



**UNIVERSITÉ
DE LORRAINE**



Application multi-services

Conception et développement d'une application
multi-services type Révolute

Table des matières

1 - Contexte	2
2 - Conception	2
2.1 - Architecture de l'application	2
2.1.1 - Banque service	3
2.1.2 - Marchand service	3
2.1.3 - Conversion service	3
2.1.4 - Localisation service	4
3 - Implémentation	5
3.1 - API RESTful	5
3.2 - Sécurité	9
3.3 - Conteneurisation	10
3.4 - Tests	11
3.5 - Services	11
4 - Conclusion	12

1 - Contexte

Ce projet consiste à réaliser une application semblable à la banque en ligne Révolute. Il doit répondre à plusieurs contraintes techniques abordées en cours. Le projet doit proposer une API RESTful utilisable par les usagers de l'application afin de gérer leurs données (cartes, comptes, opérations). De plus, l'outil doit être constitué de plusieurs services répondant à des besoins spécifiques. L'application doit permettre à un grand nombre d'utilisateurs de l'utiliser.

2 - Conception

2.1 - Architecture de l'application

Voici la vue d'ensemble que j'ai réalisée afin de décrire toutes les composantes du projet. Ce schéma n'explique pas ce que j'ai fait mais, ce que j'envisage de développer.

