

# Projet API REST

Laurene CLADT



# Table des matières

<b>Introduction</b>	<b>3</b>
<b>Conception</b>	<b>3</b>
<b>Réalisation</b>	<b>7</b>
<b>Réponse aux besoins</b>	<b>9</b>
Utilisation de l'API	10
Données et test de l'application	11
<b>Conclusion</b>	<b>12</b>
<b>Annexes</b>	<b>13</b>
GitHub	13
Architecture WOA	13
Sources	14

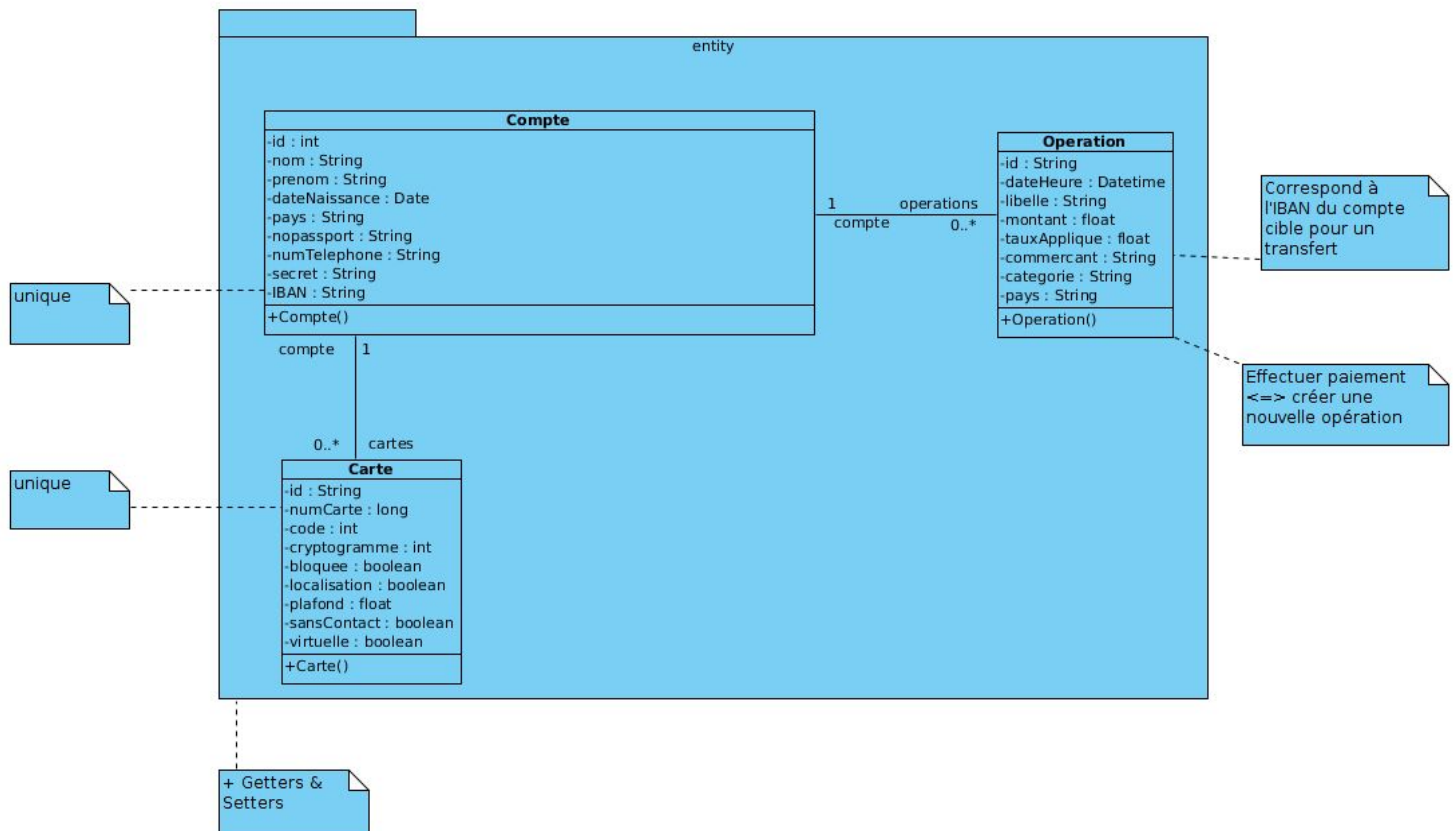
# Introduction

L'objectif de ce projet était de réaliser une API de gestion de comptes bancaires, en implémentant un certain nombre d'attributs et de routes définies. Cette API devait être développée en utilisant la technologie vue en cours (Spring Boot), mettre en pratique HATEOAS et prendre en compte l'aspect sécurité de l'application (authentification et autorisation).

## Conception

La première partie nécessaire pour mener à bien la réalisation de ce projet était la conception de l'API.

