Secteur Tertiaire Informatique

Filière « Etude et développement »

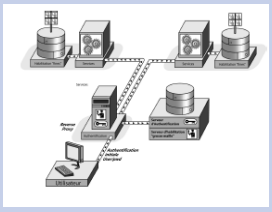
Création d'une API et configuration d'un ORM

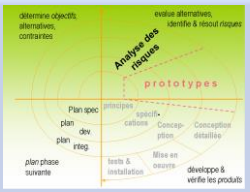
Java – ORM – Spring

Apprentissage

Mise en situation

Evaluation





1. INTRODUCTION

1.1 OBJECTIF DE DEVELOPPEMENT

Ce projet a pour objectif de vous permettre de créer une Web API REST permettant d'effectuer des opérations CRUD sur des enregistrements d'une base de données.

CREATE / READ / UPDATE / DELETE

Cette base de données a pour objectif de représenter un système **bancaire minimaliste**.

Vous allez créer les entités (et donc les tables) suivantes :

- **Account** : représente des comptes bancaires
- **Client** : représente des clients
- **Insurance** : représente des contrats d'assurance

1.2 REGLES METIER

Voici une liste de règle métier que vous allez pouvoir implémenter :

- Un utilisateur peut avoir un ou plusieurs comptes bancaires et un compte bancaire n'a qu'un seul propriétaire. Un utilisateur peut ne pas avoir de compte bancaire.
- Un utilisateur peut souscrire à une ou plusieurs assurances. Chaque assurance peut être associée à plusieurs clients. Un utilisateur n'est pas obligé de souscrire à une assurance.

  Les types d'assurances de base sont : assurance habitation, assurance santé, assurance vie, assurance automobile, assurance scolaire, responsabilité civile personnelle ou professionnelle.

1.3 STACK TECHNIQUE

Vous allez mettre en place une REST Api basée sur la « stack technique » suivante :

- Code serveur Jakarta EE :
  - Spring (avec Springboot en outil de développement) ;
  - JPA – Java Persistance API
- Base de données :
  - PostgreSQL

Vous allez apprendre à mettre en place, dans un projet Spring, un ORM basé sur l'API JPA.

Lisez la partie « Les ORM » du lien suivant pour en apprendre un peu plus :
https://gayerie.dev/epsi-b3-orm/javaee_orm/intro.html

