

UVSQ, UNIVERSITÉ PARIS-SACLAY

MASTER DATA SCALE

MicroProjet J2EE

Système d'Enchère

Jingwei ZUO
Yixuan ZHANG

Octobre 2017 — Novembre 2017

Contents

1	Introduction	1
1.1	Sujet de projet	1
2	Architecture du système	1
2.1	Pattern designs	1
2.2	Structure de données persistantes	3
2.3	Façade de services, synchrone ou asynchrone	3
2.4	UML Design	4
3	Implémentation du système	5
3.1	Technologies utilisées dans le projet	5
3.2	Mécanisme de sécurisation	5
4	Problèmes rencontrés pendant l'implémentation	7
4.1	Problème A	7
4.2	Problème B	8
4.3	Problème C	9
5	Description de scénario	9
6	Démonstration et tutorial d'utilisation	10
7	Conclusion	10
8	Annexe	11

1 Introduction

1.1 Sujet de projet

Ce projet a pour but de designer et d'implémenter un système d'enchères. Ce système permet aux utilisateurs à enregistrer et à proposer des objets à vendre. Ainsi, les utilisateurs peuvent demander les objets proposés par les autres.

2 Architecture du système

2.1 Pattern designs

Les patterns décrivent les problèmes principaux pendant le design et l'implémentation du système. Pour mieux construire notre middleware, on a proposé plusieurs pattern designs qui sont montrés au-dessous:

DAO Pattern :

- Contexte : Dans un projet industriel, il existe des opérations de bas niveau et des services de haut niveau.
- Problème : Les manipulations de bas niveau sont confondues avec les services de haut niveau, qui n'est pas favorable pour la maintenance du système.
- Solution : En utilisant les interfaces des opérations, on peut bien séparer la logique du programme avec les données.
- Diagramme : Comme montré au-dessous (Figure DAO Pattern)

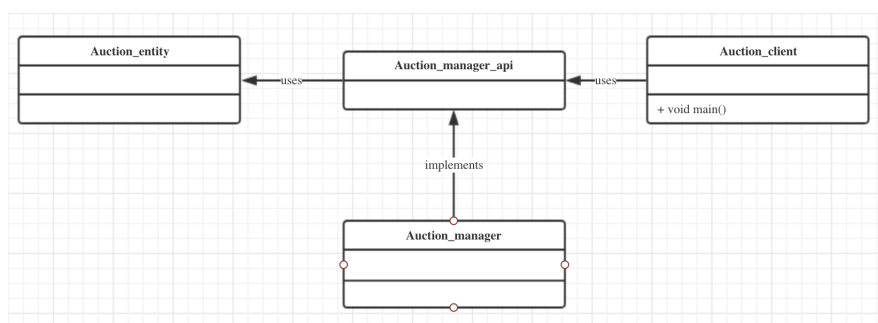


Figure 1: DAO Pattern

Proxy Pattern :

- Contexte : Dans un projet industriel, il existe des objets client et objets cibles.
- Problème : Les objets client doivent gérer les conditions particulières d'accès aux services de l'objet cible. Par exemple, le coût pour créer un objet du serveur est élevé, ainsi que le problème de sécurité, qui n'est pas favorable pour la consultation directe du serveur.
- Solution : L'objet Proxy offre la même interface que celle de l'objet cible. De ce fait, l'objet proxy prend le rôle de serveur pour communiquer avec le client.
- Diagramme : Dans notre système, on a utilisé "EJB". Comme montré dans la figure ci-dessous, EJB est censé comme un "proxy" qui prend le rôle de serveur pour donner des accès aux clients. (Figure Proxy Pattern)

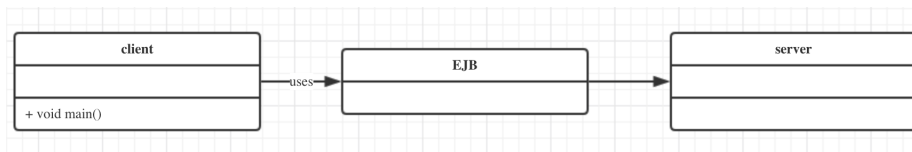


Figure 2: Proxy Pattern

Synchronous and Asynchronous calls Pattern :

- Contexte : Dans notre système, il existe des besoins d'utilisateurs pour consulter les données en même temps.
- Problème : En mode synchrone, le client doit toujours attendre la réponse avant d'émettre une 2ème demande. Cela va prendre du temps pendant l'exécution du programme. Par contre, en mode asynchrone, c'est possible d'avoir le problème de concurrence.
- Solution : En combinant ces 2 modes là dans notre système, on peut bien contrôler la performance du programme.
- Diagramme : Ici, on montre plutôt le mode asynchrone, comme dans notre système, la plupart de méthodes sont en mode synchrone. La diagramme est montré dans la Figure 4, section 3.1

Transfer Object Pattern :

- Contexte : Le système veut vérifier si un objet appartient à l'utilisateur.

