22/01/2019

Noemie pham

[nom de la société]

Création de l’application web Games Corner

Log de création

Table des matières

[Introduction 2](#_Toc536822214)

[Description des classes 2](#_Toc536822215)

[MVC 2](#_Toc536822216)

[Modèle 2](#_Toc536822217)

[Objectif 3](#_Toc536822218)

[Journal de bord 4](#_Toc536822219)

[Démarrage du projet 4](#_Toc536822220)

[Suivi du projet 4](#_Toc536822221)

[23/01/2019 4](#_Toc536822222)

[24/01/2019 5](#_Toc536822223)

[25/01/2019 5](#_Toc536822224)

[29/01/2019 5](#_Toc536822225)

[30/01/2019 6](#_Toc536822226)

# Introduction

Le projet Games Corner s’inscrit dans une optique d’apprentissage graduel de la programmation Java JEE dans le cadre de la création d’une application web d’eCommerce.

Pour progresser et comprendre l’évolution qui s’opère actuellement dans le monde de l’informatique (à l’heure où ce document est écrit, la plupart des nouveaux projets Java utilisent Spring, AngularJS avec des webservices REST pour la transmission des données aux modèles du MVC), ce projet a été démarré dans le but de créer une application web intégralement en partant de 0.

Cette appli sera répliquée 3 fois avec différentes technologies. La partie back end (services + DAO) restera la même et se greffera en tant que module à part entière pour chaque nouveau projet.

La mise à jour du modèle de MVC à partir de la couche de service se fera à travers l’utilisation de webservices en REST qui renverront, si possible, les données sous forme de JSON en gardant les annotations de Spring pour pouvoir utiliser l’IoC.

L’intérêt est de développer la partie front end selon 3 différentes technologies (JSF, Spring MVC et AngularJS) avec les combinaisons suivantes :

* JSF couplé avec Hibernate en temps qu’ORM et Primefaces pour l’interface graphique.
* Spring MVC avec Hibernate avec un framework à déterminer pour l’interface graphique.
* Angular et Node.js pour la dernière version du projet.

# Description des classes

## MVC

### Modèle

#### Super Classes

* Utilisateur : Dans cette classe, l’attribut vendeur est un Boolean qui détermine si l’utilisateur est simplement un client (false), ou un vendeur (true) qui va proposer un jeu dans la partie « vente de particulier à particulier ».

Il possède également un attribut admin (Boolean) qui détermine si l’utilisateur est un admin sur la partie forum de l’application.

L’admin pourra bannir des membres sous différentes conditions (supprimer le compte de l’utilisateur concerné), supprimer des messages dans diverses conversations, poster des annonces spéciales ayant trait à des ventes spéciales sur le site,…

Un utilisateur peut être spécialisé en plusieurs classes filles différentes :

* + Un client : Première spécialisation, il a accès aux pages de ventes du site ainsi qu’à la vente de particulier à particulier du côté achat. Il pourra également accéder aux forums.

Toutes actions du client nécessitent une inscription et une connexion sauf dans le cas de la consultation des différents catalogues de jeu.

* + Un vendeur : Seconde spécialisation de classe, il peut se définir comme étant vendeur dans la partie « vente de particulier à particulier » et aura accès à des formulaires de mises en ventes de jeu spécialement réservées à ce type d’utilisateur.

Chaque vendeur doit payer une commission de 20% par exemple sur ses ventes et cette commission va diminuer avec le temps selon des règles commerciales prédéterminées.

* Article : Elle se spécialise en plusieurs types de classes. On retrouve entre autre une classe JeuxVideo, JeuxSociete, JeuxDeRole, Livre, Figurine, …

Certains attributs sont utilisés dans certaines parties du site spécifiques. Par exemple, l’attribut etat sert dans la partie vente de particulier à particulier. Il indique si l’article est neuf ou d’occasion.

Dans l’implémentation du rating, les clients peuvent noter l’article et laisser un commentaire.

Le rating sera calculé et la note moyenne correspondra au rating de l’article.

# Objectif

Plusieurs versions de l’application seront délivrées via GitHub. Chacune représentera une nouvelle fonctionnalité à implémenter au projet :

* D’acheter tous genres de jeux vidéos (RPG, point and click, tactique,…), de jeux de société et de jeux de rôles, de figurines, de livres spécialisés, de goodies,...
* Implémentation d’un panier par utilisateur.
* La mise en place d’un système d’inscription et de connexion.
* L’interface graphique doit être jolie avec l’implémentation de Primefaces pour rendre le projet agréable à regarder.
* Utilisation de javascript pour dynamiser l’application web. Il sera possible de s’appuyer sur des frameworks pour faciliter cette étape, le but étant de comprendre l’imbrication de JS dans une page internet pour améliorer le confort d’utilisation de l’utilisateur.
* Déposer le projet sur Git et GitHub pour pouvoir versionner l’application.
* Design d’un logo pour le site (Aide de Mathieu à envisager pour l’utilisation d’Adobe).