La première étape a été d'étudier le sujet et de définir les entités et routes constituant le coeur de l'application. Cette étape a été plutôt rapide, car le sujet était assez détaillé sur les spécifications. Au niveau des entités, j'ai tout d'abord réalisé un diagramme de classes afin de faciliter l'étape de passage au code. La conception finale de l'API diffère légèrement de ce diagramme, car les types définis en amont rendaient difficile ou peu pratique la réalisation des classes Java. Par exemple, la date d'une opération est maintenant de type "String", tandis que le montant est de type "BigDecimal".

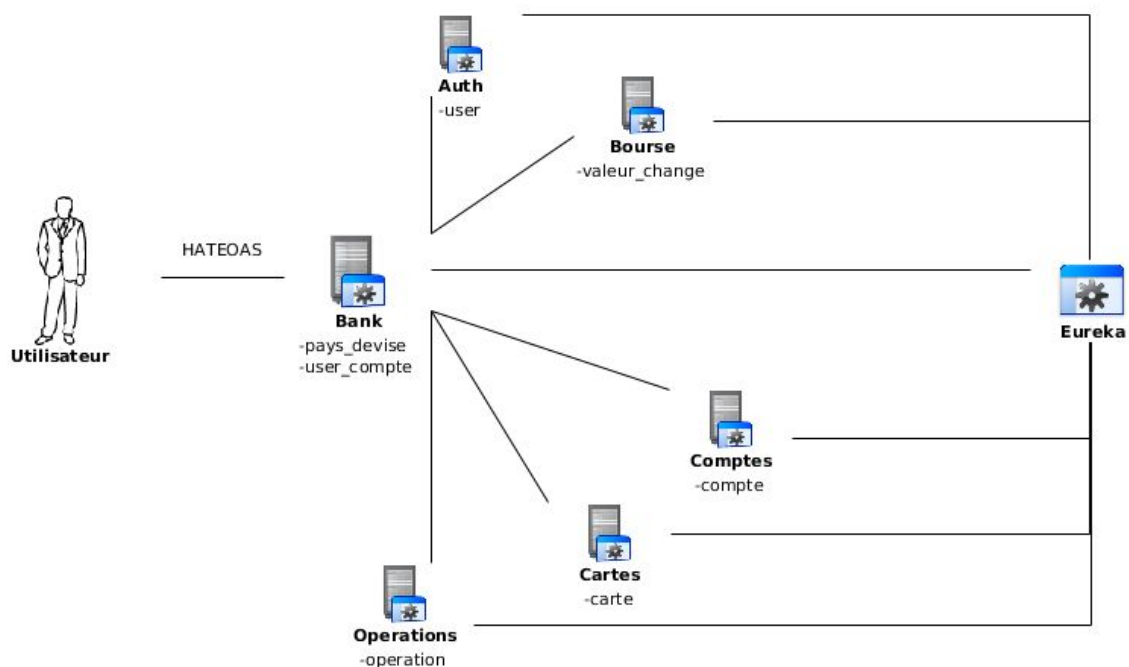


Pour les IDs des objets, j'ai tout d'abord pensé à utiliser l'IBAN d'un compte et le numéro d'une carte en tant qu'ID, car ils sont forcément uniques. Au final, j'ai décidé d'ajouter un "UUID" généré aléatoirement afin d'ajouter une sécurité supplémentaire sur l'API. Cette décision m'a néanmoins posé quelques problèmes par la suite, notamment lors de la création d'un transfert, où un compte doit être récupéré grâce à son IBAN.

Au niveau des routes et après réflexion, j'ai décidé d'implémenter uniquement celles présentes dans le sujet. J'ai tout d'abord pensé à ajouter des routes PATCH pour les opérations disponibles sur la carte : bloquer, mettre à jour le plafond, activer/désactiver le sans contact et la localisation. Bien que ces routes étaient simples à implémenter, j'ai finalement décidé de ne pas les inclure dans l'API, car la route PUT devait être impérativement implémentée et que les routes PATCH supplémentaires auraient été redondantes selon ma vision des choses.

