Picard jérémy

Projet API

Introduction

L'objectif du projet était de réaliser une API RESTful en utilisant Spring Boot, cette API avait pour but de gérer des comptes bancaires. Il fallait également intégrer HATEOAS à cette API et prendre en compte l'aspect sécurité, il était aussi nécessaire d'avoir plusieurs microservices communiquant entre eux, dans mon projet on trouve 4 microservices :

- La conversion de devise
- La géolocalisation de l'utilisateur
- Le commerçant utilisant la banque
- La banque

Pour commencer, je vais vous présenter mes choix de conception pour l'API ensuite de la réalisation et des diverses technologies que j'ai utilisées. Ensuite, je parlerai plus en détail de la sécurité de mon API et également du challenge que cela a posé pour tester l'application. Et enfin pour conclure, je parlerai de ce que j'ai appris de ce projet, et ce que je ferais différemment face un autre projet de cette ampleur.

Lien du projet sur GitHub :
https://github.com/Jeremy55/Projet-Banque-SID

Conception et choix

J'ai commencé par réfléchir à la structure de mes données en fonction du sujet. Le sujet donnant quasiment l'intégralité des attributs nécessaires pour les diverses entités, cette phase n'a pas été très longue. Pour les entités, j'ai décidé d'utiliser des IDs générées automatiquement, bien que la plupart des entités possèdent des attributs uniques (numéro de passeport pour le client, IBAN pour le compte, numéro pour la carte…) je pense que c'est plus simple à utiliser et que ça a plus de sens, cela n'empêche pas de faire des méthodes personnalisées dans les repositories pour récupérer les comptes par IBAN par exemple, ce que j'ai d'ailleurs fait dans mon projet.

Pour lier les entités, j'ai utilisé des annotations de type @OneToOne @ManyToMany… ce qui a posé des problèmes de cycle et de JSON infini, mais j'ai finalement rapidement corrigé ce problème.

J'ai très rapidement décidé de ne pas avoir de possibilité de suppression dans l'API, dans le cas d'une banque je n'ai pas trouvé de cas où une entité pouvait être supprimée, car il faut garder une trace de tout. J'utilise tout de même le verbe HTTP DELETE qui peut servir à un utilisateur à désactiver une de ses cartes bancaires par exemple.

Ensuite, après avoir étudié plus en détail les demandes du sujet, je me suis interrogé sur les fonctionnalités qui pourraient devenir des microservices à part entière en dehors de l'API de la banque. J'ai donc identifié une API de conversion de devise et une API de géolocalisation permettant la sécurisation des cartes bancaires ayant cette sécurité comme indiqué dans le sujet. À ce moment-là de la conception, la géolocalisation me semblait plutôt floue et je n'avais pas encore d'idée précise sur la réalisation.

Le choix des routes de l'API était plus difficile que je le pensais au début du projet. J'ai essayé de réfléchir à des routes courtes et logiques pour les utilisateurs. Par exemple, ayant sécurisé mon API avec Spring Security plutôt que de simplement faire une route POST sur /clients j'ai une route /clients/inscription qui est plus parlante. Pour la création d'un compte, j'avais dans la conception de base une route très longue, mais après avoir mis en place la sécurité, j'ai décidé d'utiliser simplement la route /comptes et je trouve l'utilisateur auteur de cette requête en fonction de l'email présent dans le jeton JWT. Vous pouvez trouver l'intégralité des routes dans le fichier Banque_SID.postman_collection à ouvrir avec Postman pour tester l'application.

La sécurité est l'aspect que j'ai le plus approfondi dans mon projet.

Ceci est le contenu d'un jeton JWT que j'utilise dans mon API. J'utilise les rôles pour rapidement bloquer les requêtes qui ne sont accessibles qu'à un utilisateur de type admin par exemple. (La route pour voir tous les comptes, toutes les opérations…).
Pour un utilisateur simple, j'utilise également le mail qui sert à retrouver le compte d'un utilisateur pour vérifier que la requête en cours concerne bien l'utilisateur utilisant le token.
Les tokens utilisés ont un TTL de 10 minutes, mais le serveur donne également un refresh token qui permet de récupérer un nouveau token sans se connecter en utilisant la route suivante : **/clients/token/rafraichir.**
Les tokens sont simplement signés avec "secret" ce qui n'est absolument pas sécurisé et qui pourrait permettre de falsifier des jetons et de se faire passer pour un autre utilisateur. Si cette API devait vraiment être utilisée, j'utiliserais une configuration externe avec un code plus compliqué pour signer les tokens. Mais dans cette conception, cela était trop complexe à mettre en place.