Cela résume dans un premier temps l’ensemble des fonctionnalités qu’il faudra implémenter dans la première version de l’application Games Corner.

Par la suite, on voudrait que le projet possède les fonctionnalités suivantes :

* Mise en place d’une search bar avec autocomplétion (affichage d’une icône image à côté de l’entrée proposée).
* Mettre en place un système d’envoi de mail et de contact.
* Séparer les clients des vendeurs à l’aide de l’attribut vendeur dans la super classe Utilisateur.
* Mettre en place un système pour la partie « vente de particulier à particulier » qui permet l’affichage de pages que seuls les vendeurs pourront consulter. Cela leur débloquera une option dans leurs paramètres de compte.
* Lors de la mise en vente d’un article de particulier à particulier, une commission sera prélevée par le site qui pourra être réglable (on peut partir de 20% de commission par vente exemple) et pourra descendre en fonction de la fidélité du vendeur.

Cela devra par exemple faire l’objet d’un moteur de règle qui va déterminer dynamiquement cette réduction de % de commission en fonction du moment où l’utilisateur s’est inscrit en tant que vendeur particulier ou autre règle à déterminer.

* Mettre en place un système de paiement paypal.
* Rajouter une partie jeux en ligne avec des petits jeux codés en C ou C++.
* Mettre en place un système de forum avec admins.
* Mettre en place un système de ratings pour les jeux ou pour la réception côtés clients.
* Intégrer une internationalisation des ressources de façon à proposer le site en anglais et français.
* Implémenter un système de ratings des articles par les clients. Le rating sera représenté par un ensemble de 5 étoiles.
* Envoyer l’application sur un serveur Linux.
* Créer un script qui supprime les factures les plus anciennes (plus de 2 ans dans un premier temps).

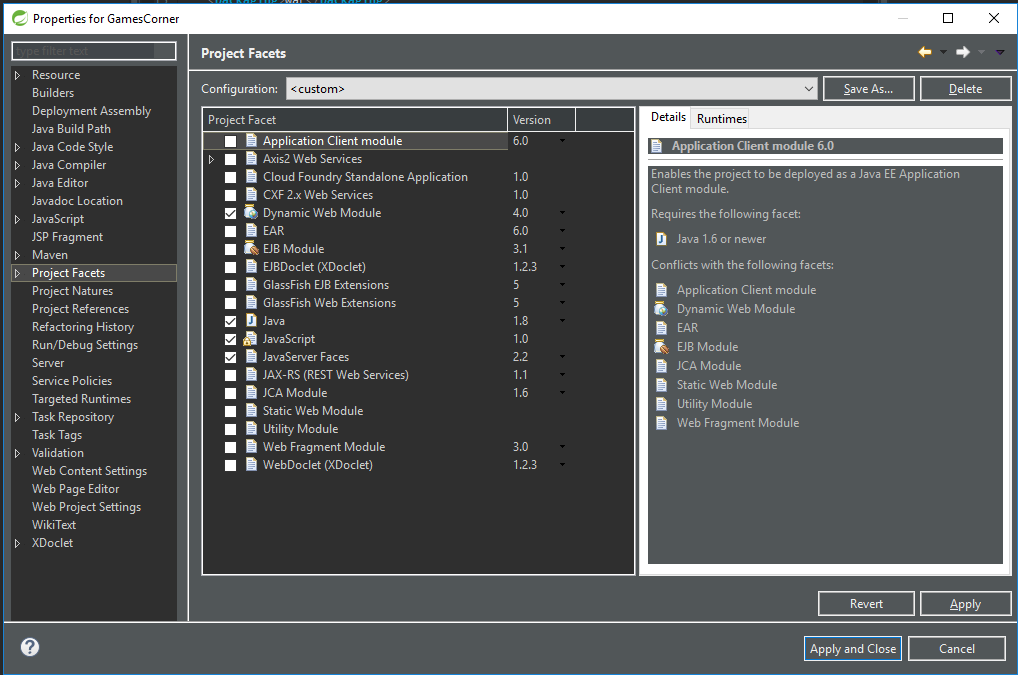
# Journal de bord

## Démarrage du projet

Le projet « Games Corner » a démarré le 21/01/2019.