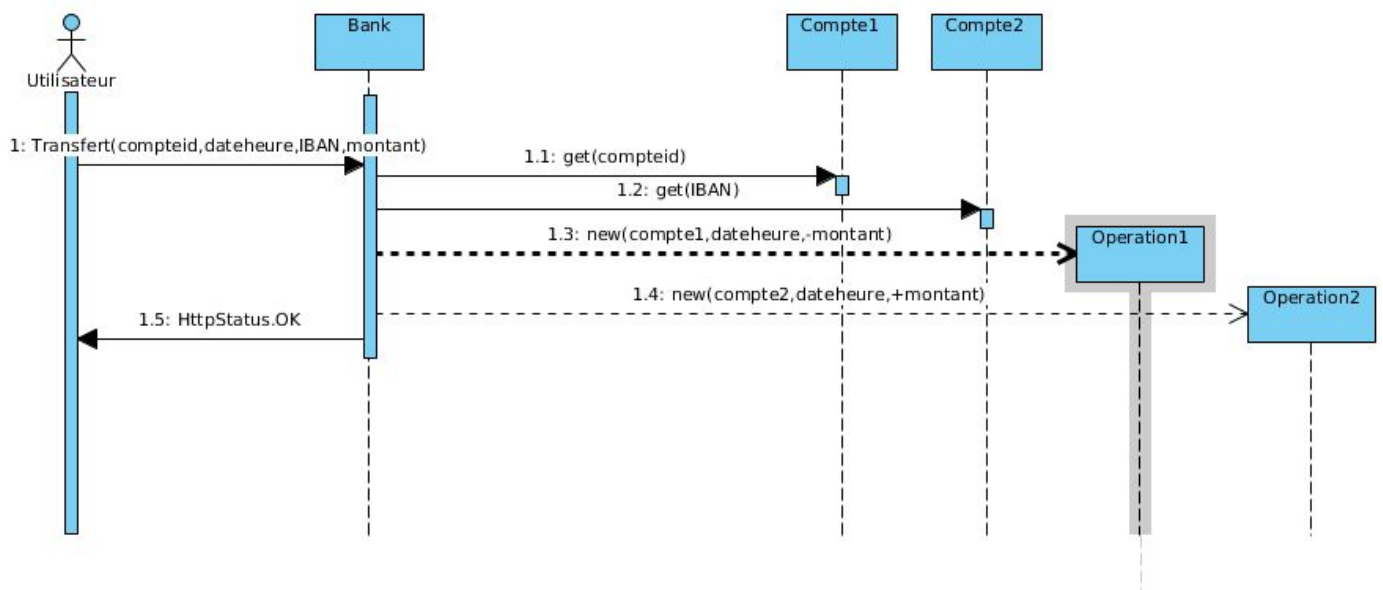
Deux routes supplémentaires en rapport avec l'authentification ont néanmoins été ajoutées : POST /sign-up et POST /login, respectivement l'enregistrement et la connexion d'un utilisateur.

Afin d'utiliser les technologies vues en cours et d'éviter de créer une application "monolithe", j'ai d'abord défini l'ensemble des micro-services dont l'application serait constituée. Pour cela, je me suis inspirée de ce qui avait été fait en CM avec le service "Eureka" et celui de conversion de devise. Je me suis également basée sur le principe de l'architecture "WOA" (Web Oriented Application) (voir annexe) : l'application sera constituée d'un service principal "Bank" (servant de "gateway") et de plusieurs micro-services, chacun représentant une entité de l'API (comptes, cartes et opérations). Les services Eureka (pour assurer le "load-balancing" et mettre les micro-services en relation), authentification et bourse (conversion de devises en fonction du taux associé) font également partie de la structure de l'API. Au niveau de la persistance des données, chaque entité est sauvegardée dans une base de données H2 appartenant à son service respectif.



*Conception initiale de l'application*

Certaines requêtes, tel que le transfert d'un compte à un autre ou la conversion de devises lors d'une opération à l'étranger, ont été plus complexes à implémenter. Un diagramme de séquence était alors nécessaire pour visualiser ce qu'il devait se passer dans l'API à ce moment là, notamment au niveau des services.



*Diagramme de séquence simplifié pour la fonction "Transfert"*

Pour l'implémentation des liens HATEOAS, cela se fait sur la couche "gateway" de l'API, c'est-à-dire le service "Bank". L'API récupère l'URL actuelle du service ainsi que les routes correspondantes aux ressources pertinentes à chaque requête HTTP. Par exemple, lors de l'appel d'une ressource "compte" on peut voir le chemin vers ses cartes ou opérations, tandis que lors de l'appel d'une ressource "carte" on voit le lien vers la liste parente et vers son compte associé. A la création d'une nouvelle ressource via une requête POST, le lien vers cette ressource est également ajouté au "header" de la réponse. Ces liens permettent une navigation facile pour tout utilisateur de l'API. Les codes HTTP renvoyés ont également été gérés : 201 quand une ressource est créée, 204 lorsqu'elle est supprimée, 404 quand elle n'existe pas, et ainsi de suite.

```
{
  "id": "2a55",
  "nom": "Cat",
  "prenom": "Luna",
  "datenaissance": "29/03/2017",
  "pays": "FR",
  "nopassport": "264242448",
  "numtel": "0033712345678",
  "secret": "secretlulu",
  "iban": "738035816301840864711953934",
  "_links": {
    "self": {
      "href": "http://localhost:8080/comptes/2a55"
    },
    "cartes": {
      "href": "http://localhost:8080/comptes/2a55/cartes"
    },
    "operations": {
      "href": "http://localhost:8080/comptes/2a55/operations?categorie=commercant,pays",
      "templated": true
    },
    "solde": {
      "href": "http://localhost:8080/comptes/2a55/solde"
    },
    "transfert": {
      "href": "http://localhost:8080/comptes/2a55/transfert"
    }
  }
}
```

*Exemple de liens HATEOAS : la ressource "Compte"*

Pour faciliter l'utilisation de l'API par un utilisateur externe, une route "swagger" a également été ajoutée : elle permet d'accéder à la documentation générée par le composant éponyme. Pour certaines requêtes telles que l'authentification ou le transfert, des classes Java servant de modèle ont également été créés afin de faciliter l'utilisation de la documentation par l'utilisateur. En effet, le format du "body" à passer à l'API pour effectuer une requête est détaillée dans la documentation (cf. partie "utilisation de l'API").