Conception de l'API de conversion :

Pour l'API de conversion, j'ai hésité à utiliser une API externe pour avoir des taux de change toujours à jour, mais j'ai finalement décidé de faire l'API moi-même. Elle fonctionne avec des paires d'échanges et retourne le montant converti en fonction des devises indiquées dans la route de l'API.

Conception de l'API de localisation :

Pour cette API je n'ai pas trouvé de solution sans utiliser de système externe, au début je voulais utiliser Redis et la fonctionnalité 'GEOSEARCH' mais je n'ai pas réussi à faire fonctionner cette solution comme je le voulais.

J’ai également essayé d’utiliser le “reverse geocoding” avec une librairie de Google, mais cette librairie ne propose plus de fonctionnalité offline, de plus ce qui est proposé par Google avec cette API était bien trop poussé pour mon besoin.

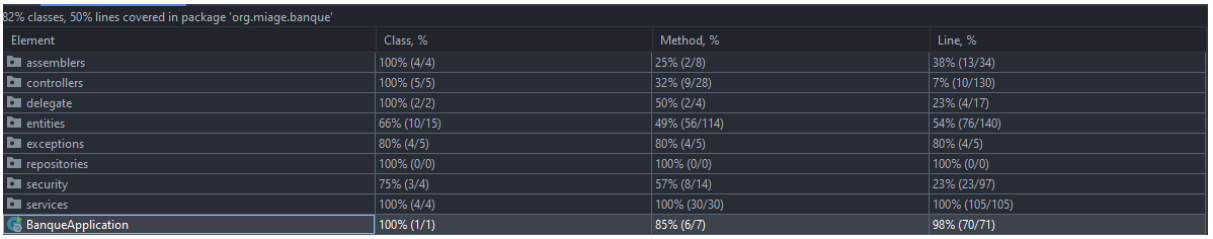
J’ai finalement trouvé une API externe qui correspondait parfaitement au besoin : “`http://api.geonames.org`” elle retourne le code d’un pays en fonction de la longitude/latitude. J’ai donc un projet Spring Boot qui communique avec cette API qui se trouve sur internet ce qui m’a permis d’apprendre à faire cela.

Conception de l’API du commerçant :

Cette API n’a rien de particulier, je savais déjà comment utiliser Consul pour communiquer avec un autre service grâce aux API précédentes.
En revanche, j'ai dû réfléchir à un moyen de changer la sécurité d’une de mes routes pour simplifier la communication entre les deux services.
J’ai donc ouvert la route POST **/operations/** à tous, même sans token, la sécurité se faisant grâce au code confidentiel à 4 chiffres de la carte bancaire, et également le cryptogramme dans le cas d’un paiement en ligne.