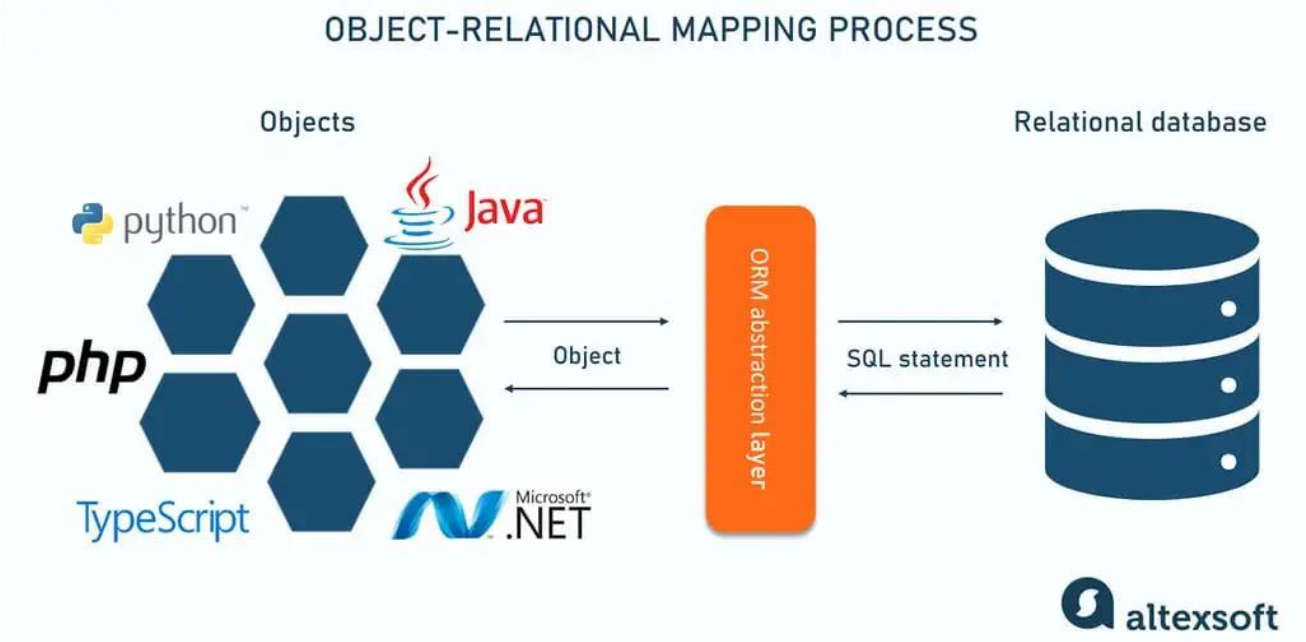


Figure 1 Image illustrant le principe d'un ORM

2. TRAVAIL A EFFECTUER

2.1 FORK DU PROJET

Vous pourrez commencer à travailler sur le projet en utilisant la base disponible communiquée par votre formateur.

2.2 CREATION DE LA BASE DE DONNEES

Démarrer la base de données fournie en utilisant Docker. Le fichier de configuration Docker fourni est un « **docker-compose.yml** ».

Un tel fichier permet d'automatiser la création d'une image et le lancement d'un conteneur, ceci en une seule commande :

```
docker compose up
```

Effectuez la commande dans un terminal (veillez à bien lancer dans le sous-dossier contenant le fichier « **docker-compose.yml** ») puis vérifiez que la base de données est fonctionnelle en vous connectant avec un client SQL.

Vous trouverez les informations de connexion contenues dans le fichier « **docker-compose.yml** ».

Attention

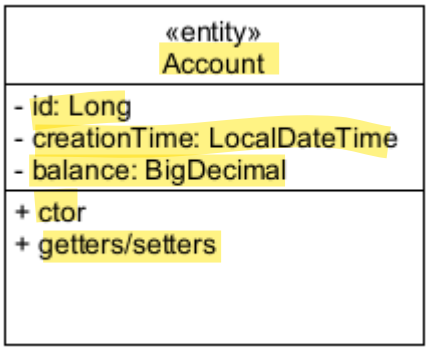
Cette base de données sera à compléter au fur-et-à-mesure de ce TP.

2.3 PREMIERE ENTITE : ACCOUNT

2.3.1 Passage de la classe « Account » en « entity »

Développez l'entité « Account » du package « **fr.afpa.orm.entities** » en suivant les indication des « TODO ».

Ci-dessous l'UML la représentant :



Attention

De nombreux liens sont donnés dans les commentaires du code.

Veillez à bien en prendre connaissance afin de bien comprendre votre implémentation.

2.3.2 Implémentation d'un « repository »

Afin de pouvoir effectuer des opérations CRUD sur la base de données en utilisant les entités, implémentez le code attendu par le « TODO » de la classe « **fr.afpa.orm.repositories.AccountRepository ».**

Attention

Les « repositories » sont les classes qui ont pour objectif de faire l'accès à la base de données.

Jusqu'à présent nous les avons appelées « DAO ». Bien prendre en compte qu'il s'agit du même concept.

2.3.3 Implémentation du contrôleur Rest

Complétez le code de la classe « **fr.afpa.orm.web.controllers.AccountRestController** » pour mettre en place votre API.

C'est le contrôleur qui pourra faire appel aux DAO (autrement appelé « reposit

2.3.4 Test des endpoints

Testez les endpoints que vous venez d'implémenter.

2.4 Relation OneToMany

2.4.1 Modification de la base de données

Vous allez mettre en place la structure UML suivante au sein de votre application :

Rappel des règles métier :

- un utilisateur peut avoir un ou plusieurs comptes ;
- un compte n'a qu'un seul propriétaire.

A partir de cet UML :

1. Établissez le **diagramme MCD** permettant de représenter la structure d'une base de données
2. Modifiez **la base de données en accord avec le MLD** grâce à votre client SGBD pour intégrer cette nouvelle architecture.

Attention

Le type de la clef primaire de la table « user » doit être **UUID**.

UUID est l'acronyme de « **Universally Unique Identifier** » (identifiant universel unique) peut-être généré automatiquement par un SGBD.

L'UUID est à privilégier dans le cas d'accès à des données exposées.