Il a débuté sur STS Spring Tools sous le nom GamesCorner en tant que projet Maven simple (sans archetype lors de la création).

Dans les propriétés du projet, la partie Project Facets se présente ainsi :



Dépendances rajoutées au projet :

* Jsf-api version 2.2.2
* Jsf-impl version 2.2.2
* Javax.servlet-api version 3.0.1

Dans le fichier faces-config.xml, le namespace <http://xmlns.jcp.org/xml/ns/javaee> a été supprimé puisqu’il est originalement dupliqué dans le fichier d’origine.

## Suivi du projet

### 23/01/2019

Ajout des dépendances suivantes :

* Primefaces version 3.5 à partir du repository primefaces et non pas du central.
* Le thème bootstrap de primefaces version 1.0.9

### 24/01/2019

Ajout des dépendances suivantes :

* Commons-fileupload version 1.3 qui rajoute également la dépendance commons-io version 2.2

Liste des problèmes rencontrés et résolution :

* **« java compiler level does not match the version of the installed java project facet »:**

On a rajouté dans le fichier pom.xml la configuration du plugin maven-compiler-plugin en définissant le jdk source et target dans la configuration. Elle a été définie à 1.8 pour correspondre à la JDK indiquée dans l’IDE.

* **« Cannot change version of project facet Dynamic Web Module to 2.5 » :**

On a modifié le fichier org.eclipse.wst.common.project.facet.core.xml présent dans le dossier .settings en modifiant la version du web dynamic module par la ligne suivante :

<**installed** facet=*"jst.web"* version=*"4.0"*/>

Il faut également vérifier que la version du dynamic module dans le fichier web.xml soit bien à 4.0

* **Problème d’update de Maven « org.eclipse.wst.validation has been removed » :**

Lié au problème précédent.

On a supprimé le dossier .settings contenant le fichier concerné.

Lorsque ce problème est rencontré, cela signifie que le fichier org.eclipse.wst.validation.prefs a été supprimé et n’a pas été réecrit lors d’un update Maven.

Pour fixer ce problème, il faut supprimer le repository de maven contenant toutes les dépendances téléchargées. On clean ensuite les projets puis on fait un maven update.

Si on vérifie dans le dossier .settings, on voit que le fichier disparu a été recrée et que les dépendances ont été de nouveau téléchargés dans le repository local.

### 25/01/2019

Ajout des paramètres suivants dans le fichier web.xml :

* Ajout des balises <context-param> suivantes :
  + javax.faces.PROJECT\_STAGE permettant d’afficher des erreurs durant la phase de développement.
  + javax.faces.DEFAULT\_SUFFIX permettant de déterminer l’extension par défaut des fichiers jsf.
  + primefaces.THEME permettant de définir le thème à utiliser pour primefaces. Correspond au thème défini dans le fichier POM.xml.

Dans le fichier POM.xml :

* Ajout d’une description du projet.
* Ajout de propriétés sur le build du projet (version des différentes dépendances).

### 29/01/2019

Dans le fichier web.xml :

* Rajout de l’url-pattern \*.faces
* Rajout du context-param javax.faces.INTERPRET\_EMPTY\_STRING\_SUBMITTED\_VALUES\_AS\_NULL
* Ajout d’une balise <welcome-file-list> qui définit le nom du fichier d’index.

Dépendances ajoutées :

* Joda-time version 2.10
* Spring-mvc version 4.3.2.RELEASE
* Servlet-api version 3.1.0
* Postgresql-driver version 42.2.5
* Modification de JSF de 2.2.2 à 2.2.18
* Modification de la version de la version de Spring de 3.2.0.RELEASE à 4.1.9.FINAL
* Modification de la version de Primefaces de 3.5 à 6.2

Ajout de fonctionnalités :

* Ajout d’une classe Utilisateur (superclasse qui sera divisée en Client ou Admin).
* Ajout des classes Client et Admin.
* Ajout d’une classe Panier.

### 30/01/2019

Ajout de fonctionnalités :

* Ajout d’une classe Article (superclasse qui sera divisée en JeuxVideo, JeuxSociete ou JeuxDeRole).

### 01/02/2019

Ajout de fonctionnalités :

* Ajout d’une classe Commande qui résume les informations de l’article acheté ou en cours d’achat.

### 02/02/2019

La plupart des Beans ont été définis. Il est à prévoir que ces beans seront redéfinis par la suite lorsqu’on rajoutera les nouvelles fonctionnalités.

Ajout de fonctionnalités :

* Ajout du fichier d’accueil index.xhtml

Ce fichier correspond à la page que l’application va afficher si l’on ne précise aucune page xhtml de base.