- Problème : Il faut chercher tous les objets de l'utilisateur sur le serveur pour vérifier que l'objet appartient à l'utilisateur. De ce cas là, il aura plusieurs transactions de l'objet. À mesure que l'utilisation de ces méthodes distantes augmente, les performances des applications peuvent se dégrader de manière significative. Par conséquent, l'utilisation de plusieurs appels pour obtenir des méthodes renvoyant des valeurs d'attribut uniques est inefficace pour obtenir des valeurs de données à partir d'un bean enterprise.
- Solution : En utilisant "Transfert Object Pattern", on peut donc récupérer un objet qui possède plusieurs sous-objets. De cette manière, le coût de transmission sera diminué.
- Diagramme : (Figure Transfer Object Pattern)

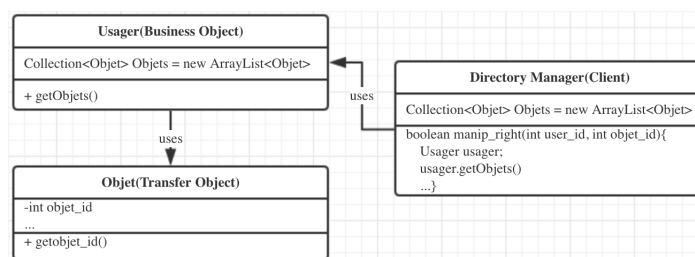


Figure 3: Transfer Object Pattern

2.2 Structure de données persistantes

Nous avons construit 3 tables pour stocker des données, dont les noms sont "Usager", "Objet" et "Auction".

- Pour les tables "Usager" et "Objet", ils enregistrent des informations de base sur des usagers et des objets.
- Dans la table "Auction"
 - il y a un attribut "max_duration" qui indique le tour maximum d'auktion, ainsi, l'attribut "duration" qui montre le tour actuel, donc si "duration" est supérieur à "max_duration", aucun bid sera accepté.
 - "inc_bidding" est l'incrémentation de prix dans chaque tour, si un usager propose un bid, le prix augmente automatiquement avec cette incrémentation.
 - "id_bidder" enregistre l'identifiant de dernier bidder.

2.3 Façade de services, synchrone ou asynchrone

Différences entre les services synchrones et asynchrones:

Dans un service synchrone, le client doit avoir la réponse du serveur avant d'émettre le 2ème paquet de communication. Cependant, le service asynchrone permet au client d'émettre les demandes sans avoir la réponse de la demande précédente.

Dans notre système, pour faciliter les accès aux données, on a déterminé d'utiliser la communication asynchrone pour certaines méthodes.

Pendant une enchère, si les clients veulent récupérer des informations sur un objet ou celles de l'enchère elle-même. On suppose que du côté EJB/serveur, il y aura un délai pour préparer les données à renvoyer aux clients. Dans ce cas là, voici les différences pour la communication en mode synchrone et celle en mode asynchrone :

Mode synchrone Chaque client va attendre jusqu'il aura reçu la réponse de serveur. Cela implique que la 2ème demande d'information arrive toujours au serveur après la réception de résultat de la 1ère demande. Dans la pratique, du côté serveur, si le traitement prend du temps, le client ne pourra pas émettre 2 demandes d'information en même temps. Par contre, il faut toujours consulter les informations du serveur l'un par l'un.

Mode asynchrone Les deux demandes d'information d'objet peuvent arriver au serveur sans délai. Le client peut consulter la réponse ultérieurement au lieu de l'attendre. De ce fait, on a finalement déterminé le mode asynchrone pour récupérer des informations d'objet et d'enchère.