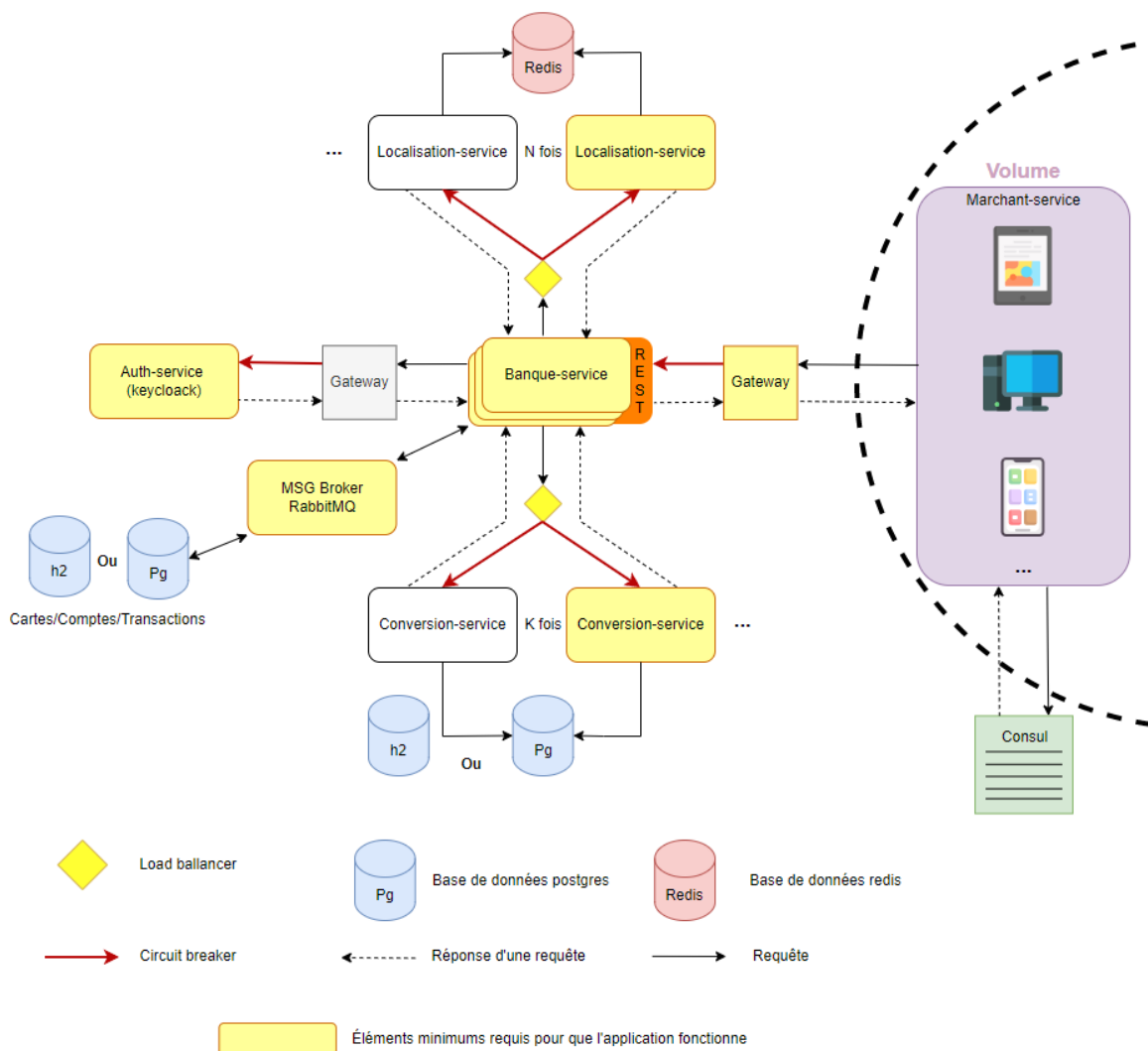


Figure 1 : architecture conceptuelle du projet

2.1.1 - Banque service

Ce service est le cœur de l'application, il permet de gérer les comptes, les cartes et les opérations des clients. Il est constitué d'une **API RESTful** donnant accès aux différentes ressources citées précédemment. Les données sont stockées dans une base de données postgresql. À noter qu'il est tout à fait possible de mettre les données dans une H2 ou une autre base donnée tel que mysql par exemple.

2.1.2 - Marchand service

Marchand service sert à simuler un terminal de paiement électronique chez un commerçant. C'est par ce service que les utilisateurs peuvent effectuer un paiement en utilisant leur carte bancaire. Afin de permettre la **scalabilité horizontale**, une gateway est placée entre ce service et celui de la banque. Ainsi, en ajoutant consul, cela nous permet de créer N banque service et lorsqu'une requête est exécutée sur un service marchand, elle est redirigée vers l'un des différents banques services disponibles à l'instant T, via un **load balancer**. De plus, cette construction permet aux utilisateurs finaux de toujours pouvoir payer, car si un service banque n'est plus disponible, d'autres prendront le relai grâce aux **circuit breaker**.

2.1.3 - Conversion service

Notre banque propose la possibilité de payer à l'étranger. Cependant, certains n'ont pas la même devise et une conversion est nécessaire. Ce service s'occupe principalement de gérer cette fonctionnalité. Lorsqu'une transaction est effectuée et que le pays du client n'est pas le même que celui du paiement, ce service va déterminer le taux de change à appliquer pour que le prix soit respecté. Il va ensuite le transmettre à la banque pour compléter la transaction. Pour fonctionner, le service a accès à une base h2 ou postgres contenant les taux de change entre les devises. Également, le service est conçu pour suivre la **scalabilité horizontale** du service banque pour éviter de le ralentir lorsque ce dernier aura un très grand nombre d'instances.

2.1.4 - Localisation service

Une fonctionnalité de notre banque est de permettre de vérifier le périmètre de réalisation d'un paiement. Un utilisateur peut renseigner une distance de validité pour une carte qui sera vérifiée lorsqu'un paiement est réalisé. L'idée est de limiter les risques en cas de vol de carte d'un de nos clients. Même chose que pour les systèmes précédents, le service est conçu de sorte à pouvoir suivre la **scalabilité horizontale** du service banque.