* Création d’un fichier CSS nommé style.css dans un folder /resources/css.
* Ajout d’un folder /resources/img dans lequel seront stockés les images utilisés dans la présentation du site.

Liste des problèmes rencontrés et résolution :

* **« java.util.zip.ZipException: invalid LOC header (bad signature) »**

Rajout d’une balise <version> dans le build pour le plugin maven-compiler-plugin en précisant la version à 3.8.0

Suppression du repository de Maven et Update de Maven pour retélécharger les dépendances du POM.

* **Problème de liaison entre le fichier CSS et le fichier xhtml**

src

`-- main

|-- java

|-- resources

`-- webapp

|-- resources

| |-- css

| | `-- style.css

| |-- images

| | `-- logo.png

| `-- js

| `-- script.js

|-- WEB-INF

| `-- web.xml

`-- index.xhtml

Il faut que le projet respecte cette architecture pour les ressources autres que les fichiers java.

Pour les fichiers d’internationalisation, on doit en revanche utiliser cette architecture :

src

`-- main

|-- java

|-- resources

| `-- com

| `-- example

| `-- i18n

| |-- text.properties

| |-- text\_en.properties

| |-- text\_es.properties

| `-- text\_nl.properties

:

Ce problème sera réglé dans une prochaine itération.

Dépendances ajoutées :

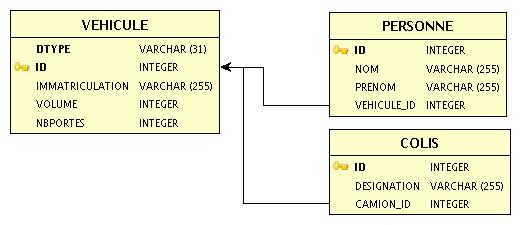
* Spring-tx version 4.3.2.RELEASE
* Hibernate-Core version 5.3.7.Final

### 03/02/2019

On s’attaque à la stratégie de persistence des données en base en travaillant sur les modèles du projet.

Pour cela, on peut utiliser 3 stratégies :

* La stratégie SINGLE\_TABLE comme on peut le voir dans l’exemple suivant :



Ou

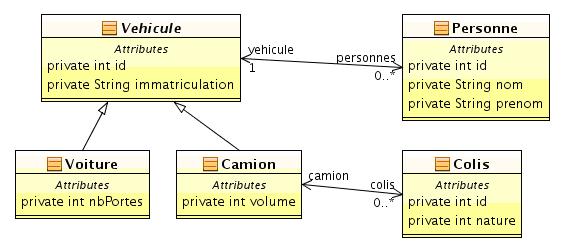


Toutes les données sont entrées dans une seule table (ici Véhicule) qui va contenir toutes les informations tant pour la classe Véhicule que Voiture ou Camion.

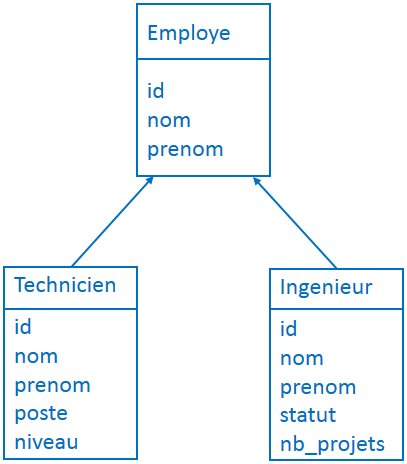
Les tables Voiture et Camion n’existe pas en base et seul une colonne discrimante présente dans la table Véhicule permet de déterminer si la ligne concerne une voiture ou un camion.

Cette stratégie créer beaucoup de colonnes vides (NULL) pour tous les champs qui ne rentrent pas dans les champs de Voiture ou Camion.

* La stratégie JOINED comme on peut le voir dans l’exemple suivant :



Ou



Dans cette stratégie, la classe mère contient les attributs communs que ses classes filles partageront. Les colonnes sont liées par une table de jointure. Cela peut entrainer des problèmes de performances pour des recherches (utilisation de table de jointure).

On n’utilisera pas la dernière stratégie (TABLE\_PER\_CLASS) qui est trop coûteuse en ressources.

Pour les annotations à utiliser, on se penchera plutôt sur les annotations du package javax.persistence au lieu du package org.hibernate.annotations.

Cela permet de pouvoir lancer le code sur d’autres projets implémentant la JPA tout cela dans un souci de compatibilité.

*@Inheritance*(strategy=*InheritanceType*.***SINGLE\_TABLE***)

Cette annotation permet de préciser qu’on utilise la stratégie de la single table. On utilisera la statégie SINGLE\_TABLE ici.

