veut pouvoir utiliser notre EJB, pour ça on va donc l'ajouter comme paramètre :

```
@EJB
PizzaService service;
```

72. On peut maintenant remplir notre méthode `do` afin de rendre accessible les éléments demandés par la page jsp, Une utilisation très simple est d'ajouter l'accès à l'EJB :

```
protected void do(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    request.setAttribute("service", service);
    request.getRequestDispatcher("index.jsp").forward(request, response);
}
```

La première ligne associe à la valeur JSP "service" la valeur java `service`, cela semble idiot, mais ce sera bien utile par la suite.

La seconde ligne permet de rediriger vers l'index.

73. Il s'agit alors de remplacer toutes les occurrences de `service` par `requestScope.service` .

13. Le site du 0 est plutôt bien sur le sujet.

14. Je ne sais pas pourquoi, IntelliJ ne fait pas ça tout seul, probablement un bug...

74. On peut alors redéployer notre application et retourner sur la page `\pizzeria`. Qu'obtient-on ?
75. Essayez maintenant d'aller sur `\pizzeria\menu`. Qu'obtient-on ?
76. On peut aussi récupérer certains bouts de code, comme par exemple la conditionnelle sur la margarita :

```
protected void do(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    ...
    Pizza margarita = service.find("margarita");
    if (margarita != null) {
        request.setAttribute("resultMargarita", "Le prix de trois margarite est de " + (3 * margarita.get...));
    } else {
        request.setAttribute("resultMargarita", "Cette pizzeria ne fait pas de Margarita (est-ce vraiment...");
    }
}
```

avec, dans le jsp :

```
<p><%= requestScope.resultMargarita %></p>
```

77. Cette syntaxe permet d'exposer l'objet `requestScope`, mais elle est très lourde. Il existe pour ça la macro `${...}` qui s'utilise directement dans le html :

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

Exercice 7 (JSP 2).

Il est depuis conseillé d'utiliser d'autres balises plus "propre" pour le JSP :¹⁵

- `<c:import url = ... />` qui peut récupérer et exécuter des urls elles même transportant des objets java.
- `<c:if test = ...> ... </c:if>` qui est une meilleur syntaxe pour le `if`,
- `<c:forEach ...> ... </c:forEach>` qui est une meilleur syntaxe pour les boucles.

78. On commence par ajouter la lib dans le jsp en utilisant la ligne suivante :

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

79. Les commandes commençant pas `<c:...>` ne sont pas natives et doivent être récupérés,¹⁶ pour ça, on a utilisé la ligne suivante :

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

placez la ou début de votre jsp.

80. On peut alors remplacer l'affichage de la liste de pizzas par :¹⁷

```
<c:forEach var="pizza" items="${listPizzas}" varStatus="i">
    La ${pizza.nom} est a ${pizza.prix} euros <br>
</c:forEach>
```

81. Cette extension de langage permet de faire pas mal de choses, par exemple, on a vu que le servlet appelait la page jsp à la fin du préprocessing avec un *RequestDispatcher*, mais pas comment on pouvait appeler la servlet depuis la jsp c'est faisable grâce à la commande :

```
<c:import url="PreprocessMenu" />
```

On trouvera des listes de commande un peu partout sur internet.

Remarque : La recommandation officiel est d'extraire tout (ou presque) code java du JSP et de n'utiliser que des balises avancées, EJBs et Servlets. Ma version (personnelle) est de d'accepter tout de même du code très simple (déclaration d'une variable, appel à une méthode, opération arithmétique à la volée...). Les raisons à ce dogmatisme sont multiples : mélange de langages, impossibilité de faire des tests unitaires (voir exercice 12), difficultés à logger, à récupérer des exceptions, etc...

15. Plus récemment, encore, il a même été conseillé d'utiliser le JSF, plus complexe mais plus puissant...

16. L'ensemble des commandes ainsi importées sont disponibles ici.

17. Remarquez que l'on écrit "pizza.nom" et "pizza.prix" au lieu de "pizza.getNom()" et "pizza.getPrix()", et que ça marche même si nom et prix sont privés. En fait, il ne s'agit pas ici d'un code java, mais d'un code javaEL (Expression Language) qui ne peut pas utiliser de fonctions, mais qui remplace à la volée les appels à des champs par des getters...

