PP3 - Architecture des Applications d'Entreprises

Ayoub Yoann, Monneret Martial et Moyse Antoine

19 décembre 2022

Table des matières

1 Introduction

1.1 Contexte

— **PrixBanque** : PrixBanque est une nouvelle « fintech » qui cherche à développer des solutions simples, fiables, sûres et 100% numériques pour la vie financière de ses clients. Elle cherche à gagner la confiance du client en créant et en fournissant de nouvelles expériences pour ses clients.
— **Nous** : En tant qu'institution, nous voulons offrir des technologies de **haute disponibilité** à nos clients et à notre culture, nous voulons que notre plate-forme financière soit construite en tenant compte de la simplicité, la fiabilité, la disponibilité / résilience et l'évolutivité de nos solutions.

L'entreprise à donc fait appel à nos services pour le grand lancement de son nouveau système bancaire en ligne.

Nous avons déjà réalisé deux premières études pour l'entreprise afin d'établir un prototype et de décrire l'architecture que nous leur proposons.

1.2 Problématique

L'entreprise souhaite maintenant que nous implémentions une architecture pouvant répondre à l'ensemble des exigences fonctionnelles ainsi qu'à l'une des exigences non fonctionnelles : le nombre de transactions par seconde. Effectivement cette exigence non fonctionnelle est centrale dans le bon fonctionnement de l'application puisque PrixBanque devra effectuer environ 100 millions de transactions bancaires par année, soit 2000 transactions par seconde.

C'est dans ce contexte, et pour répondre à cette problématique, que nous présentons dans ce document le troisième volet de notre étude, à savoir l'implémentation d'un MVP correspondant à la solution proposée et répondant aux exigences demandées. Ce document présentera également la preuve de concept en démontrant le nombre de transactions par seconde que le MVP peut effectuer et la façon dont l'architecture y contribue.

En plus de fournir ce document et le code source du MVP, nous fournirons en plus, une vidéo démontrant la preuve de concept.

2 Minimum Viable Product (MVP)

Dans cette partie, nous allons voir comment le MVP a été conçu. Nous commenceront par présenter les différentes exigences fonctionnelles qui constituent le système, les exigences de développement, les exigences architecturales nécessaires à sa bonne construction et, enfin, les exigences de documentation pour sa bonne installation et sa bonne utilisation.

2.1 Exigences fonctionnelles

Commençons par les exigences fonctionnelles. Les exigences fonctionnelles sur lesquelles nous nous sommes concentrés sont les suivantes :

EF1	Création de compte
Description	Permettre à un nouvel utilisateur de se créer un compte client
Priorité	Priorité élevée
Cas d'utilisation associé	[Création_de_compte]
Utilisateur	Un nouvel utilisateur

EF2	Voir le solde du compte
Description	Permettre au client de consulter le solde de son compte ainsi que d'autres informations supplémentaires relatives au solde
Priorité	Priorité élevée
Cas d'utilisation associé	[Voir_Solde]
Utilisateur	Un client

EF3	Connection compte client
Description	Permettre à un client de se connecter à son compte client
Priorité	Priorité élevée
Cas d'utilisation associé	[]
Utilisateur	Un client

EF4	Transfert de valeur
Description	Un client doit pouvoir envoyé de l'argent à un autre client de la banque
Priorité	Priorité élevée
Cas d'utilisation associé	[virement]
Utilisateur	Deux clients

2.1.1 Création de compte

En ce qui concerne la création de compte, il faut créer un modèle permettant d'identifier un client. Nous avons donc un modèle de classe "Client" qui contient tous les attributs nécessaires à la création et identification d'un client. La classe "Client" est constitué des attributs suivants :

- Un id unique généré automatiquement ;
- "firstname" représentant le prénom du client ;
- "lastname" représentant le nom du client ;
- "mailadress" représentant l'adresse mail du compte client ;
- "pwd" représentant le mot du passe du compte client.

Les fonctions que l'on retrouve dans cette classe sont des getters et des setters pour chacun des attributs qui sont en privés.

La page de création de compte se trouve à l'adresse localhost :8080/register. Les champs de saisie on été créés de manière à au moins vérifier une potentielle adresse mail valide. Une vérification de l'adresse mail est faite pour voir si elle n'existe pas déjà dans la base de données. Il n'y a pas de conditions particulières sur la saisie de mot de passe. Le but est ici d'avoir quelque chose de fonctionnel sans pousser le développement trop loin. Une fois tous les champs correctement saisis, le nouvel utilisateur est créé dans la base de données.

2.1.2 Authentification

L'authentification utilise le même modèle que précédemment. Ici, nous allons simplement lire la base de données afin de voir si les identifiants rentrés par l'utilisateur sont bien connus du système. Si c'est le cas, le client est directement redirigé sur sa page de solde que nous allons voir juste après.