Réalisation

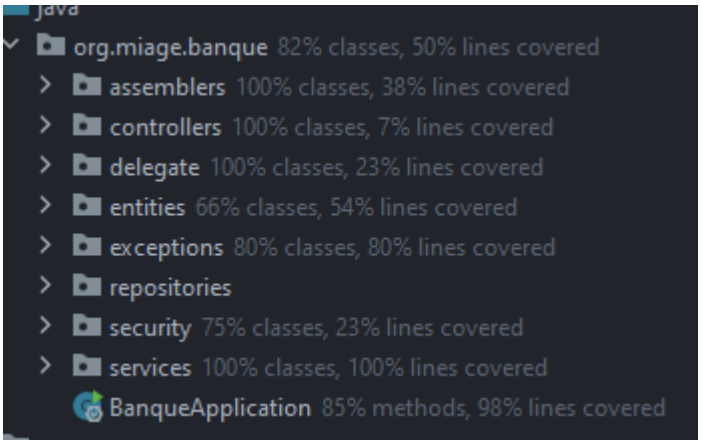
Au début, mon objectif était de terminer cette API en prenant en compte le fait que des API externes se chargeaient de certains aspects (géolocalisation, conversion). Je ne regrette pas ce choix, car la communication entre microservices s'est avérée très simple avec Consul. Ensuite tout au long de mon développement j’ai ajouté des tests, au début je pouvais coder mon application sans presque jamais utiliser des outils tels que Postman, mais j’ai fait l’erreur de ne pas prendre en compte l’aspect sécurité depuis le début de la conception. Quand j'ai décidé d’intégrer la sécurité dans mon API je me suis rendu compte que ma conception n’était pas vraiment compatible, et que j’aurai dû commencer par cela, j’ai donc presque entièrement refactorisé mon code. J’ai également dû supprimer mes tests qui n’étaient plus pertinents avec mon application modifiée, j’ai finalement réalisé mes tests en fin de projet ce que je regrette, car ils ne m’ont donc pas aidé, mais j’ai tout de même pu vraiment comprendre l’utilisation des mocks en les faisant. J'ai atteint une couverture de test que je pense être correcte.


82% classes, 50% lines covered in package 'org.miage.banque'

Element	Class, %	Method, %	Line, %
assemblers	100% (4/4)	25% (2/8)	38% (13/34)
controllers	100% (5/5)	32% (9/28)	7% (10/130)
delegate	100% (2/2)	50% (2/4)	23% (4/17)
entities	66% (10/15)	49% (56/114)	54% (76/140)
exceptions	80% (4/5)	80% (4/5)	80% (4/5)
repositories	100% (0/0)	100% (0/0)	100% (0/0)
security	75% (3/4)	57% (8/14)	23% (23/97)
services	100% (4/4)	100% (30/30)	100% (105/105)
BanqueApplication	100% (1/1)	85% (6/7)	98% (70/71)

Sauf pour les contrôleurs que je n'ai pas pu tester à cause de Spring Security, j’ai été capable de mocker un utilisateur, mais de nouveaux problèmes apparaissait constamment ce qui m’a fait perdre énormément de temps sur le projet et j’ai donc préféré avancer et avoir plus de fonctionnalité.

Pour la structure de mon application j'ai utilisé cela :



L'utilisation de services pour interagir avec les repositories m'a beaucoup aidé, j'ai pu mettre en place le principe de SRP le plus possible. Étant dans l'incapacité de tester les contrôleurs, j'ai pu aisément tester les services qui contiennent de toute façon toute la logique métier de l'API.

Pour garder des contrôleurs légers, j'ai découvert un mécanisme de Spring Boot qui permet de retourner des codes d'erreur en fonction des exceptions.

```
@ExceptionHandler(value = {
        CompteNotFoundException.class,
        ClientNotFoundException.class,
        CarteNotFoundException.class,
        OperationNotFoundException.class
        })
public ResponseEntity<Object> exception(RuntimeException exception) {
    return new ResponseEntity<>(exception.getMessage(), HttpStatus.NOT_FOUND);
}
```

Donc par exemple en cas de carte non trouvée, mon service lancera une exception de type "CarteNotFoundException" ce qui permettra de retourner un 404 avec un message personnalisé pour l'utilisateur. Ainsi je n'ai pas à gérer cela dans le contrôleur.

La partie "delegate" est utilisée pour interagir avec les autres microservices, dans le cas de la banque avec l'API de conversion et l'API de géolocalisation en cas de besoin.

Conclusion

Ce projet est l'un des projets les plus difficiles que j'ai eu à réaliser jusqu'ici, il y a eu de nombreux obstacles pour bien comprendre Spring Boot et terminer le projet. Il reste des erreurs et des parties du projet que je n'ai pas pu compléter comme je le voulais par manque de temps, mais je pense avoir beaucoup progressé en réalisant cette API. Je pense aussi avoir gagné assez de compétences pour éventuellement réaliser un projet personnel avec Spring Boot dans le futur, ce que je n'aurai pas pensé pouvoir faire avant de faire ce projet.