Exercice 8 (Complexifiez votre model¹⁸).

Nous allons rajouter quelques tables dans la BDD :

82. Ajoutez les tables `ArticleAutre` et `AutreStock`, la première contenant un nom et un prix (comme `Pizza`), et la seconde contenant un lien vers un `ArticleAutre` et une quantité (comme `Stock`).
83. Créez les entités correspondantes.
84. Modifiez le nom de `PizzaService` en `MenuService` : pour ça faites `PizzaService` → clique droit → `Refactor` → `Rename...`. Cela permet de modifier toutes les occurrences, vérifiez que votre JSP a bien été modifié.
85. Ajoutez, dans `MenuService`, les méthodes de récupérations de `ArticleAutreEntity`.
86. Ajoutez les tables `Ingredient` et `IngredientStock`, la première contenant juste un nom, et la seconde contenant un lien vers un `Ingredient` et une quantité (comme `Stock`).
87. On veut rajouter une relation *many-to-many* entre `Pizza` et `Ingredient`, listant les ingrédients nécessaires à chaque pizza. Pour ça, on crée une nouvelle table `MtM_Pizza_Ingredient` avec un identifiant généré automatiquement et deux clés étrangères vers `Pizza` et vers `Ingredient`.
88. Comme précédemment, dans la fenêtre `Persistence`, faites `Pizzeria` → `PersistenceUnit` → clique droit → `Generate Persistence Mapping` → `By Database Schema`, puis choisissez `Pizzeria` comme *datasource* et entrez `entities` comme package, puis faites `STOCKPIZZA` → `PIZZA` → clique droit → `Add Relationship`.
89. Sélectionnez `Pizzeria.Ingredient` pour la table de droite; appelez `ingredients` l'attribut de gauche et `pizzas` l'attribut de droite; sélectionnez `java.util.Collection` pour les deux types; vérifiez que `Join Table` est coché et indiquez `Pizzeria.MtM_Pizza_Ingredient`; enfin, cliquez sur `+` et indiquez `Nom`, `Pizza`, `Ingredient` et `Nom` dans les 4 cases disponibles; acceptez.
90. Cochez `Pizza.nom`, `Pizza.prix`, `Pizza.ingredients`, `Ingredient.nom` et `Ingredient.pizzas`. Validez.
91. Analysez `PizzaEntity`, `IngredientEntity` et `NewPersistenceUnit`.
92. Ajoutez un EJB `StockService` qui gère les différents stocks; en particulier il doit y avoir des setters.
93. Créez une nouvelle page JSP de gestion où l'on voit les stocks.
94. Créez un formulaire pour faire une Pizza (et modifier les stocks en conséquence). Le résultat du formulaire est exploitable dans la servlet. Il est contenu dans un objet `HttpServletRequest request`; pour y accéder, vous pouvez appeler `request.getParameter("nouvelle_pizza")` (qui est nul si ce formulaire n'a pas été soumis).
95. Déployez.¹⁹
96. Comparez le diagramme d'entités et celui de la base de donnée :
dans la fenêtre `persistence` : `Pizzeria` → `PersistenceUnit` → clique droit → `Diagram ER`,
dans la fenêtre `database` : `MaBD` clique droit → `Diagrams` → `Show visualisation`.

18. Attention, modification le 4/10/52017, `StockPizza` remplace `Stock` dans les exercices précédents.

19. Il arrive que écraser `PizzaEntity` crée des noeuds dans le JNDI, dans ce cas, et avec notre niveau d'utilisation, le plus simple est souvent de tout refaire... (ce n'est pas si long si on importe les modules au début et si on fait quelques copier/coller)

Exercice 9 (Classes abstraites).