2.1.5 - Message broker

Sachant que nous proposons un service de paiement, il est obligatoire de **garantir l'intégrité des données**, surtout pour les opérations. Pour répondre à cette problématique, un message broker est placé entre le service banque et la base de données. Si nous devons réellement développer une infrastructure qui répond à 100% à ces critères, il faudrait de la réplication de base de données utilisant les principes NRW où $N = 3$, $R = 1$ et $W = 2$ ou 3 car nous souhaitons être certains que les données enregistrées soient bonnes.

3 - Implémentation

En me basant sur la conception, voici ce que j'ai réussi à mettre en place. Les éléments manquants sont le module de localisation ainsi que le message broker. Je n'ai pas développé ces éléments par manque de temps et d'énergie.

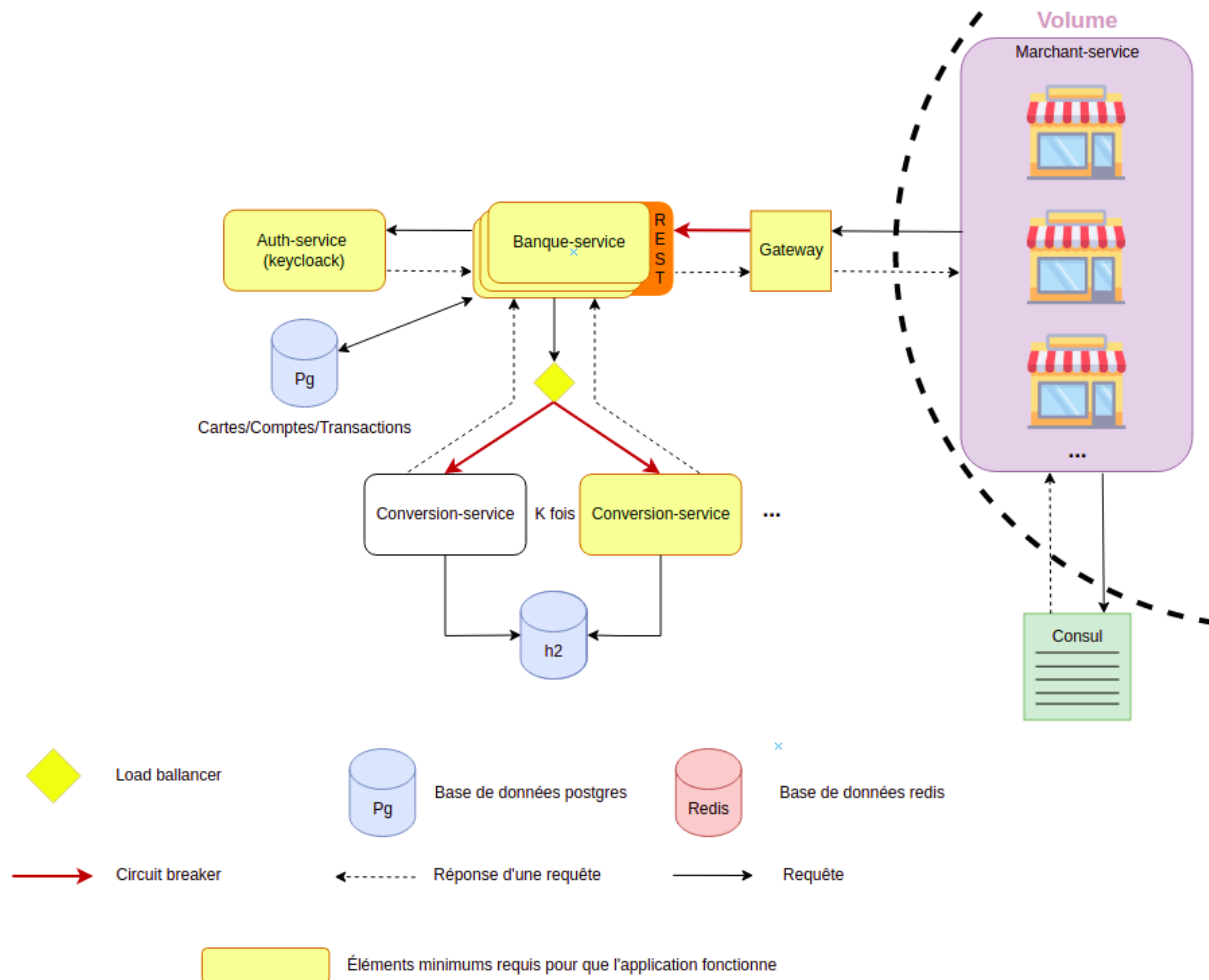


Figure 2 : architecture implémentée du projet

3.1 - API RESTful

Le cœur du service banque est constitué d'une API RESTful afin de pouvoir accéder aux différentes ressources. Pour cette partie, nous allons lister les différentes routes disponibles par l'api. Il faut savoir qu'il est possible de réaliser des requêtes directement sur un service banque, si j'avais eu plus de temps, j'aurais aimé que toutes les requêtes passent par une gateway afin d'encapsuler les services pour augmenter le niveau de sécurité.

Voici les différentes routes disponible sur l'API banque :

- **POST accounts** ⇒ En tant qu'utilisateur, je souhaite créer un compte
- **GET accounts/{accountId}** ⇒ En tant qu'utilisateur, je souhaite consulter mon compte
- **PATCH accounts/{accountId}** ⇒ En tant qu'utilisateur, je souhaite modifier mon compte
- **GET accounts/{accountId}/balance** ⇒ En tant qu'utilisateur, je souhaite consulter le solde de mon compte
- **GET accounts/{accountId}/cards** ⇒ En tant qu'utilisateur, je souhaite listes mes cartes
