 

Rapport

Revolut Miage

**Clément Mercier**

**Année 2021-2022**

**Projet réalisé dans le cadre du module**

**« Passage à l’échelle »**

**Encadrant universitaire :** Olivier Perrin

1. Introduction

Dans le cadre du projet pour le module passage à l’échelle, il nous a été demandé de réaliser une API pour la gestion de compte bancaire permettant à des utilisateurs inscrits de gérer un compte bancaire, à la façon de Revolut.

1. Travail réalisé
2. Technologies utilisées
3. Java Spring

C’est un Framework open source pour construire et définir l’infrastructure d’une application Java avec de nombreuses fonctionnalités permettant de faciliter le développement et les tests. L’injection de dépendances largement utilisé pour la réalisation de se projet fait la force de Spring. La facilité pour créer des tests est également appréciable quand on souhaite avoir un code solide.

1. Spring Boot

Spring Boot est également un Framework qui s’appuie sur Spring en y rajoutant des fonctions orientées développement web comme les standards REST avec toute la configuration déjà faite ou encore le serveur permettant en un clic d’avoir une application qui tourne sans difficulté (c.f M Blanchard pour plus de complexité avec JEE). Etant donné que le projet porte sur une API, Spring Boot répond parfaitement à cette problématique.

1. Postman

C’est une application qui permet de faire des requêtes HTTP avec n’importe quelle méthode en y incluant des headers et un body. Etant donné qu’il n’y a pas de frontend dans le projet, Postman reste la solution la plus intéressante pour tester son API. Je m’en suis servis pour stocker tous mes « end points » afin de les tester avec une variable global contenant un token automatiquement ajouté à l’authentification d’un utilisateur

1. Trello

C’est une application web permettant de gérer son développement en y ajoutant des cartes représentants les fonctionnalités à faire dans l’application avec tout un cycle de vie (SCRUM). Le tableau est accessible ici : <https://trello.com/b/HGPKFkNT/revolutmiage>

1. Dépendances du projet

Pour la création du projet et l’ajout de toutes les dépendances nécessaires à la réalisation de celui-ci, j’ai utilisé le site « Spring Initializr » pour générer un projet initial avec un pom.xml fonctionnel. On y retrouve « Lombok » qui est une librairie permettant de réduire le code des classes en y ajoutant des annotations qui générerons des getters/setters ou encore des constructeurs à la compilation. Spring Boot a également été ajouté pour inclure toutes les fonctionnalités nécessaires à la création de l’API. La dépendance JPA a également été ajouté pour faciliter la gestion de la persistance des entités au sein de l’application. C’est un ORM. D’autres librairies ont également été ajouté pour faciliter l’écriture des tests, la validation des éléments fournis par l’API ou encore la structuration des requêtes renvoyées par notre application (Hateoas).

1. Architecture du projet

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

*Figure 1 : Architecture du projet*

L’outil de Spring pour générer un projet a déjà fixé l’architecture de base du projet avec un « src » contenant tout le code du projet. Sous ce répertoire on a « main » qui concerne le code de l’application et « test » qui, comme son nom l’indique contient les tests. Dans le code de l’application, on retrouve une partie « java » avec les classes Java et « resources » contenant les ressources de l’application comme les propriétés/configuration ou encore la gestion de la base de données.

Dans le but de faire une application des plus solides possible, j’ai choisi de mettre en place une architecture hexagonale. J’espère en respecter tous les codes parce qu’on n’en a pas encore trop fait mais je trouve tout de même puissant de pouvoir séparer le code métier de l’architecture pour que l’application soit modifiable, en termes de technologies utilisées sans avoir un impact significatif sur le temps à y passer.

Une image contenant texte

Description générée automatiquement

*Figure 2 : Structure de l’Architecture hexagonale de la carte*

Le plus difficile aura été de définir ce qui fait partie du domaine et ce qui fait partie de l’infrastructure. De ce que j’ai compris, le domaine contient les éléments qui sont indépendants de toutes technologies. Comme vous pouvez le voir dans la *Figure 2* il y a l’entité carte défini dans le domaine car peu importe la technologie utilisée, l’entité ne va jamais être modifiée. Ces attributs ne bougeront pas. Sauf si bien entendu, une nouvelle fonctionnalité amène à l’ajout ou la suppression d’un champ.

