Secteur Tertiaire Informatique

Filière « Etude et développement »

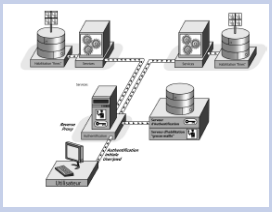
Création d'une API et configuration d'un ORM

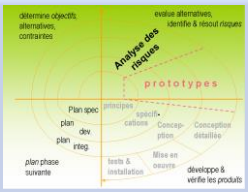
Java – ORM – Spring

Apprentissage

Mise en situation

Evaluation





1. INTRODUCTION

1.1 OBJECTIF DE DEVELOPPEMENT

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

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
- User : représente des utilisateurs
- BankAdvisor : représente des conseiller bancaires

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 avoir un ou plusieurs conseillers de référence. Chaque conseiller peut suivre un ou plusieurs clients. Un utilisateur doit obligatoirement avoir au moins un conseiller.
- Un conseiller à une spécialité parmi {« Assurance », « Placement », « Prêt immobilier », « Crédit consommation » }

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
- Base de données :
    - PostgreSQL

2. MARCHE A SUIVRE

2.1 FORK DU PROJET

Vous pourrez commencer à travailler sur le projet en utilisant la base disponible à l'adresse suivante :

2.2 MISE EN PLACE DE LA BASE DE DONNEES

Démarrer la base de données fournie en utilisant Docker.

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 ».

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 »**

2.3.3 Implémentation du contrôleur Rest

Complétez le code de la classe « **fr.afpa.orm.web.controllers.AccountRestController ».**

2.3.4 Test des endpoints

Testez les endpoints que vous venez d'implémenter.

2.4 RELATIONS MANYTOONE & ONETOMANY

2.4.1 Modification de la base de données

Vous allez mettre en place la structure de base de données suivante :

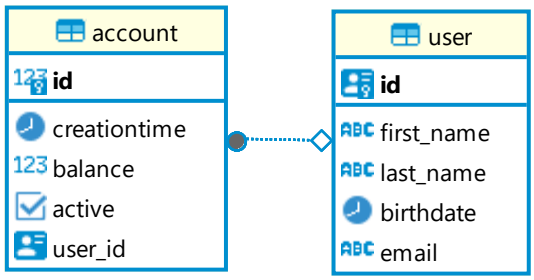


Figure 1 : diagramme ER "Utilisateurs et comptes associés"

Rappel des règles métier :

- un utilisateur peut avoir un ou plusieurs comptes ;

- un compte n’a qu’un seul propriétaire.

L’application de ces règles explique la présence de la clef étrangère « **user_id** » dans la table « **account** ».

Modifiez la structure de la base de données avec votre client SGBD de prédilection pour obtenir quelque chose similaire à Figure 1.

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) est 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 façonner un jeu d’essai exploitable.

2.4.3 Implémentation des entités

Une fois la base de données correctes, vous pourrez implémenter l'entité « User » et modifier « Account ».

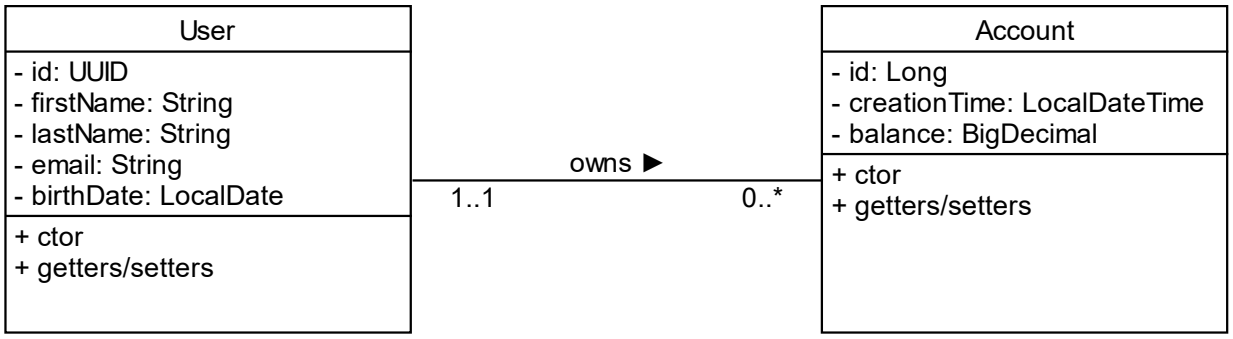
Points de vigilance lors du développement :

- 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 » 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