- **GET accounts/{accountId}/cards/{cardId}** ⇒ En tant qu'utilisateur, je souhaite consulter les informations de ma carte
- **POST accounts/{accountId}/cards** ⇒ En tant qu'utilisateur, je souhaite créer une carte pour mon compte
- **PATCH accounts/{accountId}/cards** ⇒ En tant qu'utilisateur, je souhaite changer les informations de ma carte (plafond, bloqué, code, localisation)
- **GET accounts/{accountId}/operations** ⇒ En tant qu'utilisateur, je souhaite consulter mes opérations
- **GET accounts/{accountId}/operations/{operationId}** ⇒ En tant qu'utilisateur, je souhaite consulter une opérations en particulier
- **POST /pay** En tant qu'utilisateur, je souhaite payer mon achat dans une boutique
- **POST /accounts/{accountId}/operations** ⇒ En tant qu'utilisateur, je souhaite réaliser un virement

Remarque : étant une banque très généreuse, j'offre 1000€ à la création d'un compte.

Pour les routes nécessitant des données en entrée (POST, PATCH) voici les différents schémas listant les paramètres que l'utilisateur pourra fournir. Pour plus de détails voir documentation swagger.

CarteInput	CompteInput	OperationInput
+ code : int	+ nom : String	+ numeroCarte : String
+ bloqué : boolean	+ prenom : String	+ code : String
+ plafond : Double	+ pays: String	+ crypto : String
+ contact : boolean	+ passeport : String	+ pays : String
+ longitude : double	+ tel : String	+ montant : Double
+ latitude : double	+ secret : String	+ devise : String

Figure 3 : body des entités pour les requêtes PATCH et POST

Cela nous donne le diagramme de classe représentant le contenu de toutes les entités présentes sur le service banque. Dans l'entité carte, la longitude et la latitude permet de d'indiquer le centre du cercle définissant le rayon d'utilisabilité de la carte par l'utilisateur. Malheureusement, je n'ai pas traité cet aspect du sujet, donc ces deux attributs sont inutiles pour le moment.