97. Maintenant que vous avez bien compris le principe d'entités; changez tous les noms des classes entités (telle que `PizzaEntity`) vers un nom plus utilisable (comme `Pizza`). Attention, passez par une refactorisation : dans persistance `MaDB` → `NewPersistenceUnit` → `Pizza` → clique droit `Refactor` → `Rename...`
98. Dans la fenêtre de persistance, faites `MaDB` → `NewPersistenceUnit` → clique droit → `Mapped Superclass`. Entrez le nom `Stockable` et le package `entities`.
99. Dans la fenêtre de persistance, faites `MaDB` → `NewPersistenceUnit` → `Stockable` → clique droit → `New` → `Id`
100. Modifiez `Pizza`, `Ingredient` et `AutreArticle` pour concrétiser cet interface :

```
@Entity
@Table(name = "PIZZA", schema = "PIZZERIA")
@AttributeOverride(name="nom", column=@Column(name="NOM"))
public class Pizza extends Stockable {
    ...
}
```

(dans lesquels il n'y a pas d'attribut `nom`.)

101. Jouez avec la fenêtre de persistance. (Par exemple, double cliquez sur `MaDB` → `NewPersistenceUnit` → `PizzaEntity` → `nom`.)
102. Créez (dans `entities`) une classe abstraite `Stock` avec un champ `quantite`.
103. Modifiez `StockPizza`, `StockIngredient` et `StockArticle` pour concrétiser cet interface :

```
@Entity
@Table(name = "STOCKPIZZA", schema = "PIZZERIA")
public class StockPizza extends Stock {
    ...
    public Stockable getArticle () = getPizza () ;
    ...
}
```

104. Introduisez une sous-classe abstraite de `Stockable` appelée `AVendre`.
105. Factorisez votre code.
106. Créez, dans `MenuService` la méthode suivante récupéreront tous les articles à vendre :

```
public Collection<AVendre> articlesAVendre () {
    String requete = "SELECT a FROM AVendre a ";
    Collection<PizzaEntity> listPizzas = em.createQuery(requete)
        .getResultList() ;
}
```

107. modifier votre JSP.

Exercice 10 (Encore un peu de complexification).

108. Dans les tables `INGREDIENT` et `AUTREARTICLE`, ajoutez les colonnes `PRIX_DACHAT`, `FOURNISSEUR_NUM`, `FOURNISSEUR_VOIE`, `FOURNISSEUR_ZIP` et `FOURNISSEUR_VILLE`.
109. Créez l'attribut `Prix_DAchat` dans les classes correspondantes, avec le mapping associé.
110. Dans la fenêtre de persistance, faites `MaDB` → `NewPersistenceUnit` → clique droit → `Embedable`. Entrez le nom `Adresse` et le package `entities`.
111. Ajoutez depuis la fenêtre de persistance les attributs `num`, `voie`, `zip` et `ville` à la classe `Adresse`.
112. Dans persistance, faites `MaDB` → `NewPersistenceUnit` → `Ingredient` → clique droit → `New` → `Embeded`, et ajoutez `fournisseur` de type `Adresse`.
113. Faites le mapping :

```
@Embedded
@AttributeOverrides({
    @AttributeOverride(name="num" , column = @Column(name="FOURNISSEUR_NUM" ) ),
    @AttributeOverride(name="voie" , column = @Column(name="FOURNISSEUR_VOIE" ) ),
    @AttributeOverride(name="zip" , column = @Column(name="FOURNISSEUR_ZIP" ) ),
    @AttributeOverride(name="ville", column = @Column(name="FOURNISSEUR_VILLE" ) )
})
public Adresse getFournisseur() {
    return fournisseur;
}
```

114. De la même façon, ajoutez une adresse de restaurant dans les stocks (en supposant que l'on ai plusieurs restaurants à gérer).
115. Refactorisez tout ça à travers des classes abstraites.
116. Créez et agrémentez un EJB `StockService` de gestion des stocks et une page JSP associée.²⁰
117. Ajoutez une table `restaurants` pour gérer plusieurs restaurants d'une chaîne, avec leurs propres stocks.

²⁰. S'il y a des doublons dans les résultats de requêtes c'est normal.

Exercice 11 (Requêtes JPQL).

Le JPQL est un langage de requête, comme le SQL. Au lieu de s'adresser aux tables de la base de donnée, on s'adresse ici à une base de donnée virtuelle au paradigme objet. Les "tables" sont nos classes entités, et on peut utiliser les raccourcis du paradigme objet (classes abstraites, liens objets, classes embarquées, collections...).