Exemple dans votre cas : vous allez développer un « endpoint » « **/api/users** » qui permettra au client de récupérer des informations utilisateurs. Dans le cas d'utilisation d'un identifiant auto-incrémenté un attaquant peut deviner les identifiants utilisateurs (logiquement « 1..2..3… ») et tenter d'attaquer le endpoint en utilisant un chemin pour lequel il y a une donnée (par exemple « **/api/users/2** »

Il sera beaucoup difficile à un attaquant de trouver un UUID correct.

Plus d'informations sur UUID en PostgreSQL :
https://www.postgresql.org/docs/current/datatype-uuid.html

2.4.2 Ajout d'un jeu d'essai

Ajoutez des enregistrements afin de construire un jeu d'essai exploitable.

2.4.3 Implémentation des entités

Une fois la base de données correcte, vous pourrez implémenter l'entité « **Client** » et modifier « **Account** » au sein de votre code Java.

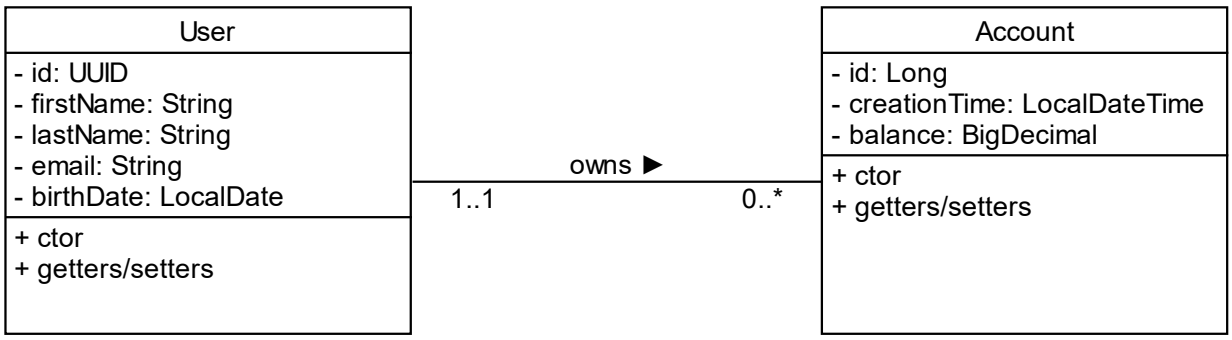
Points de vigilance lors du développement :

- Déclarez autant d'attributs de classe qu'il y a de colonnes de table
- Adaptez le type Java en fonction du type de base de données
- Pour le type UUID, utilisez la classe « UUID » du package « `java.util.UUID` »

2.4.3.1 Implémentation de la relation

A un certain stade, vous allez devoir implémenter le lien entre un utilisateur et son/ses comptes bancaires.

Vous allez, au niveau des classes Java, implémenter le diagramme UML suivant :



La relation « **owns** » indique deux contraintes :

1. 1 utilisateur détient de 0 à un nombre indéterminé de comptes (cardinalité « 0..* »)
2. 1 compte n'a qu'un seul utilisateur propriétaire

En Java, pour satisfaire la première contrainte, il faut ajouter à la classe « User » un attribut tel que :

```
private List<Account> accounts;
```

Pou satisfaire la deuxième contrainte, il faut également ajouter à la classe « Account » un attribut tel que :

```
private User owner;
```

Il va également falloir ajouter des annotations à ces attributs pour configurer l'ORM.

Instruction

Prenez connaissance du fonctionnement de certaines annotations utiles dans votre cas en regardant le cours suivant : https://koor.fr/Java/TutorialJEE/jee_jpa_many_to_one.wp

Instruction

Implémentez les classes « **User** » et « **Account** » en ajoutant bien les relations aux attributs qui en ont besoin.

2.4.4 Implémentation et test de l'API

Implémentez un nouveau contrôleur nommé « **UserRestController** ».

Testez un **endpoint** tel que « **/api/~~users~~** clients » avec une requête « GET ».

Attention

Que remarquez-vous ?

Si tout se passe bien, votre API devrait « **crasher** » (pas d'inquiétude, c'est normal).
Si ça fonctionne, c'est qu'il y a un problème.

2.4.5 Mise en place de DTO

Vous faites face à une **récursion infinie** liée au mécanisme de sérialisation.