Pour la conversion de devise, j'ai décidé d'implémenter un micro-service externe selon le modèle vu en cours (bourse-service). Ce micro-service permet d'obtenir le taux de change lors d'une opération à l'étranger et de convertir le montant associé. J'ai également implémenté une table "pays\_devise" dans l'application principale afin de pouvoir convertir facilement la devise en fonction du pays du compte. Avoir implémenté cette table dans l'application principale plutôt que dans le micro-service de conversion permet d'accélérer les traitements et d'assurer certaines transactions si jamais le service bourse tombe : en effet, le service ne sera pas appelé si deux pays partagent la même devise.

Au niveau de la sécurité, j'ai tout d'abord pensé à implémenter un service "oauth" externe, encore une fois selon le modèle vu en cours, avec une table "user\_compte" dans l'application principale permettant de trouver à quels comptes l'utilisateur connecté est autorisé à accéder. Néanmoins, j'ai abandonné l'idée de cette table car selon les spécifications du sujet, il semblait que l'utilisateur n'ait le droit d'accéder qu'à un seul compte (route GET /comptes inexistante). Il aurait bien évidemment été possible de filtrer sur les comptes autorisés lors de l'appel de "GET /comptes", mais j'ai fait le choix de n'en garder qu'un seul. Ce choix a également légèrement accéléré les transactions, car l'ID du compte est enregistré dans le "token" de l'utilisateur connecté (plus de détails dans la partie "réalisation" de ce rapport).

## Réalisation

Afin de réaliser cette API, j'ai utilisé les outils IntelliJ pour l'IDE, Postman pour exécuter facilement mes requêtes HTTP et GitHub pour garder une sauvegarde de mon code et facilement gérer mes versions. Au niveau des technologies, j'ai utilisé Java 8, Maven pour gérer mes dépendances, la dernière version en date de Spring Boot et les bibliothèques JPA, H2, OAuth, Starter Security, Swagger2 et Eureka.

J'ai tout d'abord commencé par implémenter les micro-services principaux "Comptes", "Cartes" et "Opérations" et par les faire fonctionner individuellement. Une fois toutes les routes fonctionnelles, j'ai relié les services entre eux via des "RestTemplate". J'ai pu ensuite implémenter les routes utilisant plusieurs services, comme "solde" ou "transfert" par exemple. Lors de cette étape, le "gateway" n'était pas encore présent et toutes les opérations passaient par le micro-service "Comptes".

La plupart des routes étaient basées sur les principes CRUD, mais certains mécanismes ont demandé une réflexion plus approfondie :

Pour la route "solde", j'ai décidé de renvoyer la somme des opérations effectuées sur le compte plutôt que de stocker un attribut supplémentaire. Cela permet d'obtenir une valeur toujours à jour et d'éviter les problèmes si un des services tombe, car les micro-services "Comptes" et "Opérations" sont indépendants.

```
@Query("SELECT SUM(o.montant) FROM Operation o WHERE o.compteid = :compteid")  
String findSoldeByCompteid(@Param("compteid") String compteid);
```

*Requête permettant de gérer le solde*

Pour la route "transfert", deux opérations sont créées : une opération sur le compte initiateur du transfert, avec le montant retiré, et une opération sur le compte destinataire, avec le montant ajouté (voir diagramme de séquence dans la partie "conception"). L'API fait en sorte que la seconde opération ne soit pas effectuée si la première n'a pas réussi, et qu'aucune transaction ne soit effectuée si il y a un problème de conversion de devise.

Le micro-service “Bourse” a été simple à implémenter : j’ai simplement repris le code de l’application développée en cours et l’ai adapté à cette API. Si une opération a lieu à l’étranger, l’application principale appelle le micro-service “Bourse” pour obtenir le taux de change et le montant converti dans la devise du pays de l’utilisateur.

L’étape suivante a été de créer le “gateway” en construisant un micro-service “Comptes” à part entière. Cela permet à l’utilisateur de pouvoir continuer à utiliser l’API si un des services tombe. Néanmoins, certaines requêtes telles que la création d’une opération auront toujours besoin d’utiliser ce service (pour obtenir le pays de la transaction par exemple).

Une fois le gateway créé, la couche HATEOAS a été ajoutée à l’application principale afin de permettre à l’utilisateur de naviguer facilement. Implémenter cette couche par dessus l’application plutôt que dans les micro-services permet de garantir que l’URL de l’API renvoyée est la bonne, et de respecter l’architecture WOA.