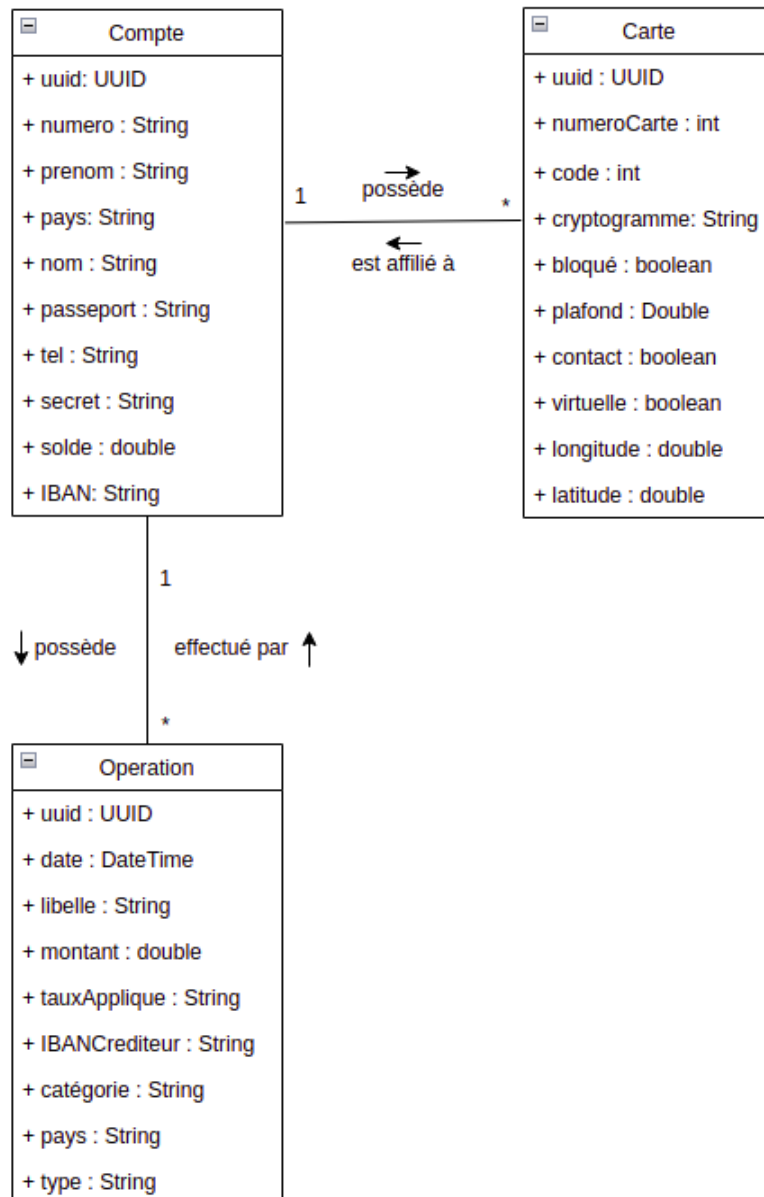


Figure 4 : diagramme de classe des entités de banque-service

Pour la partie HATEOAS, j'ai choisi d'ajouter les liens vers les différentes ressources d'un compte telles que les opérations, les cartes et le montant d'argent restant sur le compte de l'utilisateur. Voici un exemple, ci-dessous, de compte obtenu avec la requête **GET accounts/{accountId}**.

```
{
  "iban": "FR2588403613086558450600V58",
  "accountNumber": "6558450600V",
  "name": "ahaha",
  "surname": "UwW",
  "birthday": "27-07-1999",
  "country": "France",
  "passport": "123456789",
  "tel": "+0033636790462",
  "secret": "7a53028a-765a-11ec-90d6-0242ac120003",
  "balance": 1000.00,
  "_links": {
    "self": {
      "href": "http://localhost:8082/accounts/9cab10cc-5ae4-443c-96fb-9bd09202ec72/"
    },
    "balance": {
      "href": "http://localhost:8082/accounts/9cab10cc-5ae4-443c-96fb-9bd09202ec72/balance"
    },
    "cards": {
      "href": "http://localhost:8082/accounts/9cab10cc-5ae4-443c-96fb-9bd09202ec72/cards"
    },
    "operations": {
      "href": "http://localhost:8082/accounts/9cab10cc-5ae4-443c-96fb-9bd09202ec72/operations"
    }
  }
}
```