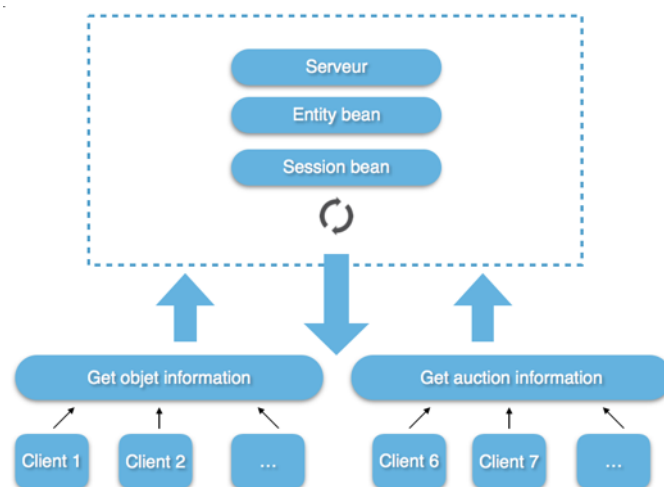


Figure 4: Synchrone Asynchrone

2.4 UML Design

Le diagramme UML est montré dans la figure 5

3 Implémentation du système

3.1 Technologies utilisées dans le projet

- Maven : Se fondant sur POMproject Object Module, Maven est un outil de gestion et d'automatisation de production des projets logiciels Java et Java EE.
- Junit Test : C'est un framework Java open source pour la réalisation de tests unitaires.
- EJB3.2 :
 1. Stateful Session Bean : La différence essentielle entre le bean stateful et le bean stateless, est leur cycle de vie. Le bean stateful garde son état pour ses clients. Quand le client instancie un session bean stateful, l'état de bean va être initialisé. Dans notre système, comme le client accède au bean en utilisant son identifiant et son mot de passe. Le bean doit avoir un état pour chaque client. De ce fait, le Stateful Session Bean est le meilleur choix.
 2. Entity Bean : C'est un mécanisme de persistance qui sert à relier les données dans le serveur avec les objets dans le programme. Depuis la version 3.0 de la spécification EJB, les EJB entité sont directement liés à la base de données via un mapping objet-relationnel(ORM). Dans notre projet, ce mapping est défini directement dans le code Java en utilisant des annotations. Cette nouvelle interface de programmation des EJB entité est appelée Java Persistance API (JPA).
- EclipseLink(JPA)+JPQL : Pour faciliter l'utilisation de JPA, on a utilisé "EclipseLink" dans notre projet. EclipseLink, le standard officiel de JPA2.0, est un JPA framework comme Hibernate.
- Glassfish : J2EE serveur d'application qui sert à manager les EJBs.
- Derby : Serveur local qui est vraiment légère pour faciliter le test.

Dans la pratique, les beans sont distribués sur les différents serveurs. En utilisant JTA, on peut donc choisir les sources venant de différents serveurs. Pour faciliter le développement, on a mis les beans/APIs sous un même endroit. Ainsi, pour les sources d'entités, on a pris la source locale pour tester.

3.2 Mécanisme de sécurisation

1. SQL Injection :

Dans notre projet, on a utilisé le JPA framework EclipseLink et le JPQL pour persister les objets et consulter les données dans la base de données.

SQL généré par EclipseLink n'est pas vulnérable aux attaques par injection SQL. Les attaques par injection SQL se produisent lorsque les paramètres exposés à un utilisateur final. De ce

fait, quand on utilise JPQL dans notre programme, c'est possible d'avoir le problème de SQL injection. Par exemple, si l'on écrit une instruction SQL ci-dessous:

- `String sql = "SELECT u FROM User u WHERE id=" + id ;`

L'attaquant peut donc modifier le format de l'id (e.g. `String id = " 1' or 'x'='x' "`) pour récupérer les informations de tous les clients, SQL devient donc:

- `"SELECT u FROM User u WHERE id='1' or 'x'='x' ;`

Solution :

1. En utilisant le query paramétré, les opérations à exécuter dans SQL sont donc fixés. Dans ce cas là, les attaquants ne peuvent pas rajouter les opérations (e.g. l'opération "or" qui est montré précédemment) par recharger les données entrées.
2. On peut fixer le format pour les données entrées, ou filtrer les entrées. Par exemple, dans l'image ci-dessus, on a fixé le format de données entrées à "int", dans ce cas là, il n'aura pas de paramètres qui sont exposées aux attaquants.

2. Mécanisme d'authentification :

Dans notre système, il existe deux types de clients. Ce système permet d'avoir un seul administrateur, et d'avoir plusieurs clients qui participent des enchères.

