

Test Technique de Recrutement

L'objectif de ce test technique est de mesurer les capacités du candidat à produire une solution de bonne qualité autour d'un cas d'utilisation.

⚠ Prérequis:

- avoir un environnement de développement avec un IDE
- avoir installé une JDK (de préférence la 16 sinon il faut modifier le POM)
- avoir installé Maven (version 3 ou plus)
- savoir forker un repository Github et le cloner en local (pour cela, il est nécessaire de posséder un compte Github dont la création est gratuite)

⚠ **10 minutes** sont conseillées pour lire **attentivement** le présent document et pour prendre en compte les différents fichiers fournis dans le test.

Déroulement du test

Le test technique est stocké ici : <https://github.com/HiringTechnicalTest/BankAccountTest>

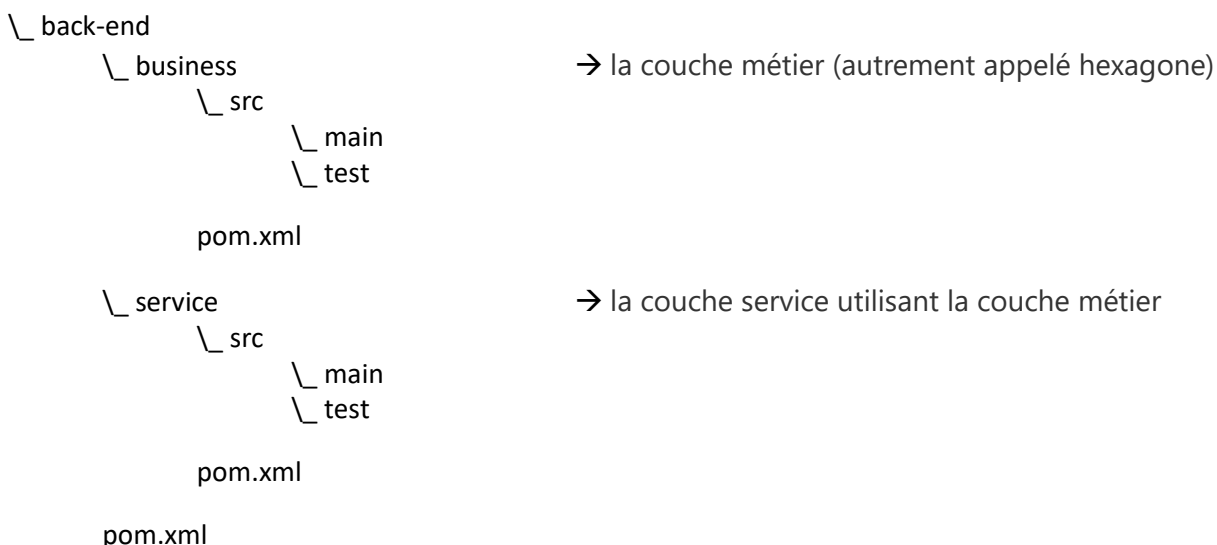
La première chose à faire est de le **forker** sur votre compte Github.

Le test est découpé en 2 étapes qui **doivent** être prises en compte dans **l'ordre** suivant :

1. Développer une logique métier à partir de scénarios fonctionnels
2. Exposer cette logique métier à travers une API RESTful (en Sprint-Boot)

Organisation du projet Java

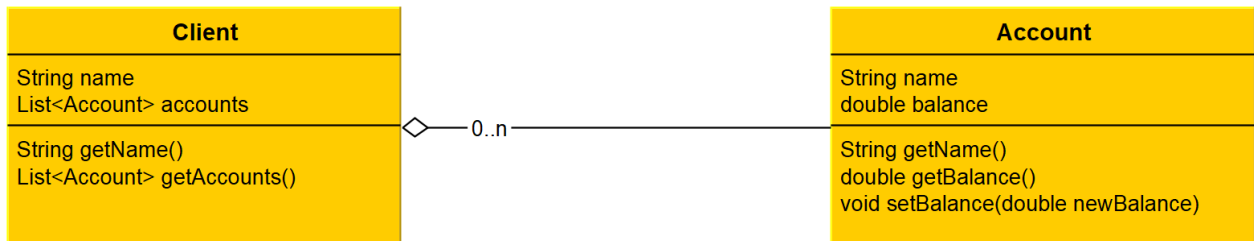
Le présent projet est un projet Maven parent contenant 2 modules :