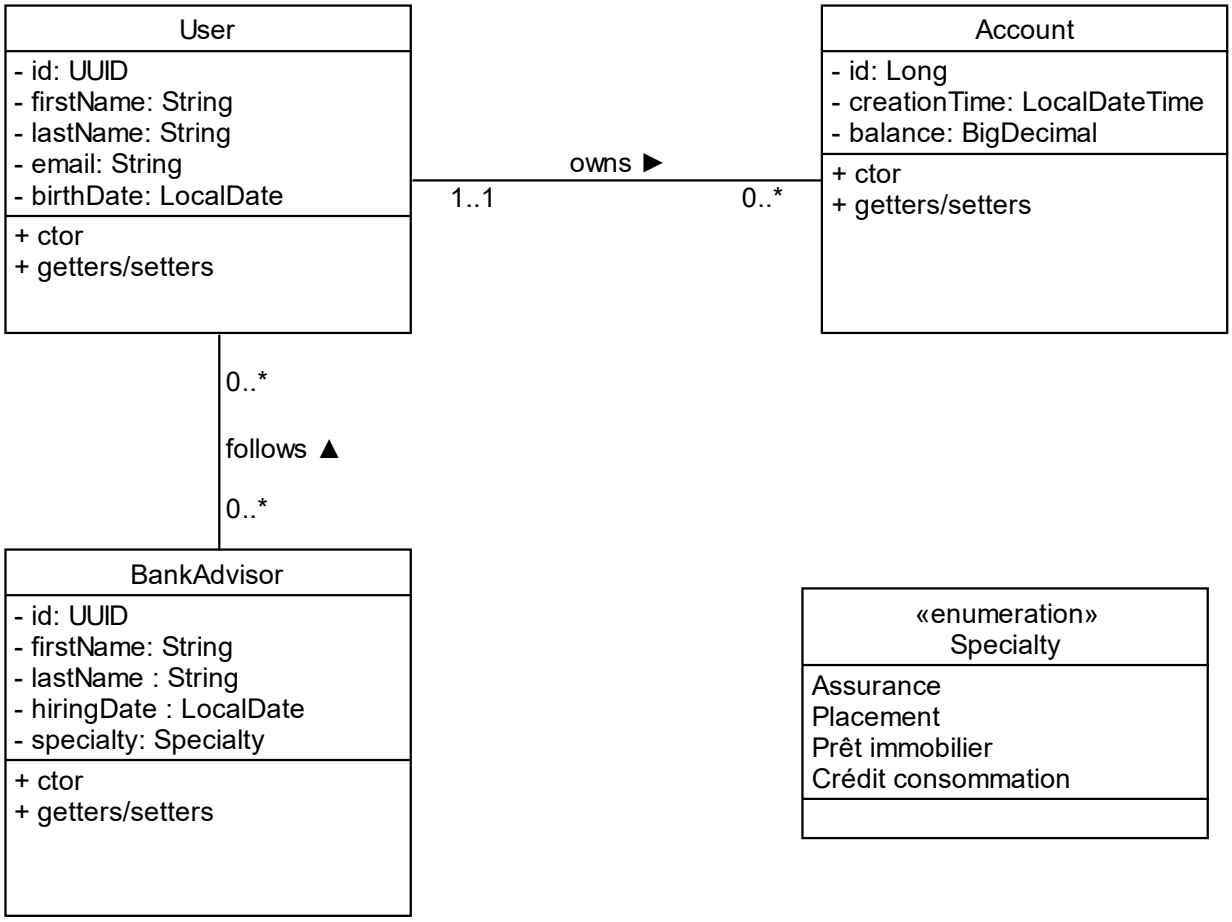
Vous faites face à une récursion infinie liée au mécanisme de sérialisation.

Suivez les recommandations de l'article suivant afin de mettre en place des DTO 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 avoir un ou plusieurs conseillers de référence. Chaque conseiller peut suivre un ou plusieurs clients. Un utilisateur doit obligatoirement avoir au moins un conseiller.
- Plusieurs utilisateurs différents peuvent avoir le même conseiller
- Plusieurs conseillers peuvent suivre plusieurs utilisateurs différents



Vous remarquerez une particularité sur ce diagramme : la présence d'une énumération « Specialty ».

Pour plus d'informations concernant les énumérations en Java : https://www.jmdoudoux.fr/java/dej/chap-enums.htm

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.

Attention

Veillez à bien utiliser un type « enum » pour la colonne « specialty » de votre base de données.

Pour plus d'information sur le type « enum » en PostgreSQL :
https://www.postgresql.org/docs/current/datatype-enum.html

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

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