Figure 5 : exemple de json obtenu suite à un GET sur compte

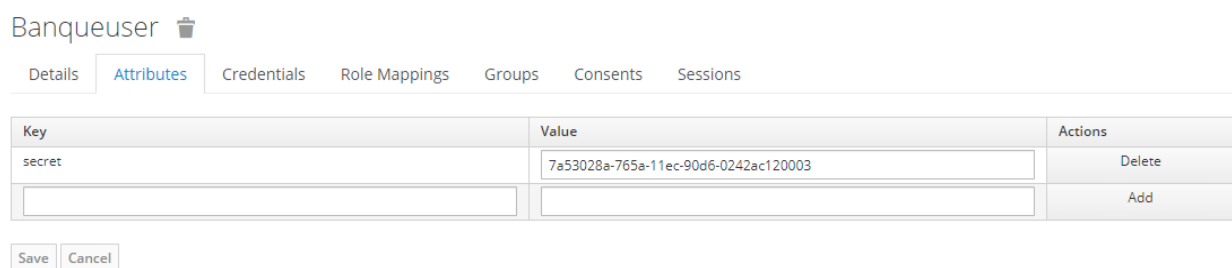
En ce qui concerne les autres ressources (opérations et cartes), je les ai simplement reliées à leur compte.

3.2 - Sécurité

Cette partie m'a donné énormément de difficultés, j'ai mis beaucoup de temps à faire fonctionner les outils et les intégrer à mon projet. Afin de répondre aux besoins spécifiés dans le sujet, deux problématiques se posaient :

- Comment protéger l'accès aux différentes ressources ?
- Comment créer le lien d'appartenance entre un usager et son compte ?

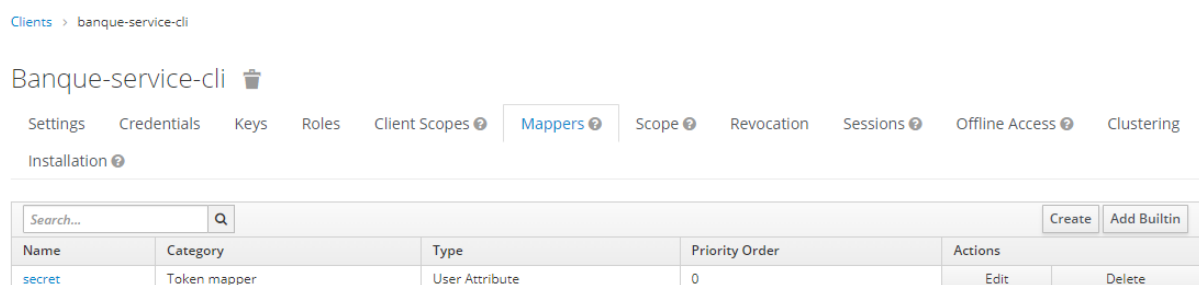
Pour répondre à la première question, j'ai choisi d'utiliser un service à côté, **Keycloak**, qui s'occupe de gérer les comptes utilisateurs. Pour la seconde, dans un premier temps, j'ai choisi de faire le lien entre le compte utilisateur et le compte bancaire à travers le nom d'utilisateur keycloak. Cependant, je ne trouvais pas ça très propre donc, j'ai cherché une autre solution. Ainsi, en explorant keycloak j'ai observé qu'il était possible de spécifier des attributs personnalisés pour un utilisateur. De ce fait, j'ai ajouté un attribut **"secret"** à mon utilisateur keycloak qui doit concorder avec l'attribut **"secret"** de l'entité **Compte** de banque-service.