Ajout de fonctionnalités :

* Modification de la classe Commande pour que celle-ci soit transient. La commande correspond à un article ajouté au panier et à la quantité de cet article ajouté.
* Ajout d’une classe Facture qui correspond à l’achat de la totalité du panier.

Dans l’ordre des choses, le client choisit ses articles en les ajoutant à son panier. Chaque ajout d’un article dans le panier créer un objet commande qui sera transient. Cette commande représentant un seul article sera ajouté au panier qui pourra contenir une liste de commande et qui sera lui aussi transient.

Lorsque le client valide intégralement son panier, il créer un objet facture qui contient le panier du client. Cette facture sera elle en revanche enregistrée en base afin d’être envoyée au client et gardée pendant 2 ans comme preuve d’achat.

### 05/02/2019

* Installation de Tortoise Git.
* Migration du projet dans Git et GitHub.
* Synchronisation des 2 repositories (local et GitHub).
* Installation du plugin SonarLint 4.1 sur Eclipse.

### 06/02/2019

* Création d’un compte GitLab et import des repositories depuis GitHub.
* Ajout de la classe Commentaire. Il y a une relation OneToMany bidirectionnelle. Il faut d’ailleurs privilégier la bidirectionnalité à l’unidirectionnalité (sauf cas particulier) pour des raisons de performances en termes de requêtes SQL.

Ajout des méthodes addCommentaire et removeCommentaire selon le tutoriel suivant :

<https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/>

### 07/02/2019

D’après ce post (<https://stackoverflow.com/questions/3827494/inherited-abstract-class-with-jpa-hibernate>) on peut annoter les classes abstraites comme des entités avec une stratégie Single table inheritance.

### 08/02/2019

* Modification de la classe Commande en classe Selection. Elle correspond à un article sélectionné depuis une page du eshop. Cette sélection est ensuite envoyée dans le panier qui va regrouper l’ensemble des sélections que le client a choisi.
* Modification de la classe Facture en classe Commande. Elle correspond à

### 11/02/2019

* Ajout de la classe Post qui est une classe abstraite regroupant tous types de commentaires. On y retrouve autant les commentaires sur les articles que les posts qui sont faits dans la partie forum.

### 12/02/2019

Ajout de dépendances :

* Hibernate-validator version 6.0.13.Final
* Hibernate-search version 5.10.5.Final

### 17/02/2019

* Ajout de la balise <sourceDirectory> dans le fichier web.xml
* Création du fichier persistence.xml dans le dossier src/main/java/META-INF
* Création d’un fichier dao.properties dans le répertoire src/main/resources/com/projet/properties

### 18/02/2019

* Création des packages DAO et des interfaces pour les classes mères.
* Création du fichier applicationContext.xml pour la configuration du contexte de Spring.
* Ajout d’un el-resolver dans le fichier faces-config.xml

### 19/02/2019

Ajout de dépendances :

* Javax.inject version 1

### 20/02/2019

Ajout de dépendances :

* Spring-orm version 5.1.1.RELEASE

### 21/02/2019

* Séparation en 2 fichiers properties (db et hibernate).
* Ajout de l’import des fichiers properties dans le fichier applicationContext.xml (balise propertiesPlaceHolder).
* Début de la configuration d’un datasource dans Tomcat.

### 23/02/2019

* Modification du fichier ${CATALINA\_HOME}/conf/context.xml pour y ajouter une resource pour une datasource.
* Ajout de tags dans les fichiers web.xml et applicationContext.xml
* Ajout du jar jdbc pour Postgresql dans ${CATALINA\_HOME}/lib

### 02/03/2019

* Modification du fichier persistence.xml, les propriétés

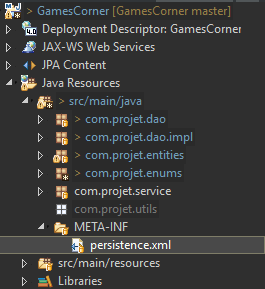
# Configuration du projet

## Configuration de la couche DAO

La partie DAO du projet est prise en charge par Hibernate avec les annotations de JPA.

D’autres ORM seront envisagés pour un autre projet (MyBatis) qui passe par des requêtes configurées dans un fichier xml et qui sont directement appelées dans la couche DAO.

La configuration d’Hibernate avec les annotations JPA dans le projet passe par la création d’un fichier persistence.xml dans le répertoire META-INF à la racine du projet.