1. Administration client: Le client administrateur a du droit pour gérer les usagers, les objets. Ainsi, il peut mettre à jour des droits d'usagers à proposer des enchères. De ce fait, dans le bean "DirectoryManager", on a défini des méthodes qui ne sont accessibles que pour l'administrateur :
 - Add/remove users
 - Add/remove objects
 - Update user rights

Les autres méthodes ayant une fonctionnalité de consulter des informations d'utilisateur et d'objet, sont accessibles pour tous les clients authentifiés.

2. Auction client:

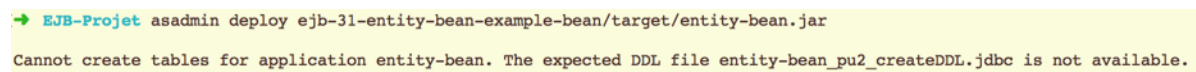
- Le droit d'accès: Notre système permet d'avoir plusieurs Auction Client en même temps. Cependant, chaque client doit accéder au système avec son propre identifiant et mot de passe. Comme parlé précédemment, c'est aussi la raison pour laquelle qu'on utilise "Session bean stateful" dans le système. Si et seulement si un client a été ajouté dans la base de donnée par un administrateur, il peut donc avoir le droit d'accès au système.

- Le droit de gérer une enchère: Avant de gérer une enchère (e.g. ajouter, démarrer et supprimer) d'un objet, le client doit prouver son droit sur cet objet. Cela implique qu'il faut vérifier si l'objet appartient au client. De ce fait, le client n'aura que le droit pour gérer son propre enchère, mais il pourra consulter les informations de toutes les enchères existantes dans le système.

4 Problèmes rencontrés pendant l'implémentation

4.1 Problème A

Les détails sont montrés dans l'image ci-dessous: (Figure Problème A)



```

→ EJB-Projet asadmin deploy ejb-31-entity-bean-example-bean/target/entity-bean.jar
Cannot create tables for application entity-bean. The expected DDL file entity-bean_pu2_createDDL.jdbc is not available.

```

Figure 5: Problème A

Intention originale : Pour que chaque session Bean (Auction manager et Directory manager) puisse consulter son propre entité/serveur (limiter les droits de chaque session Bean), on a défini 2 contextes différents pour les deux sessions Beans :

- Configurations
 1. Dans le fichier "persistence.xml", on a défini 2 « persistence-unit » qui partagent une source locale.
 - <persistence-unit name="pu1">
 - <persistence-unit name="pu2">
 - <jta-data-source>jdbc/_default</jta-data-source>
 2. Pour manager le contexte d'entité, au lieu d'utiliser le management par l'application, GlassFish utilise par default le conteneur pour automatiser le management. Pour définir les contextes dans les Beans, on ajoute alors des annotations dans les programmes (un seul contexte dans un bean) :
 - @PersistenceContext(unitName = "pu1"), dans "Directory manager bean"
 - @PersistenceContext(unitName = "pu2"), dans "Auction manager bean"
 3. JPA supporte 2 types de transactions
 - RESOURCE_LOCAL, qui se fonde sur JDBC pour manager les transactions locales.
 - JTA, Java Transaction API.

Dans le mode que les entités sont managés par le conteneur, le type de transaction est par default JTA. En utilisant JTA, on peut donc choisir plusieurs sources venant de différents serveurs.

Problème Le programme n'arrive pas à générer le fichier DDL pour la connexion avec la base de données, sous le contexte "pu2".

Débogage

1. Vérifier la connexion avec la base de données, sous le contexte "pu1"
 - La connexion a été bien établie.
2. - Changer l'ordre de définition de contextes dans le fichier "persistence.xml", vérifier les connexions.
 - La connexion sous le premier contexte est toujours établie, mais pas pour celle sous le deuxième contexte.

Conclusion : comme ce qu'on a défini dans "persistence.xml", les deux contextes partagent une source locale. De ce fait, le programme ne peut pas créer 2 DDL fichiers pour la connexion avec une même source. Autrement dit, on ne peut pas installer 2 connexions avec une source/une serveur. C'est la raison pour laquelle le programme produit une erreur.

4.2 Problème B

Génération de la clé primaire dan la base de donnée :

Pour générer automatiquement l'identifiant (clé primaire) d'objet, d'enchère et d'usager dans la base de données, on utilise l'annotation

@GeneratedValue(strategy=GenerationType.AUTO) avant chaque définition de clé primaire.