Key	Value	Actions
secret	7a53028a-765a-11ec-90d6-0242ac120003	Delete
<input type="text"/>	<input type="text"/>	Add

Figure 6 : ajout du secret pour un utilisateur keycloak

Ensuite, pour que le secret soit transmis au service banque, il faut qu'il figure dans le token d'accès. Pour ce faire, j'ai créé un mapping dans mon client keycloak. Il s'occupe d'injecter l'attribut secret dans l'accès token.



Clients > banque-service-cli

Banque-service-cli

Settings Credentials Keys Roles Client Scopes Mappers Scope Revocation Sessions Offline Access Clustering Installation

Name	Category	Type	Priority Order	Actions
secret	Token mapper	User Attribute	0	Edit Delete

Figure 7 : injection du secret dans l'accès token

Enfin, lorsqu'une requête est exécutée sur un service banque, la classe **AccountMatcher** s'occupe de vérifier à partir d'un compte et des informations d'authentification l'appartenance à un compte bancaire.

Pour ce projet, j'ai réalisé une sécurité simple dans laquelle je vérifie l'appartenance à un compte d'un usager où on fournit à la main le token dans la requête. Pour la partie paiement, c'est open bar, ce qui n'est pas très bien. Avec d'avantage de temps, j'aurais essayé de sécuriser cette route qui est appelée par un le service marchand. Ma première idée aurait été de créer une clé d'accès enregistrée dans le service pour pouvoir y accéder. De ce fait, personne n'aurait pu se faire passer pour un service marchand afin de payer à la place d'un autre usager.

The screenshot shows a REST client interface with a POST request to `http://localhost:8180/auth/realms/BanqueService/protocol/openid-connect/token`. The request body is set to `x-www-form-urlencoded` and contains the following parameters:

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	password	123456			
<input checked="" type="checkbox"/>	username	banqueuser			
<input checked="" type="checkbox"/>	grant_type	password			
<input checked="" type="checkbox"/>	client_id	banque-service-cli			
<input checked="" type="checkbox"/>	client_secret	594a256a-344a-47f5-8eea-b1dcba1...			
	Key	Value	Description		

Figure 8 : exemple de requête que je réalise pour récupérer l'accès token

Ressource : <https://www.baeldung.com/keycloak-custom-user-attributes>

3.3 - Conteneurisation

Un objectif que je m'étais donné pour ce projet, était de réaliser un projet "clé en main" grâce à docker. Pour y parvenir, j'utilise un fichier docker-compose à la racine de mon projet qui s'occupe de build tous mes services. Cependant, je n'ai pas réussi à rendre cela complètement fonctionnelle. Donc mon docker-compose contient uniquement les éléments clé de mon application tels que keycloak, la base de données postgres du service banque et la base de données keycloak. Si vous le souhaitez, vous pouvez tout de même consulter la version complète presque fonctionnelle : **docker-compose-full.yml** sur github.

3.4 - Tests

J'ai testé la majorité de routes présentes dans le service banque. La seule que je n'ai pas réussi à tester et celle qui s'occupe de faire un paiement. La raison est que j'ai des difficultés à **mock** un service externe et par conséquent, je n'ai pas réussi à faire quelque chose de propre. De plus, la moitié de mes tests ne fonctionnent pas suite à l'introduction de la partie sécurité. C'est pour cette raison que je vous rends une version avec sécurité et sans sécurité. Sur git, cela correspond à la branche master pour le tout avec sécurité et la branche "without-securiry" pour la partie sans sécurité.

3.5 - Services

Les services que j'ai implémentés sont : banque, conversion et marchand.

Conversion service :

- **GET** /conversion-devise/source/{source}/target/{target}/amount/{amout}

Ce service permet principalement de convertir 2 monnaies ayant des devises différentes. Il y a une base de donnée H2 qui contient les données de conversion suivante :

- **EUR vers USD** et son inverse
- **EUR vers CHF** et son inverse
- **EUR vers GBP** et son inverse

Je n'en ai pas mis plus parce que je ne voulais pas perdre de temps à enrichir ces données.