Une fois tous les services mis en place et correctement configurés, j’ai implémenté un service “Eureka” afin d’assurer le “load balancing” (pouvoir lancer les services plusieurs fois afin d’éviter les problèmes si l’un viendrait à tomber ou si une mise-à-jour était à effectuer) et de pouvoir éviter la configuration des hôtes et ports en dur dans le code.

La dernière étape a été d’implémenter un service “Authentification”. Cette étape a été la plus compliquée, car il existe beaucoup de manières différentes de gérer la sécurité sur Spring Boot. Après plusieurs tentatives plus ou moins fructueuses, j’ai décidé de repartir sur ce qui avait été fait en cours avec “OAuth” et de suivre un tutoriel en parallèle afin de développer un micro-service renvoyant un token d’authentification. Au niveau de l’application principale, l’utilisateur doit tout d’abord s’inscrire via l’URL “/sign-up”, puis se logger via l’URL “/login” afin de recevoir un token propre à son compte. Ce token contient l’ID du compte auquel il a droit d’accéder dans l’API. A chaque requête effectuée, l’API va vérifier que l’ID du compte contenu dans le token est bien le même que celui du compte sur lequel il cherche à obtenir des informations. Cette implémentation permet une bonne sécurité au niveau de l’accès aux données. Bien évidemment, pour une application bancaire il est nécessaire de chiffrer toutes les données de la base, mais cela n’était pas le but de ce projet. Le mot de passe que l’utilisateur utilise pour se connecter est néanmoins chiffré.

Une fois l’API entièrement fonctionnelle, j’ai également ajouté des tests unitaires dans chaque micro-service à l’exception de l’application principale et du service d’authentification. Ces tests permettent de garantir le bon fonctionnement de l’API à chaque modification. Au niveau de l’application principale, il aurait été intéressant d’implémenter des tests utilisant des “mock” pour tester les différentes requêtes et encore une fois assurer le bon fonctionnement de l’application après toute modification. Malheureusement, je n’ai pas trouvé de manière satisfaisante et à ma portée pour parvenir à mes fins.



## Réponse aux besoins

Au niveau des besoins exprimés par le sujet, l'API réalisée y répond parfaitement. Toutes les routes détaillées sont implémentées, et toutes les caractéristiques des entités sont respectées. Les liens HATEOAS ainsi que les codes réponse HTTP ont également été correctement implémentés. Au niveau de la sécurité, j'estime que ma réalisation est satisfaisante par rapport à ce qui a été demandé : l'utilisateur doit s'authentifier pour accéder aux routes principales de l'API, et il ne peut accéder qu'au compte lui étant associé.

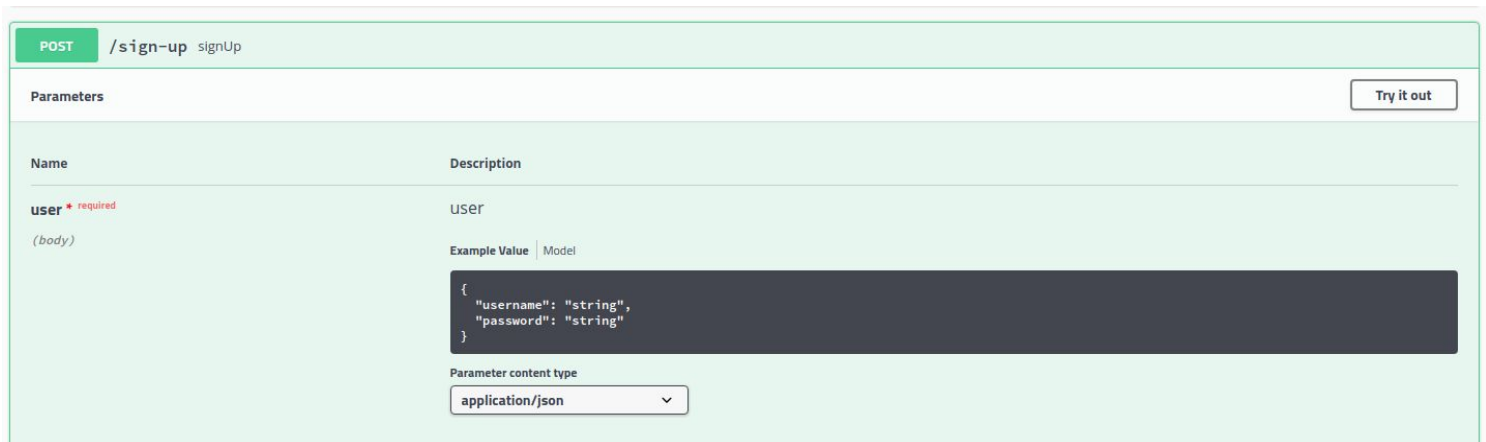
Néanmoins, et afin d'avoir un regard objectif sur mon travail, il me semble important de noter que certaines améliorations étaient encore possibles. L'implémentation du système de cartes aurait pu être meilleure : il aurait été intéressant de rester sur ma conception initiale et d'implémenter les routes PATCH, pour permettre à l'utilisateur de réaliser facilement les actions bloquer, activer la localisation, etc. sur sa carte. Bien évidemment, ce problème pourra facilement être réglé lors de la conception de l'interface graphique utilisant l'API via la route PUT. Toujours en lien avec les cartes, il est dommage qu'une carte ne soit pas liée à une opération : on ne garde pas en mémoire avec quelle carte une opération a été effectuée, ce qui peut être utile pour une application bancaire. Il est donc également impossible de gérer la suppression automatique d'une carte virtuelle avec cette implémentation. Enfin, comme mentionné dans la partie conception, l'utilisation d'ID plutôt que de l'IBAN du compte et du numéro de la carte n'était peut-être pas très judicieuse, bien qu'elle m'ait semblée appropriée lors de la conception. Cela reste néanmoins une question de point de vue et de spécifications du cahier des charges.