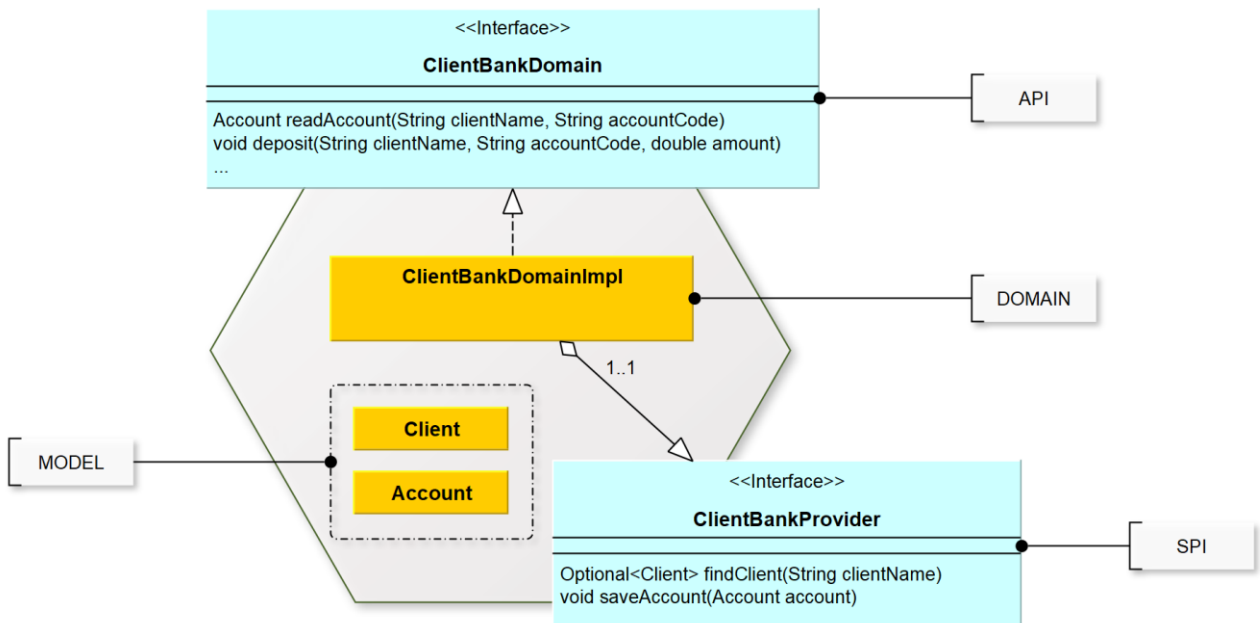
1. Couche « métier »

⚠ Ce module Maven ne doit contenir **aucune dépendance avec Spring, JPA...** (pure Java). Il est important de conserver ce **découplage** car c'est lui qui garantit à terme que le modèle métier est maintenable (architecture hexagonale).

Soit un client qui peut posséder aucun ou plusieurs comptes bancaires.



Votre objectif est d'écrire le modèle métier (hexagone) en répondant à des scénarios fonctionnels :



Scenario: a client should be able to read his accounts

Given I am **steve**
And I own **130.0** on my account **FORTUNEO**
And I own **210.0** on my account **N26**
When I check my account **FORTUNEO**
Then balance of my account **FORTUNEO** should be **130.0**

Scenario: a client should be able to make a deposit on his accounts

Given I am **elon**
And I own **100.0** on my account **BNP**
When I deposit **10.0** on my account **BNP**
And I check my account **BNP**
Then balance of my account **BNP** should be **110.0**