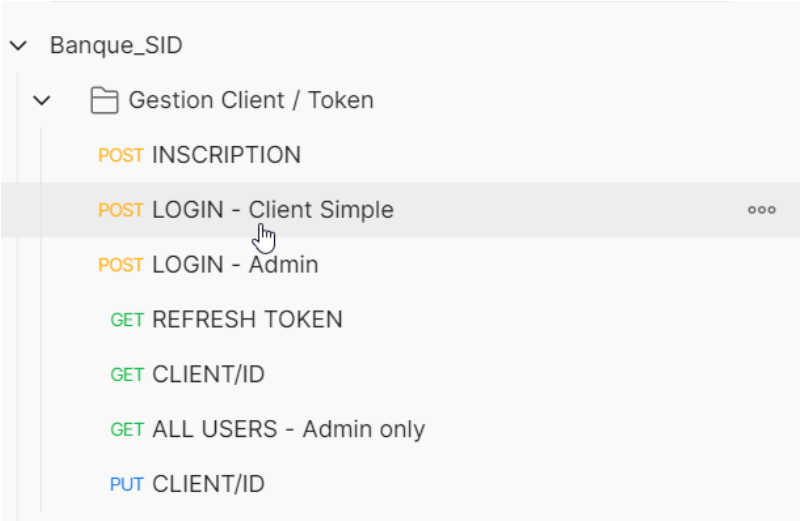
Si je devais refaire le projet maintenant il y a beaucoup de choses que je ferai différemment. Premièrement, je prendrais immédiatement la sécurité en compte, ayant trouvé la bonne façon de réaliser des tests à la fin du projet, je n'ai pas pu profiter de leurs intérêts pendant la phase de développement, je pourrais donc cette fois en profiter et progresser beaucoup plus rapidement.

Annexes

Utilisation de l'API avec postman :

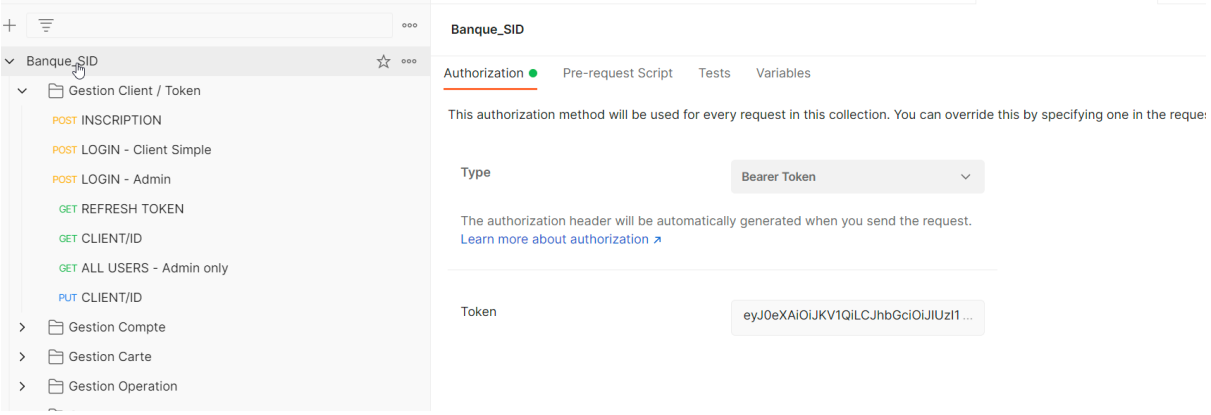
Une fois le fichier : Banque_SID.postman_collection.json chargé dans Postman, vous aurez accès à des requêtes pour tester l'API. Ce fichier est trouvable dans le dossier du projet à la racine.

Pour utiliser la plupart des requêtes, vous aurez besoin d'un token, pour en obtenir un, il faut utiliser la requête de connexion.



Une fois exécuté, vous récupérerez un token en réponse :

Il faut cliquer sur le dossier Banque_SID et dans l'onglet authorization choisir Bearer Token et faire un copier-coller du token précédemment obtenu dans l'input Token.



Comme cela, toutes les requêtes utilisent le token et cela sera plus simple à tester.