- Problème : Cependant, quand on vérifie les données dans les tables, nous avons remarqué que la clé primaire des tables ne commence pas par 1. Les 3 tables partagent une série de numéro. Le problème est montré dans les images ci-dessous:
- Débogage : Si l'on utilise l'annotation
 - @GeneratedValue(strategy=GenerationType.AUTO)

la clé primaire est générée de façon automatique lors de l'insertion en base. Comme on n'a pas défini le générateur de la clé primaire dans les entités, les 3 tables partagent donc un générateur

- Solution : La génération de la clé primaire se fera par une séquence définie dans le SGBD, auquel on ajoutera l'attribut "generator". Cette stratégie doit être utilisée avec une autre annotation qui est @SequenceGenerator. Cette annotation possède l'attribut "name" pour le nom du generator, l'attribut "sequenceName" pour le nom de la séquence et enfin "allocationSize" qui est l'incrément de la valeur de la séquence. Par exemple, dans entity bean "auction", on ajoute les annotations ci-dessous :

- @GeneratedValue(strategy = GenerationType.SEQUENCE,generator="auctionGenerator")
- @SequenceGenerator(name = "auctionGenerator", sequenceName = "Auction_sequence", initialValue = 1, allocationSize = 1)

4.3 Problème C

Le choix de mode de communication pour la méthode “Boolean bid(int objet_id, int bidder_id)” dans Auction manager.

Intention originale : Pour que les clients puissent bidder un objet en même temps, on a choisi le mode asynchrone pour cette méthode.

- Problème : Quand on fait 3 bids pour l’objet 2, nous avons remarqué que le résultat de l’objet 2 n’est pas montré comme ce qu’on envisage. Il n’y a qu’un seul bid qui est réussi.
- Débogage : Le serveur log a montré que le bidding est en mode asynchrone. Selon le résultat d’exécution, on peut savoir qu’il avait un seul bid réussi sur l’objet 2.
- Conclusion : En mode asynchrone, il aura une concurrence de ressource si les clients font le bid pour un objet en même temps.

```
Objet 2 information:
  Description of object: Notebook
  Category of objet: objet public
Bidding is: true
Bidding is: true
Bidding is: true
Auction for objet 2 information:
  objet_id is: 2
  state is: active
  duration is: 1
  max_duration is: 5
  starting_price is: 50.0
  current_price is: 51.5
  increment for bidding is: 1.5
  the id of last bidder is: 5
```

Figure 6: Problème C

5 Description de scénario

Application "administrateur" :

Ayant été authentifié, l’administrateur peut accéder au système pour manipuler les objets ou les usagers. Ainsi, il peut modifier le droits d’usagers.

- Ajouter un usager : indiquer le nom, prénom, mot de passe, adresse, mail, et le nombre maximal de proposition. Une fois que l’opération est réussie, le système va retourner un user_id

- Supprimer un usager : indiquer user_id
- Ajouter un objet : indiquer son propriétaire, la description, la catégorie. Une fois que l'opération est réussie, le système va retourner un objet_id
- Supprimer un objet : indiquer objet_id

Application "Auction_client" :

Une fois que le client a été authentifié, il peut accéder aux services correspondants à son droit.

- Ajouter une enchère : indiquer objet_id, le prix départ, la durée maximale et l'incrément de prix, le programme va retourner une valeur pour dire que si cette opération est réussie.
- Démarrer une enchère : changer l'état d'enchère à "active".
- Fermer une enchère : changer l'état d'enchère à "closed".
- Faire un bid sur un objet : le système va d'abord vérifier si la durée est dépassée à la durée maximale d'un objet et si l'objet est en état "active". Si oui, le bid est réussi.
- Consulter la liste d'enchères selon l'état, le système va retourner tous les "objet_id" qui répondent aux critères (état).
- Consulter l'information d'un objet en utilisant son objet_id.
- Consulter l'information d'une enchère selon "objet_id".