Au niveau de la sécurité, on pourrait noter que les microservices utilisés par l'API ne sont pas sécurisés (et ne renvoient pas de liens HATEOAS si utilisés sans passer par le service principal). Néanmoins, personne à part le service principal n'est censé accéder à ces services. Pour ajouter un niveau supplémentaire de sécurisation, on pourrait par exemple ajouter un proxy NGINX sur le serveur où sont lancés les services, qui autorisera uniquement les requêtes venant du service principal à accéder aux microservices.

## Utilisation de l'API

L'application principale est le service "Bank-service" : toutes les requêtes devront donc se faire sur ce service.

Afin d'utiliser l'API, l'utilisateur doit tout d'abord s'enregistrer via la requête "POST /sign-up" contenant un pseudo et un mot de passe de son choix au format JSON.



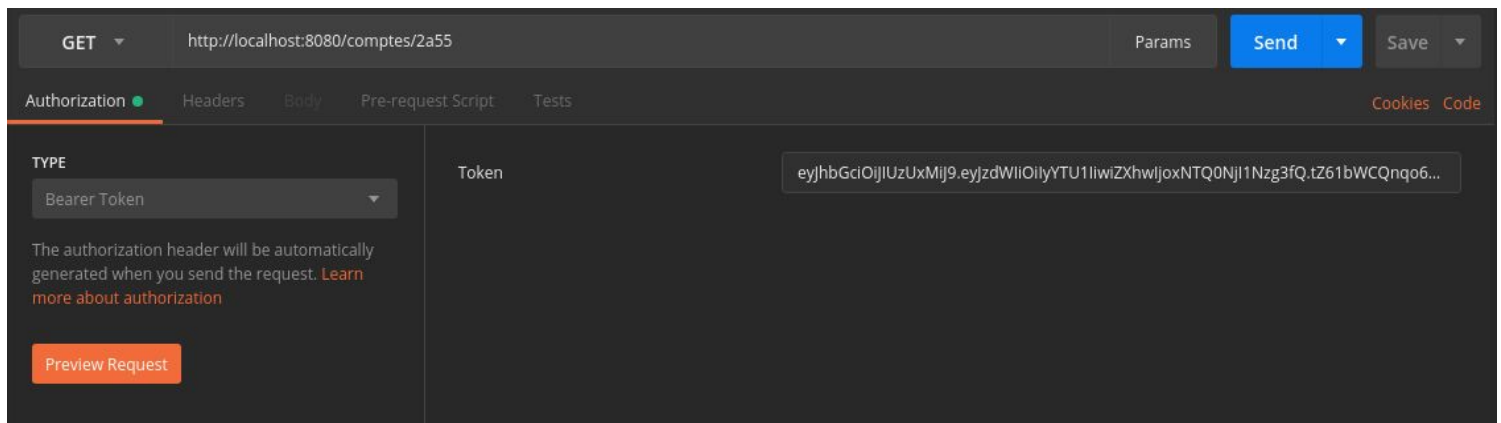
The image shows the Swagger UI for the POST /sign-up endpoint. The endpoint is labeled "POST /sign-up signUp". Under the "Parameters" tab, there is a parameter named "user" which is required and located in the body. The description for this parameter is "user". An example value is provided in a dark box: 

```
{  "username": "string",  "password": "string"}
```

. The parameter content type is set to "application/json".

### *Requête /login détaillée sur Swagger*

Si tout s'est bien passé, l'utilisateur devra ensuite se connecter via l'adresse "POST /login" et les mêmes informations JSON afin d'obtenir un "Bearer token" qu'il devra ensuite envoyer dans le "header" de chaque requête pour s'authentifier auprès de l'API



The image shows the Postman interface. The URL bar shows "http://localhost:8080/comptes/2a55". The "Authorization" tab is selected, and the "Type" is set to "Bearer Token". The "Token" field contains a long alphanumeric string: "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiIyYTU1IiwiaXNjaXNjaXNTQ0NjI1Nzg3fQ.tZ61bWCQnqo6...". There is a "Preview Request" button at the bottom left.