118. Remplissez un peu votre BDD.

119. Dans `MenuService`, modifiez la méthode qui liste les pizzas pour qu'elle ne donne que celles qui sont disponibles (pour lesquelles il existe un stock non vide) à l'aide des commandes suivantes :

```
String requete = "SELECT s.pizza FROM StockPizza s WHERE s.quantite <> 0";
Collection<PizzaEntity> listPizzas = em.createQuery(requete)
    .getResultList() ;
```

Remarquez que l'on manipule les tables comme s'il s'agissait de classes.

120. Utilisez cette méthode dans le jsp du menu. Compilez, que constatez-vous ?

121. Pour éviter les doublons, il y a deux méthodes : utiliser une autre structure que `Collection`, comme `Set` qui enlève les doublons, ou bien utiliser le mot-clé `DISTINCT` :

```
String requete = "SELECT DISTINCT s.pizza FROM StockPizza s WHERE s.quantite > 0"
```

122. Pour tester des requêtes JPQL, dans la fenêtre de persistance, faites `Pizzeria` → `persistanceUnit` → cliquer droit → `Console`. Cela vous ouvre une console pour interagir avec la BDD en utilisant des requêtes JPQL.

123. Essayez de récupérer en console tous les articles (pizzas, ingrédients...) qui sont stockés quelque part.

124. Essayez de récupérer toutes les adresses de fournisseurs.

125. Dans `MenuService`, ajoutez la méthode suivante qui récupère tous les articles à vendre sous un certain prix :

```
public Collection<AVendre> articlesAVendre (int n) {
    String requete = "SELECT a FROM AVendre a WHERE a.prix < :prix";
    Collection<PizzaEntity> listPizzas = em.createQuery(requete)
        .setParameter(":prix", n)
        .getResultList() ;
}
```

126. N'afficher que les pizzas prêtes est un peu idiot (à moins de les vouloir immédiatement ?), puisque le but est de les faire. On veut en fait afficher les pizzas dont on a les ingrédients dans un restaurant donné.

Utilisez et analysez la requête suivante :

```
String requete = "SELECT p
FROM Pizza p
WHERE 0 < ALL (
    SELECT s.quantite
    FROM IngredientStock s
    WHERE s.ingredient MEMBER OF p.ingredients
    AND s.restaurant = :resto) " ;
Collection<PizzaEntity> listPizzas = em.createQuery(requete)
    .setParameter(":resto", resto)
    .getResultList() ;
```

127. Si on avait rajouté un attribut `Collection<IngredientStock> stocks` dans `Ingredient`, on aurait aussi pu écrire la requête suivante :

```
String requete = "SELECT p
FROM Pizza p
WHERE 0 < ALL (
    SELECT s.quantite
    IN(p.ingredients) i,
    IN(i.stock) s
    WHERE s.restaurant = :resto )" ;
Collection<PizzaEntity> listPizzas = em.createQuery(requete)
    .setParameter(":resto", resto)
    .getResultList() ;
```

Essayez-le.

128. On a maintenant plein de code JPQL au milieu de notre code java, au point que c'en est difficilement lisible. Afin d'améliorer la lisibilité, on peut utiliser l'annotation `@NamedQuery`. On peut la placer avant la méthode ou (mieux) avant la définition de la classe, en mettant toutes les requêtes ensemble :

```
@NamedQuery{
    name = "AVendre",
    query = "SELECT a FROM AVendre a WHERE a.prix < :prix"
}
...
@NamedQuery{
    name = "pizzasDispo"
    query = "SELECT p
            FROM Pizza p
            WHERE 0 < ALL (
                SELECT s.quantite
                FROM IngredientStock s
                WHERE s.ingredient MEMBER OF p.ingredients
                AND s.resautant = :resto)"
}

@Stateless(name = "PizzaServiceEJB")
public class PizzaService {
    @PersistenceContext(unitName = "NewPersistenceUnit")
    private EntityManager em;
    ...
    public Collection<PIZZA> listPizzas (Adresse resto) {
        return em.createQuery("pizzasDispo")
                .setParameter(":resto", resto)
                .getResultList() ;
    }
    ...
}
```

Exercice 12 (Tests unitaires).