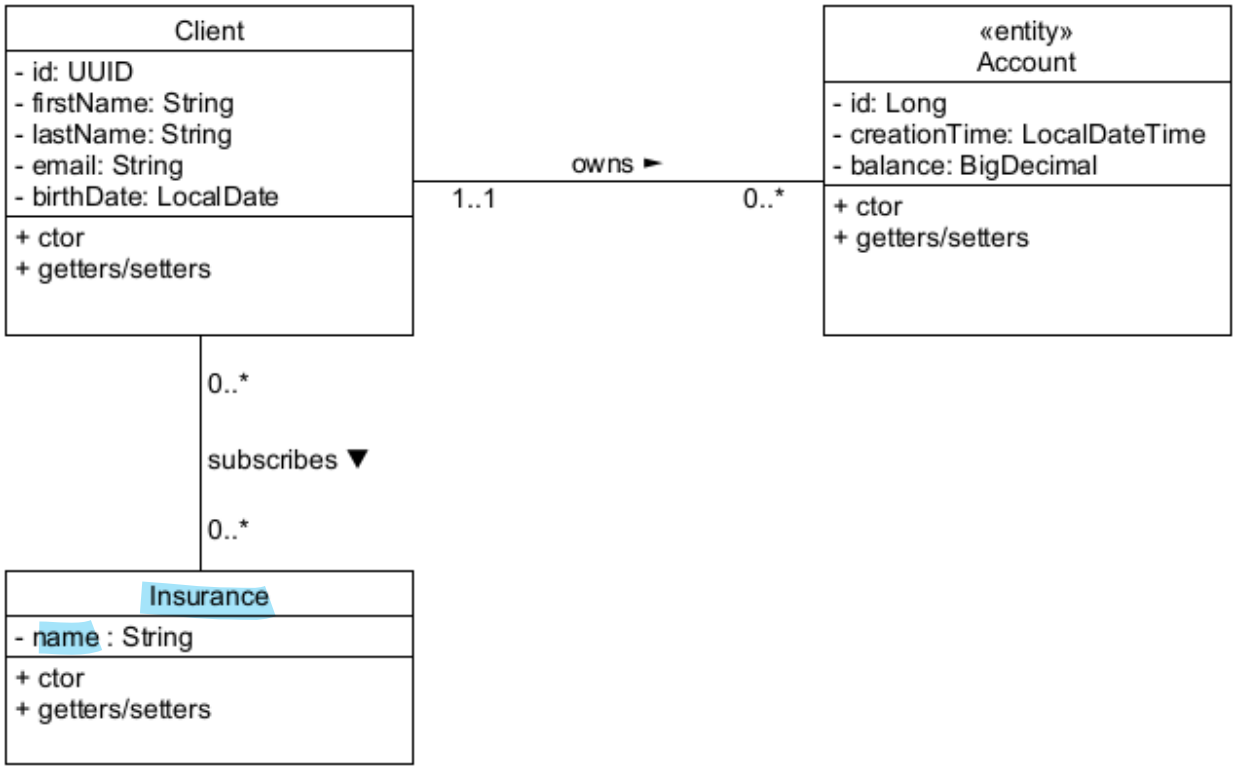
Suivez les recommandations de l'article suivant afin de mettre en place un design pattern appelé **DTO** (« Data Transfer Object ») et ainsi corriger la boucle infinie : https://medium.com/@zubeyrdamar/java-spring-boot-handling-infinite-recursion-a95fe5a53c92

2.5 RELATION MANYTOMANY

Cette partie va vous permettre d'implémenter les règles métier suivantes :

- Un utilisateur peut souscrire à une ou plusieurs assurances. Chaque assurance peut être associée à plusieurs clients. Un utilisateur n'est pas obligé de souscrire à une assurance.

Les types d'assurances de base sont : assurance habitation, assurance santé, assurance vie, assurance automobile, assurance scolaire, responsabilité civile personnelle ou professionnelle.



Instruction

Cette fois-ci le diagramme E/R ne vous est pas fourni.

A vous d'ajouter une table et de modifier la base de données de façon à être en accord avec ce diagramme UML.

Il vous est demandé de créer le MCD associé à cette nouvelle modélisation.