Marchand service :

- **POST** /pay

Pour pouvoir payer avec votre carte bancaire, il vous faut : Le numéro de la carte, le cryptogramme, le code, le montant de la transaction, le numéro IBAN du compte créditeur, le pays et la devise de la monnaie demandée. A noter que seul l'USD, le CHF et la GPB sont acceptés pour cette application.

Gateway-banque-service ⇒ ce service permet de faire le lien entre le service marchand et le service banque et ainsi, il permet la scalabilité du service banque.

Gateway-conversion-service ⇒ ce service permet de faire le lien entre le service banque et le service conversion et ainsi, il permet la scalabilité du service conversion.

4 - Conduite de projet

Pour ce projet, j'ai choisi de le réaliser en mode agile, cela m'a permis de réaliser beaucoup de fonctionnalités sans être pressé par le chrono à la fin même si je n'ai pas tout fait.

Aa Tache	# Nombre d'heures
Analyse du sujet	1
Conception globale du projet	3
Plannification du projet	1
Conception API RESTful	2
Réflexion structure du projet	3
En tant qu'utilisateur, je souhaite consulter mon compte	2
En tant qu'utilisateur je souhaite créer un compte	2
En tant qu'utilisateur je souhaite mettre à jour mon compte	2
M2 → PgsqI	0.5
En tant qu'utilisateur, je souhaite listes mes cartes	3
En tant qu'utilisateur, je souhaite consulter les informations de ma	0.5
En tant qu'utilisateur, je souhaite créer une carte pour mon compte	1
En tant qu'utilisateur, je souhaite éditer ma carte	2
tests cartes	3
En tant qu'utilisateur, je souhaite consulter mes opérations	0.5
En tant qu'utilisateur, je souhaite consulter une opérations en parti	0.5
setup Docker	2.5
documentation	1
keycloak	5
Commerçant-service	2
Gateway	1
Banque-service payer	1
Rapport	1
Conversion-service	4
Refactoring	3
Faire fonctionner la sécurité	5
Rapport final	4
+ New	
VALUES 27	
SUM 56.5	

Figure 9 : temps de travail

5 - Conclusion

Pour ce projet, j'ai réalisé une **API RESTful** gérant des comptes bancaires, des cartes et des opérations. J'ai géré les aspects **multi-service** avec l'intégration de **consul** et des gateways permettant de faire du **load balancing**. J'ai également pris en compte les aspects du **fail over**, car nous savons qu'il va y avoir des crash et ainsi puisque mon application peut démarrer N banque service et M conversion service (**scalabilité**) lorsqu'une erreur se produit le **Circuit Breaker** et le retry permettent tout de même de fournir une réponse à l'utilisateur. J'ai également pris en compte les aspects **HATEOAS** avec les différents liens ajoutés à mes entités. Enfin, avec quelques difficultés, j'ai réussi à sécuriser mon application avec keylock (cela m'a coûté mes tests, mais ce n'est pas grave).

Je n'ai pas réussi à traiter tout le sujet. Typiquement, je ne traite pas les aspects de localisation pour les paiements. Je n'ai pas traité les catégorisation des opérations lorsqu'un paiement est réalisé en boutique. Je n'ai pas traité les aspects liés aux virements entre comptes bancaires. Je n'ai pas testé tout ce que je voulais sur mon projet. Enfin, je n'ai pas traité la partie message broker qui aurait permis de rendre l'enregistrement des données beaucoup plus fiable, car actuellement, si je kill la base de données au mauvais moment, il est possible de perdre une transaction ce qui n'est pas acceptable pour une banque.

Ce projet a été très riche en technicité et je suis déçu de ne pas pouvoir le continuer par manque de temps, j'aurais bien aimé le passer en tant que projet principal des SID en retirant les projets des autres enseignants et en les remplaçant par des modules pour ce projet.