L'application commence à être assez grosse; il serait temps d'ajouter quelques tests. Nous ne faisons pour l'instant que des tests unitaires (teste d'une méthode sur un ou plusieurs cas) à l'aide de l'API `JUnit`.

L'idée d'un test unitaire n'est pas tant de vérifier que la méthode que l'on vient d'écrire marche bien (on fera des tests plus poussés pour ca...), mais d'y associer une assertion pour vérifier que l'on ne la casse pas en faisant autre chose. En fait, les tests unitaires sont rassemblés par `JUnit` et sont compilés séparément et ensemble, l'idée est de lancer les tests régulièrement pour vérifier que l'on a rien cassé.

129. Placez `MenuService` dans la fenêtre d'édition (e.g., dans la fenêtre `EJB`, faites `EJB` → `MenuServiceEJB` → `MenuService` → cliquer droit → `Jump to Source`).
130. Allez dans `Navigate` → `Test` → `Create New Test`... Validez et remplissez le formulaire :
131. sélectionnez `JUnit4` comme librairie en ajoutant la librairie dans le module (càd cliquez sur `Fix`), cochez `setUp/@Before` ainsi que les méthodes que vous voulez tester (et en particulier le `find`), laissez le reste tel quel et acceptez.
132. Cela vous génère une classe de test, résolvez les erreurs en important la librairie `JUnit` dans le class path (`Alt-Entré` sur les mots en rouge).
133. Lancez les tests vides pour vérifier qu'ils n'y a pas de soucis (on ne test encore rien), pour ça cliquez sur la flèche ► à coté de la déclaration de classe.
134. Vous avez alors le résultat dans la fenêtre de run, simplement on ne fait tourner que les tests en local et le minimum autour pour qu'ils aient du sens, en particulier on ne fait pas appel à glassfish.
135. En haut à droite, là ou vous avez lancé le premier run dans l'exercice 1'exercice ??, le serveur Glassfish a été remplacé par `JUnit`. Pour revenir à Glassfish (et faire tourner l'application plutôt que les tests unitaires), cliquez dessus et sélectionnez `Glassfish`.
136. Resélectionnez `JUnit`, et puis `Configurations`.
137. Dans `Test Kind`, sélectionnez `All in Directory` (pour exécuter tous les tests écrits dans le projet à la fois), aucun `fork mode` et aucune répétition (à moins que vous ne vouliez générer des valeurs aléatoires auquel cas ça peut avoir un sens de faire quelques répétitions). Rentrez le dossier de votre projet dans `Directory` et acceptez.
138. Dans votre classe de test, ajoutez un paramètre `public MenuService service;`. Initialisez ce paramètre dans `setUp()` qui est une sorte de constructeur (méthode appelée avant de faire les tests).
139. Dans la méthode `MenuServiceTest.find`, écrivez le teste suivant :

```
@org.junit.jupiter.api.Test
void find() {
    Pizza margarita = service.find("margarita");
    Pizza hawaiSomon = service.find("hawaiSomon");
    assertNotNull(margarita,
        "A moins que vous ne teniez une fausse Pizzeria, le find n'arrive pas a recuperer de pizzas");
    assertNull(hawaiSomon,
        "Soit vous avez des gouts bizarres, soit le find recupere n'importe quoi...");
}
```

140. testez.
141. Faites de même avec d'autres méthodes de vos différents `EJBs`. Vous pouvez utiliser d'autres assertions comme `assertTrue` ou `assertEquals`, vous pouvez même utiliser `fail` (un `assert` qui échoue toujours) si besoin.
142. Vous pouvez aussi utiliser des `Assert(exp_bool)`, où `exp_bool` est un test booléen, dans n'importe laquelle de vos classe; elle ne sera exécutée que pendant la phase de test (à moins que vous n'exécutiez la VM avec des options bizarres...).

Exercice 13 (Journalisation).