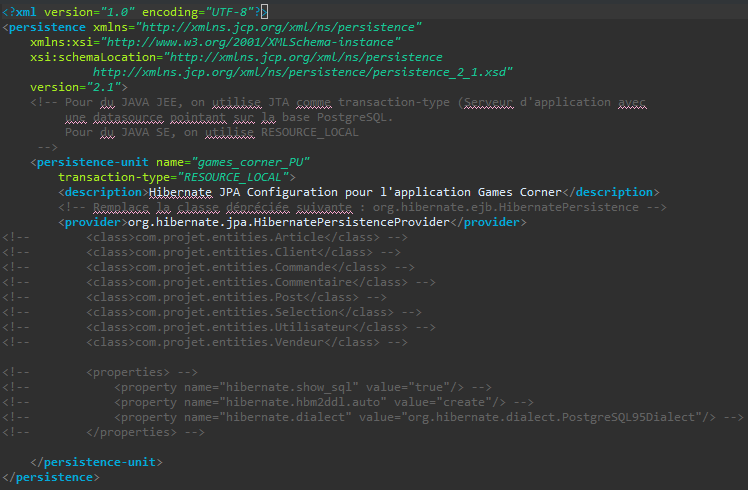


Le fichier persistence.xml permet de définir l’unité de persistence qui sera chargé par Spring. Les propriétés afférentes à la connexion à la base de données sont définies dans le fichier applicationContext.xml qui se trouve dans le même répertoire que le fichier web.xml

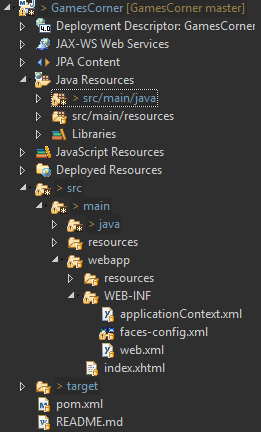
Ici, on se contente de définir la balise persistence-unit qui utilise une transaction de type RESOURCE-LOCAL car on n’utilise pas un serveur d’application qui prendra en charge les transactions de type JTA. En effet, pour se familiariser avec une configuration plus complète du projet, on va travailler sur un conteneur de servlet simple (Tomcat) dans un premier temps.

Pour configurer la persistence pour un serveur d’application du style Glassfish ou WildFly, il faudra rajouter une autre persistence-unit configurée pour définir JTA comme type de transaction.

Dans le cas de fichier présenté ci – dessous, en plus de ne pas préciser ici les propriétés de connexion à utiliser pour se connecter à la base, on ne précise pas non plus les classes model à utiliser. Cela sera fait dans le fichier applicationContext.xml à l’aide d’une propriété qui va scanner le package comportant ces classes.



Dans le répertoire WEB-INF, on rassemble les différents fichiers de configuration tels que web.xml, les fichiers de configuration de Spring ou encore la configuration de JSF. On suit l’arborescence suivante :



Le fichier est annoté pour préciser le rôle de chaque balise. Voici un résumé des différentes balises avec, dans un premier temps, les balises qui vont activer les fonctionnalités IoC de Spring :

* <**context:annotation-config** /> permet de détecter les annotations de Spring @Autowired ou @Configuration
* <**context:component-scan** base-package=*"com.projet"* /> indique à Spring quel package scanner pour trouver les beans annotés avec @Component.
* <**tx:annotation-driven** transaction-manager=*"transactionManager"* /> permet de détecter les annotations @Transactional dans la couche Service par le biais du bean défini portant le nom de transactionManager.

On définit ensuite les beans de connexion à la base de données, l’entity manager, la datasource, la configuration de la transaction, ...

* <**bean** id=*"propertyPlaceHolder"* class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*> permet de préciser à Spring le chemin des fichiers properties à utiliser depuis le classpath.
* <**bean** id=*"dataSource"* class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*> défini la datasource à utiliser (pool de connexion). Ce bean ne permet pas d’instancier une vrai datasource, il faudra pour cela implémenter un vrai datasource par la suite (c3P0 par exemple).

C’est ici que l’on détermine les propriétés de connexion à la base de données telles que le nom du driver associé à la base de données SGBD spécifique utilisée, l’url de la base, le nom et le mot de passe de l’utilisateur associé.

* <**bean** id=*"entityManagerFactory"* class=*"org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"*> défini la fabrique d’entité avec hibernate (la configuration des propriétés de ce bean permet justement d’indiquer que c’est l’implémentation Hibernate qui est utilisée pour la norme JPA).

C’est dans ce bean que l’on associe l’unité de persistence défini dans le fichier persistence.xml ainsi que la datasource créée précedemment.

C’est également ici que l’on définit le package que Spring doit analyser pour trouver les model du projet.