Pour implémenter la relation « **ManyToMany** » au niveau des classes Java, prenez connaissance du fonctionnement de l'annotation « **@ManyToMany** » en regardant le cours suivant : https://koor.fr/Java/TutorialJEE/jee_jpa_many_to_many.wp

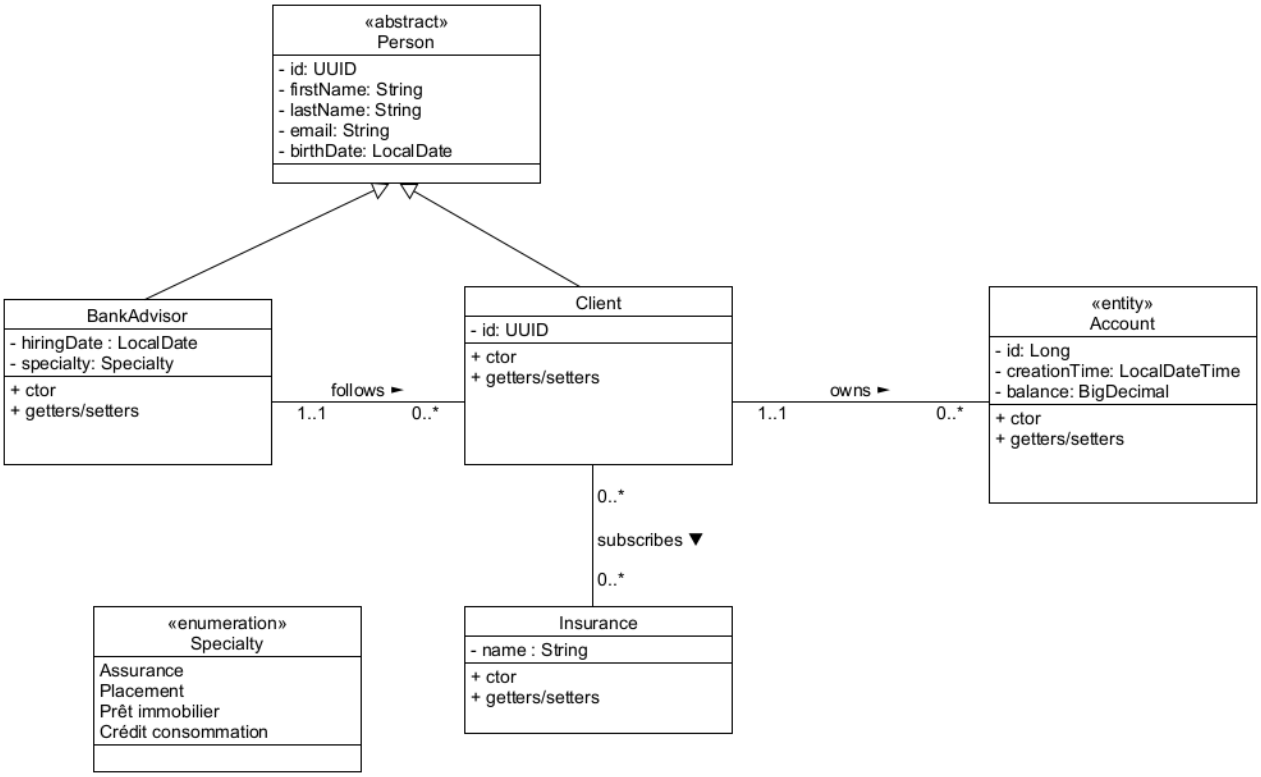
3. POUR ALLER PLUS LOIN

3.1 MISE EN PLACE DE L'HERITAGE

Voici un ensemble de nouvelles règles métiers :

- Un client peut avoir un conseiller et un conseiller peut suivre un ou plusieurs clients.
- Un conseiller est caractérisé par : un UUID, un nom, un prénom, un email, une date de naissance, une date d'embauche et une spécialité (assurance, placement, prêt immobilier, crédit à la consommation).

Ci-dessous une nouvelle proposition d'architecture logicielle des classes métier :



Inspirez vous de l'article suivant pour mettre en place la nouvelle architecture :

https://www.jetdev.fr/articles/lheritage-avec-hibernate/

CREDITS

ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
Michel Coulard – Formateur Evry
Chantal Perrachon – IF Neuilly sur Marne

Date de mise à jour : 23/04/2024