Dans l’architecture hexagonale, il y a également une autre règle importante : L’infrastructure peut dépendre du domaine mais jamais l’inverse. Ce qui signifie qu’une classe du domaine peut être appelé dans l’infrastructure mais une classe de l’infrastructure ne peut jamais être appelé dans le domaine. Pour que cela soit possible, j’ai utilisé ce que nous offre de mieux Spring : « L’injection de dépendances » !

Dans l’infrastructure, j’ai ajouté l’interface de JPA sous le package Hibernate.

Une image contenant texte

Description générée automatiquement

*Figure 3 : Interface carte de JPA*

Dans le package « cmd » du domaine, il y a les classes qui permettent la validation des différents appels à l’API. Mais certaines validations passent par un appel en base pour vérifier l’existence d’une entité. Par exemple, lors de la création d’une carte, il faut vérifier si le compte assigné à celle-ci existe. Malheureusement, pour se faire, il faudrait faire appel au JPA présent dans l’infrastructure mais comme dit précédemment, on ne peut pas appeler l’infrastructure depuis le domaine. C’est là que l’injection de dépendance opère. Dans le domaine, j’ai créé un package « catalog » contenant les interfaces des opérations pouvant être effectuées sur les entités de l’application.

Une image contenant texte

Description générée automatiquement

*Figure 4 : Catalogue de l’entité carte*

Ces interfaces sont implémentées par des classes adapteurs pour JPA en tant que service. En appelant le catalogue, il est donc possible d’accéder aux méthodes exclusives à JPA.

Une image contenant texte

Description générée automatiquement

*Figure 5 : Adapteur entre JPA et le catalogue carte*

1. Le format des réponses

Dans un premier temps pour les réponses des requêtes à l’API, je renvoie toujours un DTO pour être sûr que les entités soient toujours bien sérialisées. Ce qui veut dire que pour chaque entité, un DTO est créé.

Pour exemple, l’entité Utilisateur contient un attribut « dateDeNaissance » avec un type « LocalDate » et on ne sait pas trop comment celui-ci va être sérialisé dans la réponse donc l’utilisation d’un DTO peut être intéressant.

Une image contenant texte

Description générée automatiquement

*Figure 6 : Entité Utilisateur*

Pour passer d’une entité à son DTO il est nécessaire d’utiliser un « mappeur ». Celui-ci reprend tous les attributs de l’entité pour les convertir dans des types simples. Comme on peut le voir dans la *Figure 6*, il y a un attribut contenant la liste des comptes et cela peut devenir vite chronophage de reparcourir les attributs de chaque compte pour les mapper en DTO. Si une modification doit être apporté à la classe compte, il serait nécessaire de la répercuter partout. C’est pour cela qu’il est possible de définir un DTO comme type « simple » pour cette liste, ce qui donnerait une liste de DTO de compte à la sortie du mappeur. Le mappeur peut être rébarbatif à mettre en place pour toutes les entités car c’est toujours la même opération à faire : récupérer tous les attributs en les convertissant s’il le faut pour créer le DTO. Mais grâce à MapStruct, il n’est pas forcément nécessaire de le faire, il suffit de créer une interface en y ajoutant les bonnes annotations pour qu’à la compilation du projet, Spring génère l’implémentation de cette interface pour nous éviter de le faire nous-même.

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

*Figure 7 : Interface Mapper de l’utilisateur*

Comme vous pouvez le voir dans la *Figure 7*, il est possible de fixer les attentes concernant la conversion d’un type complexe dans une annotation directement. Il suffit ensuite de simplement packager le projet pour que l’implémentation soit créée. Il suffira par la suite d’appeler cette interface dans le code et l’injection de dépendances fera le reste en allant directement chercher l’implémentation de votre mappeur dans le projet. Moins il y a de code a écrire et moins il y a de bug.

Une image contenant texte

Description générée automatiquement

*Figure 8 : Emplacement du mappeur Utilisateur généré*

Une image contenant texte

Description générée automatiquement

*Figure 9 : Mappeur Utilisateur généré*

Il est également possible de tester ces mappeurs pour être certains qu’il n’y a pas un souci sur la génération de l’implémentation. Il suffit d’appeler l’interface du mappeur à tester et de vérifier si les valeurs du DTO sont celles attendues.