Sinon, un petit texte apparaît pour lui dire que les informations rentrées sont incorrectes. Le mot de passe, que ce soit pour sa création ou pour sa modification, est crypté puis vérifié dans la base de données que le mot de passe saisi est bien le même que celui stocké.

La page de connexion est accessible sur l'adresse localhost :8080/login. Cette page contient aussi un bouton permettant de s'enregistrer si on ne possède pas déjà de compte client.

2.1.3 Solde

Pour la consultation du solde par le client, nous devons créer une nouvelle classe qui va contenir toutes les informations relatives à cette donnée. La classe "Solde" est donc constituée des attributs suivants :

- Un id unique généré automatiquement ;
- "montant" représentant la somme présente sur le compte client ;
- "client" représentant le client auquel est lié le compte.

Avant toute chose, il faut savoir qu'à chaque fois qu'un nouvel utilisateur s'enregistre, un solde est automatiquement créé et est attribué à ce client. Pour les phases de tests, le montant de tout nouveau compte est de $10000.

Pour le bon affichage du solde du compte client, nous devons d'abord retrouver le compte client de la session pour ensuite aller chercher le solde associé au client. Une fois le solde trouvé, le montant est directement affiché sur la page web.

L'adresse pour accéder au solde est à l'adresse localhost :8080/solde. Cette page est uniquement accessible si l'utilisateur est connecté, il est impossible d'y accéder autrement.

2.1.4 Virements

Enfin, pour les virements, il nous faut un nouveau modèle. Dans notre cas, la classe "Virement" est composée des attributs suivant :

- Un id unique généré automatiquement ;
- "montant" représentant la somme à transférer ;
- "Clientpayeur" représentant le client qui envoie l'argent ;
- "Clientreceveur" représentant le client qui reçoit l'argent.

Il n'est pas nécessaire dans notre cas de stocker ces données de transfert, le but principal de ce MVP étant de montrer la fiabilité et la robustesse de notre application. Mais il est intéressant, pour l'entreprise et pour notre preuve de conception, de garder une preuve écrite de chaque transaction.

Une fois que le client arrive sur la page de virement, il peut entrer le montant et l'email du destinataire. La saisie du montant est contrôlée pour ne pas avoir de valeur négative et d'accepter deux chiffres après la virgule.

Tout comme la page de solde, il est nécessaire d'être connecté pour pouvoir accéder à cette page. Une fois que le client a rentré les informations nécessaires et validé la saisie, le système va récupérer le client connecté à la session et celui associé à l'adresse mail rentrée par l'utilisateur. Il va ensuite retrouver les soldes associés aux deux clients pour soustraire le montant au solde du client payeur et ajouter le montant au solde du client receveur.

Il n'y a pas de vérification poussée quant à l'adresse mail rentrée par l'utilisateur. Si l'adresse mail n'est associée à aucun client dans la base de données, le système renverra une erreur pour un utilisateur. De plus, il n'y a pour l'instant, aucune vérification pour savoir si le client à suffisamment de fonds pour effectuer le payement.

Nous nous sommes vraiment concentrés sur le respect de l'architecture du projet tout en amenant des fonctionnalités pertinentes pour prouver la robustesse et la fiabilité du système. Les erreurs que nous avons énumérées précédemment sont des erreurs facilement corrigeables mais que nous n'avons pas jugé très importantes dans notre cas.

2.2 Exigences de développement

Dans cette partie, nous allons aborder les différentes exigences de développement lié à la construction de notre MVP.

2.2.1 Contrôle de version

La gestion de versions consiste à gérer l'ensemble des versions d'un ou plusieurs fichiers. Essentiellement utilisée dans le domaine de la création de logiciels, elle concerne surtout la gestion des codes source. Cette activité étant

fastidieuse et relativement complexe, un appui logiciel est presque indispensable.

Dans notre cas, nous avons utilisé Git pour plusieurs raisons. La première est que c'est le contrôleur de version le plus connu et le plus utilisé. Il est relativement simple d'utilisation et beaucoup de plateformes de gestion de version utilise cette technologie. Dans ce sens, nous avons opté pour un dépôt sur GitHub pour gérer les versions de notre code source.

Le code est disponible à l'adresse suivante : `https://github.com/AntoineMoyse/application-architecture-prixbanque/tree/master`

Le README de notre projet contient des informations supplémentaires quant à la bonne compréhension du système dans sa globalité.

2.2.2 Mécanisme de "build"

Pour le mécanisme de build, nous utilisons Maven. Maven est un outil de gestion et d'automatisation de production des projets logiciels Java en général et Java EE en particulier. Il est utilisé pour automatiser l'intégration continue lors d'un développement de logiciel.

