|  |
| --- |
| MASTER 2 MIAGE IF APPRENTISSAGE 2017-2018 |
| Projet Architectures Microservices |
| Enseignant : Mr Menceur |

|  |
| --- |
| Richelle DJEUKAM YOUDJEU & Tianome RASOLOHERY  30/06/2018 |

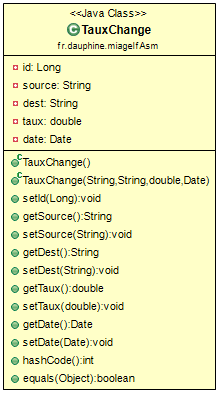
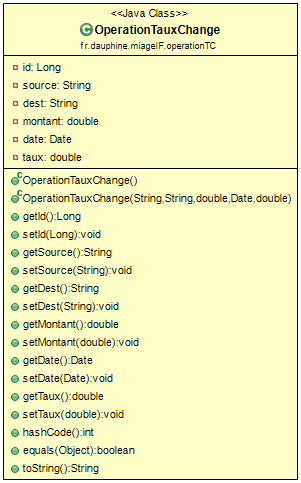


# Introduction

Ce document a pour but d’expliquer le processus de réalisation du projet architecture microservice, les choix qui ont été faits et les difficultés rencontrées lors de son développement. Nous avons essayé d’adopter une méthode de développement itérative et incrémentale en produisant une application fonctionnelle à chaque étape, à laquelle on rajoute des fonctionnalités petit à petit.

# Architecture de l’application

Ci-dessous le diagramme de classes global de nos classes métiers :

Grâce à l’utilisation du Framework Hibernate nos deux classes métiers ci-dessus correspondent aux tables « TAUX\_CHANGE et OPERATION\_TAUX\_CHANGE » dans nos bases données. Par conséquent, la communication avec ces bases données se font de manière transparente avec l’utilisation de l’interface JpaRepository du micro Framework SpringData.

# Fonctionnalités de l’application

Nous avons découpé l’architecture de notre application en deux microservices par rapport aux fonctionnalités mentionnées dans le sujet. L’implémentation de ces dernières a été faite avec le Framework SpringBoot 2.0.3.

## La gestion des taux de change entre deux devises

Nous pouvons observer plusieurs « API REST » à l’intérieur de cette première microservice qui correspondent aux operations CRUD (Create Read Update Delete) sur les taux de changes.

* **@PostMapping("/devise-change")** qui permet de créer un nouveau taux de change entre deux devises. Les informations relatives à cette opération sont communiquées dans le « Body » de la requête elle-même.
* **@GetMapping("/devise-change")** qui permet de récupérer tous les taux de changes existants dans la base de données.
* **@GetMapping("/devise-change/source/{source}/dest/{dest}")** permet de retrouver un taux de change à partir d’une devise source et destination. Par exemple (source = euro et destination = dollar)
* **@PostMapping("/devise-change/{id}")** permet de modifier un taux de change dans la base de données. Elle reçoit en paramètre l’identifiant du taux change dans la base et la nouvelle valeur que l’on voudra renseigner.
* **@DeleteMapping("/devise-change/{id}")** permet la suppression d’un taux de change en renseignant l’identifiant de celui-ci dans l’url.

## La gestion d’une opération de change

La gestion d’une opération de change faite par un utilisateur quelconque se concrétise via l’implémentation des « API REST » suivantes :