Une image contenant texte

Description générée automatiquement

*Figure 10 : Test du mappage d’une entité Utilisateur en DTO*

1. La validation

Dans une API, le plus important est de vérifier toutes les données qui sont envoyés par un utilisateur, le but étant de ne faire confiance à personne et de faire valider la moindre saisie pour ne pas avoir de soucis. Il a donc été nécessaire que je mette en place un moyen de valider les saisies de façon simple.

Pour la création d’un utilisateur, je récupère le body du POST sous forme d’un Input. J’envoie ensuite cet Input dans une classe qui va gérer la validation.

Une image contenant texte

Description générée automatiquement

*Figure 11 : Ressource qui permet de créer un utilisateur*

« UpdateUtilisateur » est une interface implémentant l’interface « Consumer ». Cette interface a comme seule utilité d’offrir une méthode qui prend un paramètre, effectue un traitement et qui ne retourne rien. L’objectif étant de retourner une exception dans cette unique méthode dès qu’il y a un problème de validation. S’il n’y a pas de soucis, alors on fait le traitement souhaité (dans le cas présent, on créer l’utilisateur)

Une image contenant texte

Description générée automatiquement

*Figure 12 : Consumer de l’update d’un Utilisateur*

Une image contenant texte

Description générée automatiquement

*Figure 13 : Implémentation du Consumer Utilisateur*

Dans la *Figure 13* on a bien dans un premier temps la validation sur les saisies de l’utilisateur qui est appelée. Si aucune exception est retournée alors on passe à l’exécution du traitement. Comme dit précédemment, la validation étant dans le domaine, on n’appelle pas directement le repository d’utilisateur pour le sauvegarder mais bien le catalogue.

La validation se découpe en deux parties. La première concerne la validation du format des données transmise et la deuxième partie concerne plutôt la validation de la cohérence des données saisies.

Pour ce qui est de la première partie, j’utilise le moyen vu en cours qui permet de définir certaines propriétés sur le format d’un attribut.

Une image contenant texte

Description générée automatiquement

*Figure 14 : Contraintes de validation pour les champs d’un utilisateur*

La validation est ensuite faite dans une classe abstraite qui est identique à chaque entité. Le but étant de factoriser un maximum le code pour en réduire la dette technologique. Cette classe fait l’appel à la méthode qui permet la validation. La première validation touche le format des données. Si le format est incorrect alors on ne passe pas à la suite et on retourne une exception contenant les attributs avec leurs problèmes détectés. Si le format des données est correct on passe à la deuxième partie de la validation qui est une méthode qui sera définie dans la classe qui héritera de notre classe abstraite

Une image contenant texte

Description générée automatiquement

*Figure 15 : Classe abstraite qui gère la validation*

Dans la classe qui hérite de « DefaultValidator » j’ai pu définir différents comportements suivant les contraintes des entités bloquer une éventuelle mauvaise saisie (impossible de créer deux utilisateurs avec le même passeport, opération sur un même compte, …). L’utilisateur saisissant les données sera alors informé du problème lié à sa saisie.

Une image contenant texte

Description générée automatiquement

*Figure 16 : Classe qui redéfini la méthode de validation spécifique*

Il me suffit alors de faire des conditions pour vérifier la cohérence des saisies. Dans la *Figure 16*, si je suis entrain de créer l’utilisateur, alors je vérifie si le numéro de passeport renseigné n’existe pas déjà. Si c’est le cas alors j’ajoute une description du problème dans la Map pour pouvoir la renvoyer à l’utilisateur. Le but étant de donner un identifiant unique au problème pour qu’avec un éventuelle frontend de l’application, on puisse mettre des messages personnalisés à l’utilisateur suivant le souci rencontré. C’est à la ligne 27 de la *Figure 15* que je retourne ou non une exception si jamais il y a une incohérence dans les informations saisies.

La génération d’une exception n’est pas suffisante pour renvoyer les bonnes informations à l’utilisateur. Il est nécessaire de « catcher » l’exception quand il y en a une pour faire un traitement dessus afin d’organiser les données avant de les envoyer au client. Pour se faire, Spring vol une nouvelle fois à notre secours. Il existe une annotation qui fait appel à une méthode spécifique si jamais une exception définie est levée. Il suffit alors de faire le traitement souhaité pour l’exception afin de le retourner à l’utilisateur.