C’est également ici que l’on défini des paramètres propres à l’implémentation JPA pour la gestion du dialecte à utiliser ou encore la génération des tables en base à partir du model JAVA.

Il est possible d’utiliser 2 classes spring.orm différentes pour définir l’entityManagerFactory. On aurait pu utiliser la classe actuelle LocalContainerEntityManagerFactoryBean ou LocalEntityManagerFactoryBean. Les 2 ne possèdent pas les mêmes paramètres et il est plus facile de configurer ce bean avec la classe utilisée actuellement.

La seule différence réside dans le fait que la configuration des 2 beans sera différente en fonction de celui que l’on choisit d’utiliser.

Elle est plus facile avec LocalContainerEntityManagerFactoryBean alors qu’il faudrait configurer de façon programmatique avec LocalEntityManagerFactoryBean en créant une nouvelle classe utils.

Dans le cas présent, il est difficile de trouver de la documentation concrète sur internet de l’utilisation de LocalEntityManagerFactoryBean. Nous sommes donc passés LocalContainerEntityManagerFactoryBean pour que le projet puisse compiler après ajout des informations nécessaires au bean. On peut combiner ce bean avec le type de transaction RESOURCE\_LOCAL défini dans le fichier persistence.xml

* <**bean** id=*"transactionManager"* class=*"org.springframework.orm.jpa.JpaTransactionManager"*> défini quel type de transaction sera utilisée. Elle prend en paramètre l’entityManagerFactory défini précédemment.

### Principe de fonctionnement

C’est Spring en tant que conteneur léger qui va s’occuper de l’implémentation de l’entity manager factory lorsqu’on utilise la classe LocalContainerEntityManagerFactoryBean en tant que Bean à instancier dans le contexte de l’application.

Si on avait utilisé la classe LocalEntityManagerFactoryBean, il aurait fallu configurer soi-même de façon programmatique directement dans le code en passant par une classe util par exemple.

Le fait de laisser Spring gérer la configuration de l’entityManagerFactory facilite grandement le travail du développeur qui peut gérer les paramètres du bean directement depuis un fichier de configuration xml.

La configuration de cet entity manager factory passe par la définition de propriétés et comporte entre autre :

* Une datasource qui contient la définition des paramètres à utiliser pour se connecter à la base de données.
* La liaison à la persistence-unit créée dans le fichier persistence.xml
* Le chemin vers le fichier de persistence.xml
* Une propriété permettant de déterminer quel package Spring va scanner pour trouver les entités du projet. De ce fait, il n’est plus utile de préciser quelles sont les classes qui comptent comme entité dans le fichier de persistence.xml comme on pouvait le faire auparavant.
* Des propriétés spécifique à Hibernate qui permettent, entre autre, de générer les tables de la base de données à partir du model du projet.

# Dépendances POM

Le POM (Project Object Model) est un fichier xml contenant les informations sur le projet, la configuration de Maven.

On regroupe ici les différentes dépendances ainsi que les versions utilisées.

On utilise les versions suivantes :

* Junit-version : 4.10
* Spring-version : 5.1.1.RELEASE
* Hibernate-version : 5.3.7.Final
* Hibernate-validator-version : 6.0.13.Final
* Hibernate-search-version : 5.10.5.Final
* Postgresql-version : 42.2.5
* Jsf-version : 2.2.18
* Servlet-api-version : 3.1.0
* Commons-version : 1.3.3
* Primefaces-version : 6.2
* Joda-version : 2.10
* Bootstrap-version : 1.0.9
* Maven-compiler-plugin-version : 3.8.0
* Jdk-source : 1.8
* Jdk-target : 1.8

On a la liste des dépendances suivantes :

* jsf-api de com.sun.faces (Jsf-version).
* jsf-impl de com.sun.faces (Jsf-version).
* java.servlet-api de javax.servlet (Servlet-api-version).
* primefaces de org.primefaces (Primefaces-version).
* bootstrap de org.primefaces.themes (Bootstrap-version).
* commons-fileupload de commons-fileupload (Commons-version).
* joda-time de joda-time (Joda-version).
* postgresql de org.postgresql (Postgresql-version).
* spring-webmvc de org.springframework (Spring-version).
* spring-tx de org.springframework (Spring-version).
* spring-orm de org.springframework (Spring-version).
* javax-inject de javax.inject version 1.
* hibernate-core de org.hibernate (Hibernate-version).
* hibernate-search-orm de org.hibernate (Hibernate-search-version).
* hibernate-validator de org.hibernate.validator (Hibernate-validator-version).