6 Démonstration et tutorial d'utilisation

Dans le dossier "EJB_Projet", il y a un script " runcmd.sh" pour exécuter la démonstration automatiquement. Quand on lance le programme, il va d'abord deployer l'EJB au serveur local(GlassFish), et puis, il va exécuter notre application:

- Sous répertoire "EJB_Projet", taper " sh runcmd.sh "
- Application d'administrateur, mot de passe est: admin
- Application de client d'enchère: Exécution automatique avec le scénario prédéfini

7 Conclusion

Dans ce projet, on a utilisé les technologies J2EE pour implémenter le système d'enchère. Se fondant sur EJB 3.2(Stateful Session Bean, Entity Bean), on peut accéder aux méthodes à distance et exécuter l'application d'enchère à locale. Par le mécanisme d'authentification fourni par le système, les clients peuvent avoir les droits différents pour accéder aux services (e.g. le rôle d'administrateur, gestion d'enchères, consultation d'informations).

8 Annexe

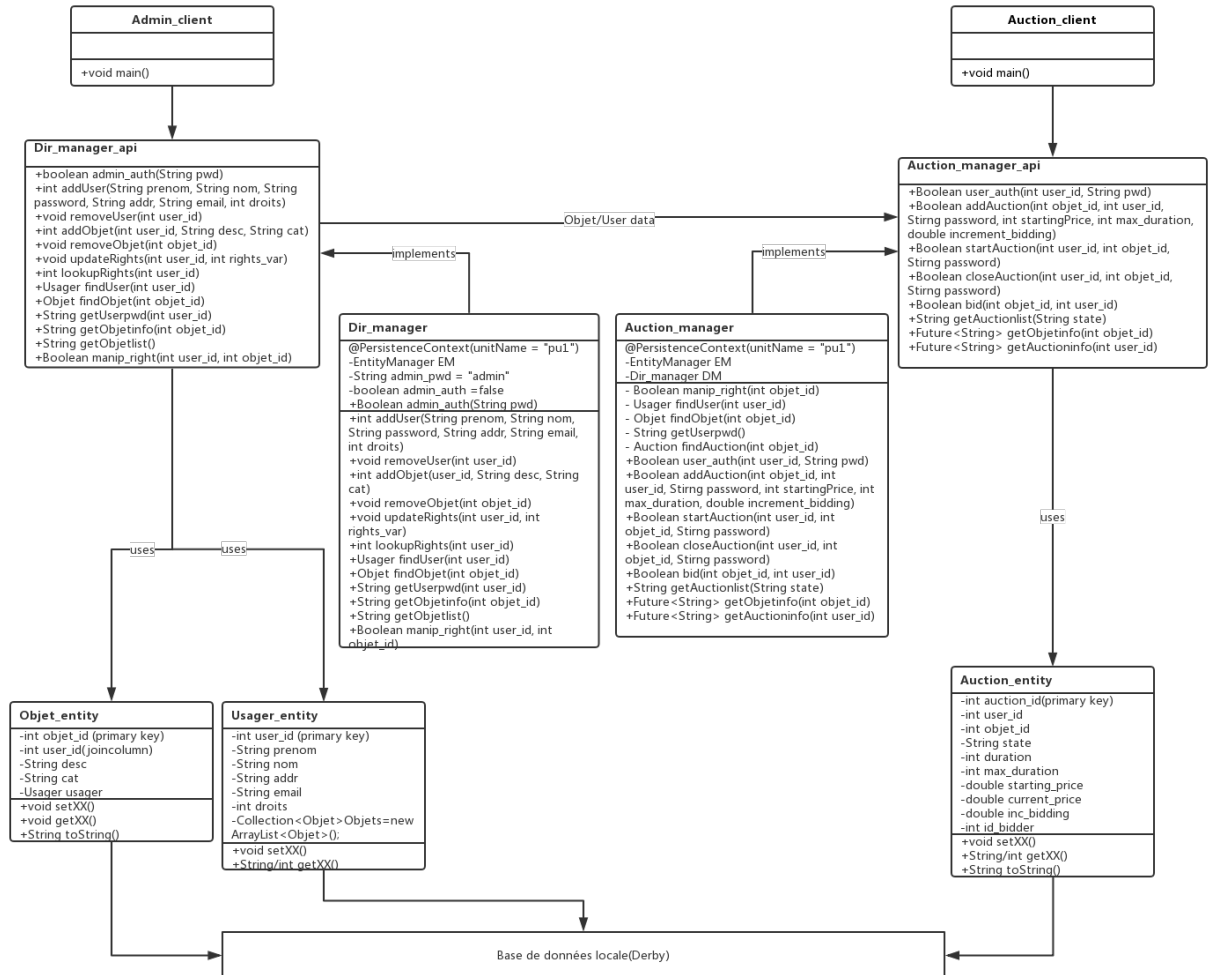


Figure 7: UML_Structure