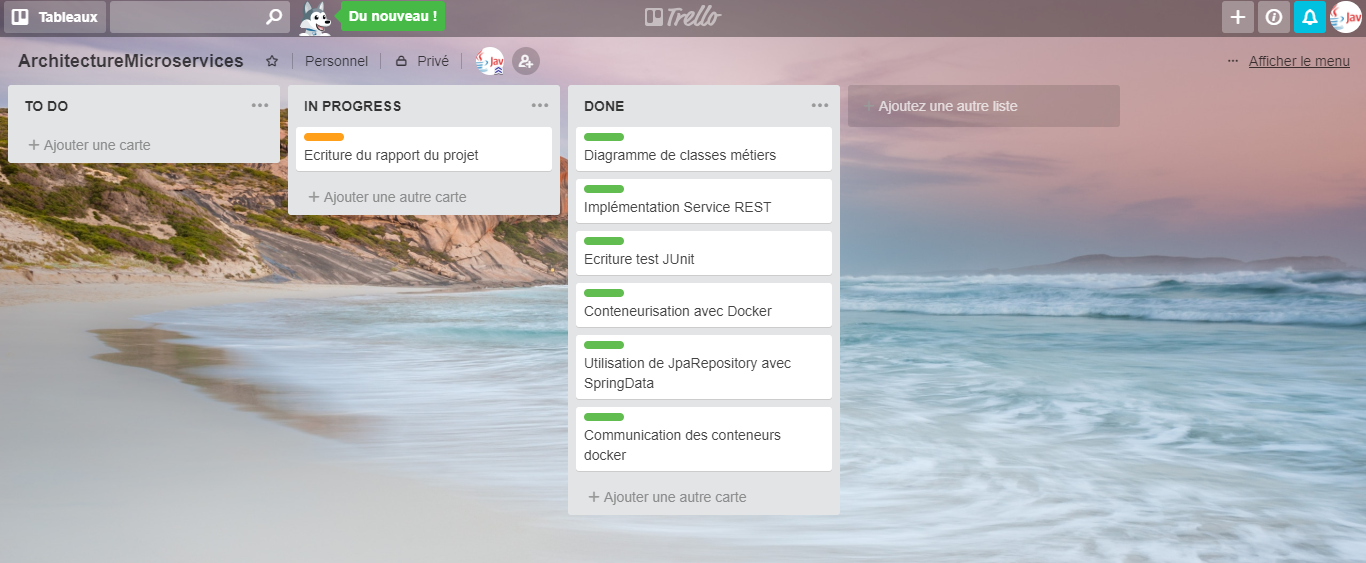
* **@PostMapping("/operation-change")** qui permet à n’importe quel utilisateur de créer un nouvel échange de devise en renseignant en paramètre dans le corps de la requête les devises sources et destination, le montant qu’il veut échanger. Cependant, le taux de change entre les deux devises est récupéré directement du microservice « gestion des taux de changes » pour normaliser toutes les transactions.
* **@GetMapping("/operation-change")** permet de récupérer toutes les opérations de change existantes dans la base de données.
* **@GetMapping("/operation-change/{id}")** permet de retrouver une opération de change en renseignant son identifiant dans l’url de la requête.
* **@PostMapping("/operation-change/{id}")** permet de mettre à jour une opération existante. Elle reçoit en paramètre l’identifiant de l’opération de change dans la base et la nouvelle valeur que l’on voudra renseigner.
* **@DeleteMapping("/operation-change/{id}")** permet la suppression d’une opération de change en renseignant l’identifiant de celui-ci dans l’url.

Etant donné qu’on a mis en place une architecture orientée ressource, nous avons donc décider d’implémenter l’ensemble de nos microservices avec comme formalisme d’échange de données le format JSON (JavaScript Object Notation).

On a aussi mis en place la conteneurisation de nos microservices grâce à l’utilisation du logiciel « Docker ». En effet, pour chaque microservice on a utilisé le plugin Maven de Spotify « docker-maven-plugin » pour construire et pousser des images Docker.

# Répartition du travail

Nous nous sommes organisés à l’aide de Trello. Nous avons créé des cartes pour chacune des tâches principales à réaliser et nous nous les sommes répartis dans un premier temps.



Par la suite, nous avons revu et modifié ensemble les fonctionnalités créées. La répartition finale a été globalement la suivante :

• **Tianome Rasolohery** : définition et implémentation des classes métiers, implémentation des API REST, analyse statique du code avec Sonarlint.

• **Richelle Djeukam** : utilisation du plugin Docker pour la conteneurisation des microservices, écriture des tests unitaires pour tester le fonctionnement des API REST.

# Réflexions, difficultés et pistes abandonnées

Les principales difficultés de ce projet ont été la conteneurisation des microservices et la communication entre les conteneurs. La gestion de l’ajout des opérations de change dans la base de données a fait l’objet de plusieurs refontes et a exposé de multiples problématiques, notamment en ce qui concerne la communication entre deux services REST. Cette difficulté a été résolue grâce à la documentation officielle de [SpringBoot](https://spring.io/projects/spring-boot) et aux différents forums notamment [StackOverflow](https://stackoverflow.com/).

Après avoir expérimenté l’utilisation de Docker sur le système d’exploitation Windows 10, nous avons conclu qu’utiliser ce logiciel sur une plateforme Linux serait la solution la plus adaptée au problème de communication des conteneurs.

En ce qui concerne le traitement des réponses http lors de la récupération du taux de change dans le microservice opération taux de change, nous avons utilisé la classe JSONObject du ce [repository](https://mvnrepository.com/artifact/org.json/json/20180130) qui répondait parfaitement à nos besoins.

Nous avions également pensé à l’utilisation de la méthode HTTP PUT présent dans les normes REST pour la mise à jour des opérations de change et des taux de changes, mais cela posait le problème de « Firewall » avec notre environnement de développement. Par conséquent, nous avons utilisé la méthode HTTP POST qui permet aussi de créer une nouvelle ressource ou de remplacer une représentation de la ressource ciblée par le contenu de la requête.

# Bilan

Ce projet nous a fait prendre conscience de la difficulté de découper un projet informatique en microservices, même à la simple échelle de deux développeurs travaillant très proches géographiquement et sans décalage horaire. Le respect des jalons fixés en cours s’est révélé assez difficile et le raffinement du code après la réalisation des fonctionnalités principales a pris plus de temps que prévu.

Hormis ces challenges d’ordre organisationnels, l’utilisation de technologies externes à Java a également posé ses propres difficultés (notamment pour la communication entre les deux conteneurs Docker). Malgré les accrocs rencontrés, nous sommes tout compte fait parvenus à développer des microservices fonctionnelles et utilisables. Nous en ressortons ainsi plus expérimentés et plus compétents.