### *Utilisation du Bearer Token sur Postman*

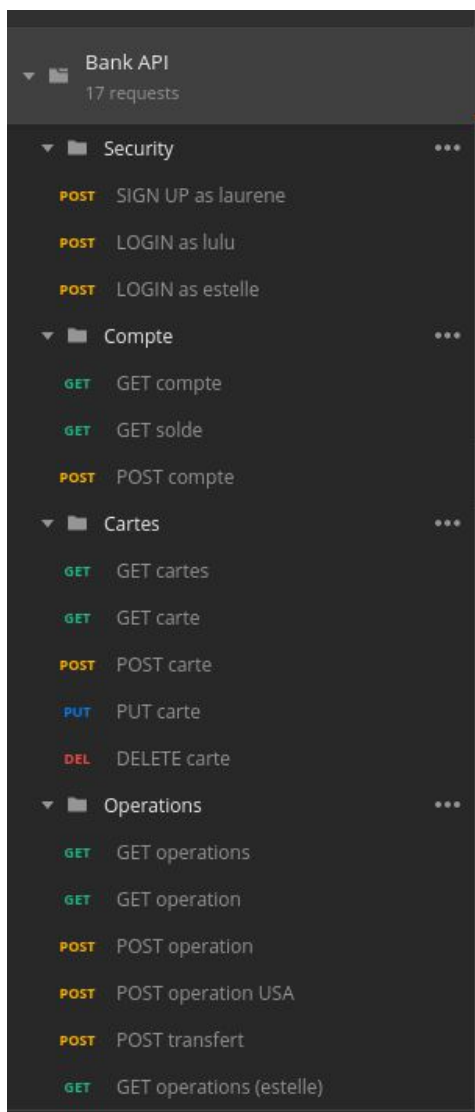
L'utilisateur pourra ensuite créer un compte via la requête "POST /comptes" en précisant les informations requises. L'ID du compte sera géré par l'API, qui se chargera de le récupérer en analysant son token. Une fois cette étape effectuée, l'utilisateur sera libre de naviguer sur l'API. A tout moment, il pourra se référer à la documentation Swagger afin de se renseigner sur le format de la requête à effectuer par exemple.

## Données et test de l'application

Afin de faciliter le test de l'application, des données concernant chaque route ont été ajoutées à l'application. Ces données se trouvent dans les fichiers "data.sql" de chaque micro-service. Ces données ne sont pas nécessaires pour lancer les tests unitaires. En revanche, il est nécessaire de lancer le service "Eureka" pour faire fonctionner les tests unitaires de chaque micro-service.

Pour fonctionner, l'application a besoin des services "eureka-service", "auth-service" et "bank-service". Les autres micro-services sont nécessaires pour des routes particulières, mais restent indépendants. Par exemple, "GET /comptes/{id}/cartes" a uniquement besoin du service "cartes-service" pour fonctionner.

Vous trouverez en plus des sources de l'application un fichier JSON contenant une collection de requêtes à effectuer sur Postman afin de tester facilement l'application. Ces requêtes utilisent les données contenues dans les fichiers "data.sql" et permettent d'éviter de créer un compte ou un utilisateur afin de tester l'application.



*Routes définies dans le fichier JSON*

## Conclusion

Ce projet a été un des plus intéressants réalisé au cours de mon cursus MIAGE. L'apprentissage de Spring Boot m'a permis de découvrir comment réaliser facilement une application et une API en Java. J'avais déjà dû réaliser une API au cours d'un stage, mais ce n'était pas avec les mêmes technologies, ni avec l'utilisation de micro-services. Mon expérience m'a permis de m'orienter plus facilement lors de la conception, mais un gros travail a été à fournir pour mener à bien la réalisation de l'API en respectant les spécifications et ma conception initiale.

L'utilisation de micro-services et l'authentification ont été les deux plus grandes difficultés : j'ai passé beaucoup de temps à me renseigner et à expérimenter afin d'obtenir un résultat fonctionnel et surtout compréhensible. Les utilisations de Spring Boot sont vastes, et on peut facilement se perdre si on ne reste pas fixé sur un objectif précis ou si on teste plusieurs manières d'implémenter une même fonctionnalité.

Au niveau du temps, une trop grande partie a été consacrée à la conception : ce n'est qu'à la réalisation que je me suis rendue compte que je ne savais pas comment réaliser certaines fonctionnalités, ou que c'était beaucoup trop complexe. Il m'a fallu recommencer le projet plusieurs fois avant d'arriver à un début d'API utilisable. Néanmoins, avoir dédié une grande partie du temps à la conception a été bénéfique car à tout moment de la réalisation, je savais dans quelle direction m'orienter et quelles fonctionnalités devaient être implémentées. Par exemple, commencer avec de simples "RestTemplate" puis ajouter Eureka par la suite a été une bonne décision par rapport au fait de commencer directement avec des "FeignClient".

Pour conclure, l'utilisation de Spring Boot est assez complexe et nécessite un grand temps d'apprentissage afin de réaliser un tel projet. Mais une fois l'architecture définie et l'API mise en place, on se rend compte que l'outil est très pratique et possède de nombreuses fonctions pour nous faciliter la tâche (Beans, Repository, Mapping, etc.)