Une image contenant texte

Description générée automatiquement

*Figure 17 : Classe qui « catch » les exceptions pour préparer l’affichage des soucis*

Une image contenant texte

Description générée automatiquement

*Figure 18 : Requête avec Postman contenant des problèmes de saisie*

1. La sécurité

Concernant la sécurité, Spring Security m’a permis de gagner du temps. Cette librairie rajoute une couche de sécurité concernant l’accès à l’application. Par défaut, ça bloque tous les accès et il est nécessaire de définir une politique route par route pour gérer l’accès à ceux-ci.

Une image contenant texte

Description générée automatiquement

*Figure 19 : Classe de configuration de spring-security*

Par défaut, j’autorise la route pour créer un utilisateur ainsi que celle pour se connecter. Cela veut dire qu’il n’est pas nécessaire d’être connecté pour faire une requête sur ces deux routes là. La ligne 47 permet de dire que pour toutes les routes qui commence par « /api » il est nécessaire d’être authentifié sinon, l’application retourne une 401 pour indiquer que l’utilisateur n’est pas connecté. Avec une annotation, je peux ensuite définir le rôle qui a accès à une route pour par exemple définir des routes réservées aux administrateurs ou autre.

Pour ce qui est du moyen de s’authentifier, c’est avec un numéro de passeport et un mot de passe que l’utilisateur se connecte en faisant appel à la route « /authenticate ». Si tout est bon, l’application retourne un token JWT à l’utilisateur. Il lui suffit ensuite pour chaque route, d’ajouter ce token dans le header. Initialement, je voulais utiliser l’Identity Provider Keycloak pour laisser la gestion de l’authentification à ce service mais malheureusement, je n’ai pas trouvé de solution pour créer un utilisateur sans passer par l’interface graphique (jusqu’à ce qu’un autre étudiant me la donne…) et j’ai dû gérer directement dans mon application l’authentification. Récupérer le token, vérifier s’il est valide et récupérer les informations à l’intérieur pour connaître l’identité de l’utilisateur qui essaye d’accéder à l’API (et ça n’a pas été simple à mettre en place !).

1. Les tests

Dans l’application, j’ai essayé de faire un maximum de tests pour garantir un code fonctionnel et sûr. J’étais déjà convaincu avant et je trouve que c’est une véritable sécurité de faire cela. Même si cela m’a pris beaucoup de temps, j’ai testé les mappeurs pour garantir leurs bons fonctionnements (je ne suis pas convaincu que ce soit nécessaire mais dans le doute, je l’ai fait). J’ai également testé l’API en simulant des requêtes avec différentes saisies pour garantir la couverture de tous les cas possibles. J’ai également testé les validateurs des entités pour garantir que je n’ai pas oublié un cas spécifique.

1. Hateoas

J’ai ajouté Hateoas au projet pour que les utilisateurs de l’API puissent facilement retrouver les requêtes qui sont possibles de faire. Les liens apparaissent uniquement dans des requêtes de type GET car les autres n’ont pas d’intérêt.

1. Application qui communique avec la banque

J’ai ajouté une application sur un autre port qui correspond à un paiement avec un terminal dédié. L’objectif étant de donner une carte bleue avec un code ainsi que le montant à payer. Une fois cela fait, le terminal fait une requête à notre banque application pour valider et effectuer l’opération. Je l’ai représenté de façon simple : le numéro de la carte, le code ou le sans contact, le montant sont les seules infos à transmettre quand un fait une requête avec l’API des ventes.

Une image contenant texte

Description générée automatiquement

*Figure 20 : Appel de l’API des ventes pour faire une opération*

Cette route n’est pas sécurité par un token car elle doit permettre à des commerces de s’y connecter. Il serait judicieux de leurs fournir un token pour des raisons de sécurité mais je n’ai pas eu le temps de le faire. La validation est toujours présente et permet s’il y a le moindre souci avec le paiement de donner l’information au commerce de la même manière que le fait la banque API avec ces utilisateurs.

1. Scalabilité

Je n’ai pas eu le temps d’intégrer cette fonctionnalité dans mon projet.

1. Dockerisation

Je n’ai pas eu le temps d’intégrer cette fonctionnalité dans mon projet.

1. Circuit Breaker

Je n’ai pas eu le temps d’intégrer cette fonctionnalité dans mon projet.