Scenario: a client should be able to make a withdraw from his accounts

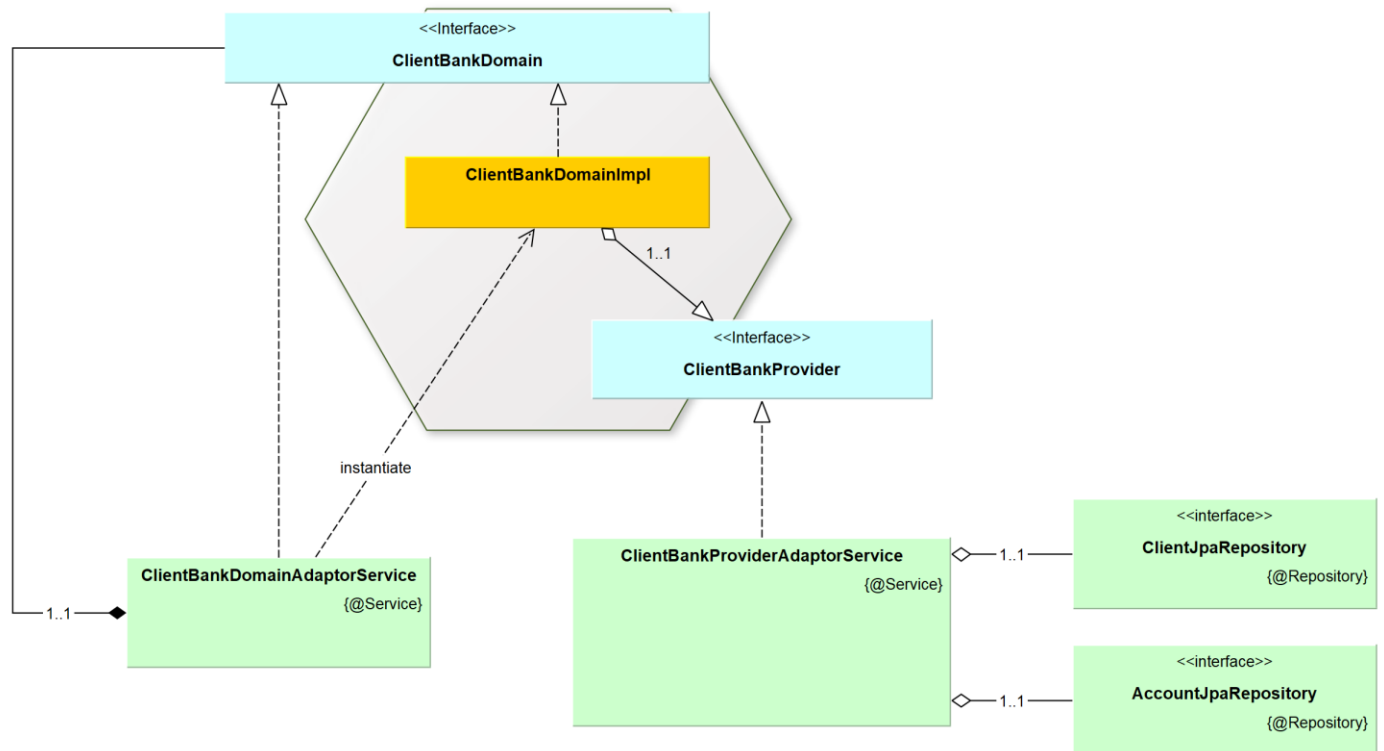
Given I am **jeff**
And I own **30.0** on my account **BFORBANK**
When I withdraw **10.0** on my account **BFORBANK**
And I check my account **BFORBANK**
Then balance of my account **BFORBANK** should be **20.0**

⚠ Une attention toute particulière sera apportée **aux tests unitaires**. L'implémentation des tests Cucumber n'est pas obligatoire. **Mais c'est un plus**. Les dépendances avec Cucumber ont déjà été ajoutées afin de ne pas perdre de temps à configurer.

2. Couche « service »

L'objectif est désormais d'exposer le modèle métier précédent à travers une **API RESTful** développée en **Spring-Boot** en respectant le **découplage des couches métier et service**.

Pour vous aider, voici une partie du diagramme de classe attendu (où Client et Account sont persistés dans un SGBD Relationnel) :



Exemple d'utilisation de l'API RESTful

GET <http://localhost:8080/api/v1/clients/elon/accounts/BNP>

RESPONSE : { "code": "BNP", "balance": 100.0 }

GET <http://localhost:8080/api/v1/clients/steve/accounts>

RESPONSE :

```
[
  { "code": "FORTUNEO", "balance": 130.0 },
  { "code": "N26", "balance": 210.0 }
]
```

PATCH <http://localhost:8080/api/v1/clients/elon/accounts/BNP>

BODY : { "deposit" : 10.0 }

RESPONSE : 110.0

⚠ Une attention toute particulière sera apportée **aux tests unitaires** de ce service.