# Problèmes rencontrés

* **« java compiler level does not match the version of the installed java project facet »:**

On a rajouté dans le fichier pom.xml la configuration du plugin maven-compiler-plugin en définissant le jdk source et target dans la configuration. Elle a été définie à 1.8 pour correspondre à la JDK indiquée dans l’IDE.

* **« Cannot change version of project facet Dynamic Web Module to 2.5 » :**

On a modifié le fichier org.eclipse.wst.common.project.facet.core.xml présent dans le dossier .settings en modifiant la version du web dynamic module par la ligne suivante :

<**installed** facet=*"jst.web"* version=*"4.0"*/>

Il faut également vérifier que la version du dynamic module dans le fichier web.xml soit bien à 4.0

* **Problème d’update de Maven « org.eclipse.wst.validation has been removed » :**

Lié au problème précédent.

On a supprimé le dossier .settings contenant le fichier concerné.

Lorsque ce problème est rencontré, cela signifie que le fichier org.eclipse.wst.validation.prefs a été supprimé et n’a pas été réecrit lors d’un update Maven.

Pour fixer ce problème, il faut supprimer le repository de maven contenant toutes les dépendances téléchargées. On clean ensuite les projets puis on fait un maven update.

Si on vérifie dans le dossier .settings, on voit que le fichier disparu a été recrée et que les dépendances ont été de nouveau téléchargés dans le repository local.

* **Problème de liaison entre le fichier CSS et le fichier xhtml**

src

`-- main

|-- java

|-- resources

`-- webapp

|-- resources

| |-- css

| | `-- style.css

| |-- images

| | `-- logo.png

| `-- js

| `-- script.js

|-- WEB-INF

| `-- web.xml

`-- index.xhtml

Il faut que le projet respecte cette architecture pour les ressources autres que les fichiers java.

Pour les fichiers d’internationalisation, on doit en revanche utiliser cette architecture :

src

`-- main

|-- java

|-- resources

| `-- com

| `-- example

| `-- i18n

| |-- text.properties

| |-- text\_en.properties

| |-- text\_es.properties

| `-- text\_nl.properties

:

Ce problème sera réglé dans une prochaine itération.

* **Balises JSF non interprétées dans les fichiers xhtml**

Les fichiers xhtml n’interprétaient pas les balises JSF jusqu’à présent. Pour corriger ce problème, il faut corriger la configuration du fichier web.xml en rajoutant une balise url-pattern dans le servlet-mapping :

<**url-pattern**>\*.xhtml</**url-pattern**>

Bien penser à relancer le serveur après avoir modifié le fichier web.xml pour prendre en compte les modifications.

* **Avertissement JSF1064**

En lançant le serveur et en chargeant une page JSF, on obtient l’avertissement suivant :

AVERTISSEMENT: JSF1064 : Impossible de localiser ou de servir une ressource, images/ui-bg\_highlight-hard\_70\_000000\_1x100.png, depuis la bibliothèque primefaces-bootstrap.

Pour résoudre ce problème, il faut télécharger le fichier ui-bg\_highlight-hard\_70\_000000\_1x100.png et le placer dans le dossier images directement dans le jar bootstrap.

* **Eclipse Marketplace ne s’ouvre pas**

Il peut arriver que l’on ne puisse pas accéder au market place d’Eclipse à partir de l’onglet Help 🡪 Eclipse Marketplace …

Cela est du dans notre cas à l’espace de Program Files qui entraine une erreur de caractère dans l’IDE.

Pour régler ce problème, il faut aller dans l’onglet Help 🡪 Install New Software 🡪 Add et rentrer cette adresse dans les 2 champs :

<http://download.eclipse.org/mpc/photon/>

Il faut par la suite télécharger cette source et redémarrer Eclipse.

* **Problèmes Git avec le dossier target**

Dans le projet Git de base, le fichier .gitignore n’a pas été créé à la main mais a été créé automatiquement depuis github.

Il faut donc modifier le fichier .gitignore pour que le dossier target ne sois pas inclus dans les commit. On rajoute ou modifie la ligne comme suit :

target/

Il faut ensuite aller dans l’onglet Windows 🡪 Preferences 🡪 Team 🡪 Git 🡪 Projects et cocher la dernière ligne.

* **Error creating bean with name 'articleServiceImpl': Unsatisfied dependency expressed through field 'articleDAO'; nested exception is org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'com.projet.dao.IArticleDAO' available: expected at least 1 bean which qualifies as autowire candidate. Dependency annotations:** [**{@org.springframework.beans.factory.annotation.Autowired(required=true)}**](mailto:%7b@org.springframework.beans.factory.annotation.Autowired(required=true)%7d)