# Annexes

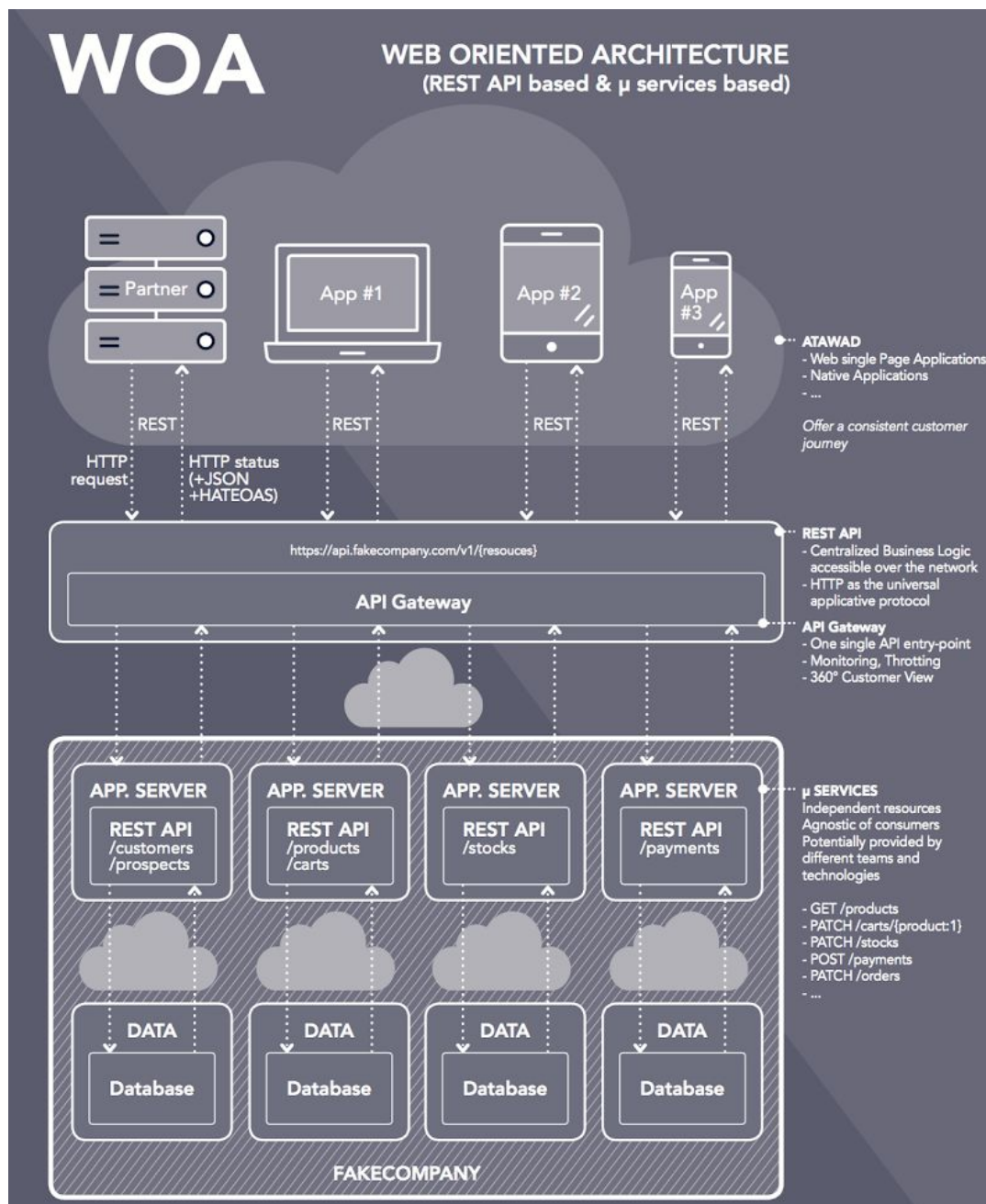
## GitHub

Lien de mon GitHub contenant les sources du projet :

<https://github.com/claurene/projet-api-rest>

(Sera disponible à partir du 03/12/2018)

## Architecture WOA



Source : <https://blog.octo.com/strategie-d-architecture-api/>

## Sources

- CM “Déploiement, passage à l’échelle et disponibilité”, Olivier Perrin
- CM “Méthodes de développement de SI distribués”, Rodislav Moldovan
- From Zero to Hero with Spring Boot, Brian Clozel  
(<https://www.youtube.com/watch?v=aA4tfBGY6jY>)
- Guides Spring (<https://spring.io/guides/>)
- Guides Baeldung (<https://www.baeldung.com/>)
- DZone pour le service authentification  
(<https://dzone.com/articles/implementing-jwt-authentication-on-spring-boot-api>)
- StackOverflow pour diverses interrogations (<https://stackoverflow.com>)
- Spring Initializr (<https://start.spring.io/>)