On va maintenant ajouter des logs. S'il y a des soucis avec la BDD (corruption, hack...), ou simplement si on veut pouvoir déboguer des problèmes complexes, il est pratique d'avoir des logs. Pour ca, nous utilisons le logger JEE (autrement on aurait fait appel à une API telle que Log4J2). Il y a plein de manière de récupérer les logs : les envoyer sur le terminal, les stocker en local, les envoyer par mail ou même les stocker dans une BDD. Ici on utilisera d'abord l'envoi sur le terminal.

143. Allez dans `File` → `Project Structure...` puis dans `Libraries` → `+` → `From Maven`.

144. Recherchez `org.apache.logging.log4j:log4j-core:2.0-rc1` et attendez (ne tenez pas compte du message en rouge), installez l'une des sources obtenues.²¹

145. Dans `MenuService`, écrivez la ligne suivante :

```
private static final Logger logger = LogManager.getLogger(MenuService.class);
```

Et importez les librairies Apache de Log4j proposée. Notez le `MenuService.class`, cela dit juste que l'on indexera les logs avec le nom (et le path) de la classe.

146. Ajouter La ligne suivante dans `find` :

```
logger.error("Crash de la BDD (Blague de mauvais gout...)");
```

Lancez les tests JUnit. Observez.

147. Compilez le programme normalement ? Trouvez-vous le log ?

148. Changez l'erreur en :

```
logger.trace("start find ( {} )", pizza);
```

Que se passe-t-il ?²²

149. Créez un fichier²³ `log4j2.xml` dans `src` avec le contenu suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

150. Recompiliez le test. Que se passe-t-il ? Rien ? C'est normal, j'ai recopier la configuration par défaut...

151. Changez la ligne `<Root level="error">` en `<Root level="trace">` et relancez le test. Mieux ? Il y a six niveaux de log disponibles :

- TRACE : traces d'exécution
- DEBUG : débogage
- INFO : information
- WARN : avertissements
- ERROR : erreur
- FATAL : arrêt imprévu de l'application

152. On peut faire encore mieux : dans la classe entité `Pizza`, faites cliquer droit → `Generate` → `To String`. Relancez le test.

21. Maven est un builder (remplace MAKE), et en particulier permet d'inclure aisément des librairies; par manque de temps, nous ne l'utilisons pas dans le cours.

22. La configuration par défaut ne log que les erreurs...

23. Normalement il est maintenant possible d'utiliser un format YAML qui est plus lisible, mais je n'ai pas réussi à le faire fonctionner. Si quelqu'un a une idée...

153. Même si les traces ne sont pas activées, elles sont calculées. Lorsque la résolution est lourde cela peut être problématique. Pour ceux qui ont eu la dernière version de log4j2, on peut utiliser la commande suivante qui est une version paresseuse utilisant les λ -expressions de java :

```
logger.trace("start find ( {} )", () -> pizza);
```

154. Remarque : pour les traces d'entrée et de sortie de méthodes, il est conseillé d'utiliser ²⁴ `traceEntry(param1, ..., paramn)` et `traceExit(result)`.

155. Si vous voulez envoyer des mails en cas d'erreur normal et surtout fatal (particulièrement utile pour vous apercevoir qu'il y a un soucis), il faut changer l'appendeur SMTP :

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <SMTP name="Mail" subject="Error Log" to="errors@logging.apache.org" from="test@logging.apache.org"
      smtpHost="localhost" smtpPort="25" bufferSize="50">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
    </SMTP>
  </Appenders>
  <Loggers>
    <Root level="traces">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

156. Si on veut avoir des logs différents pour les tests et le serveur, c'est faisable : une des manières est de mettre le `Log4J2.xml` du serveur dans `src`, de mettre un `Log4J2-for-tests.xml` où on veut et dans la configuration de JUnit (Edic Configuration dans le panneau de run en haut à droite), ajoutez l'option suivante dans la VM `-Dlog4j.configurationFile=Path` où `Path` est le chemin d'accès à `Log4J2-for-tests.xml` (depuis le module il me semble). Il est aussi possible d'avoir différents appendeurs pour des niveaux d'importance différent, ou même de moduler l'importance selon le fichier traité (c'est le principe des loggers).

157. Loggez toute votre appli.

24. Selon la version, ils peuvent aussi s'appeler *entry* et *exit*.