Comme nous utilisons spring boot pour le développement de notre MVP, il est plus simple, pour nous, d'utiliser les dépendances de Maven afin de mener à bien notre MVP. Maven nous offre beaucoup de bibliothèques Java très utiles à la construction de notre système notamment sur la gestion de base de données ou encore de la sécurité avec la dépendance springframework security qui nous a beaucoup aidé pour tout ce qui est gestion d'identification et d'inscription.

Maven est de plus très simple d'utilisation : il suffit de rajouter quelques lignes dans le fichier "pom.xlm" puis de mettre à jour le projet pour pouvoir utiliser de nouvelles bibliothèques. C'est aussi le mécanisme de build avec lequel la totalité des membres de l'équipe se sentaient confortables à utiliser.

2.3 Exigences architecturales

Dans cette partie, nous allons aborder toutes les exigences architecturales dont nous avons déjà parlées plus en détails dans notre dernier rendu. Le but ici est d'en respecter l'implémentation afin de prouver leur efficacité.

2.3.1 Microservice pour chaque exigence fonctionnelle

Chaque exigence fonctionnelle à été construite en suivant le modèle de microservice. Chaque service a son modèle, son controller, son repository et sa classe service. Les controllers communiquent tout de même entre eux mais uniquement d'un contrôleur à un autre. Les repository communiquent avec les base de données et les services avec les controllers, ce qui nous permet, au final, de pouvoir utiliser toutes les fonctionnalités nécessaires en passant uniquement par les controllers de chacun.

Chaque service possède aussi sa base de données unique mais nous y reviendrons plus tard.

L'idée est de décomposer une application en plus petites briques élémentaires qui sont réutilisables dans plusieurs contextes différents. Chaque brique va alors exécuter son propre processus et va communiquer avec le reste de l'application via des mécanismes qui demandent peu de ressources, en utilisant notamment des APIs REST. C'est un style qui est de plus en plus utilisé pour créer des applications d'entreprise.

Cette façon de développer une application présente plusieurs avantages. La maintenabilité et l'évolutivité de l'application est optimale. Cependant, et dans la pratique, l'implantation de chaque service demande des connaissances et de l'expérience que nous n'avons pas pour l'instant. Nous avons donc essayé de se rapprocher le plus possible de cette architecture en ayant tous les services sur une seule et même application mais faire en sorte de les isoler les uns des autres.

Au final, même si nous n'avons pas réussi à implanter un système de microservices pur et dur, nous avons fait en sorte de s'y approcher au maximum afin de pouvoir tester si une architecture de ce type est idéal pour notre MVP et les contraintes qui y seront appliquées.

2.3.2 Load balancer

Le load balancing est un mécanisme permettant de distribuer efficacement le traffic arrivant sur une application ou un service. Un load balancer est placé en amont des serveurs et permet de router les messages vers les différentes instances du système afin d'équilibrer la charge de travail. L'objectif est d'optimiser la vitesse de réponse d'un service et de s'assurer qu'aucun d'eux n'est surchargé. De plus, cela permet de faire face aux problèmes de services indisponibles. Dans notre architecture microservices, l'intêret est de faire tourner plusieurs instances d'un même service sur différent ports. Il est ensuite possible de rediriger les requêtes en fonction de l'algorithme de notre choix.

Il existe plusieurs algorithmes de load balancing dont les suivants :
- Round-Robin : route les informations vers les serveurs dans un ordre précis. C'est une technique assez facile à mettre en place.
- Least connection : regarde quelle instance possède le moins de connexions.
- Random : on route de manière aléatoire les messages vers les instances.

Dans un premier temps on pourrait mettre en place un algorithme round-robin ou random car ce sont des algorithmes faciles à mettre en place. Cependant, un mécanisme comme le least connection parait être le plus efficace mais légèrement plus complexe à implémenter.

Pour mettre en place un load balancer avec Spring Boot Load Balancer, il est nécessaire d'avoir au moins deux applications actives : une application serveur et une application client, c'est-à-dire l'application mobile de l'utilisateur et les services qui tournent comme des applications. Dans notre cas, nous n'avons pas d'application côté service. Il n'a donc pas été possible de mettre le load balancer. Cependant, il serait possible de faire cela en modifiant légèrement l'architecture du programme afin de rendre indépendante l'exécution de chaque service.

2.3.3 Base de données MongoDB

Pour la construction de notre MVP, nous avons utilisé les bases de données MongoDB, et ceci pour plusieurs raisons.

Tout d'abord, il est possible de créer plusieurs base de données et de les lier à différents modèles présents dans le code source. C'est notamment ce système qui nous permet d'avoir une base de données pour chaque services de l'application.

De plus, elle est relativement simple d'utilisation et beaucoup plus légère à l'installation que peut l'être Microsoft SQL Server et MySQL. Le fait est, qu'il n'y pas d'interface graphique et tout se fait en invite de commande pour savoir ce qu'il se trouve exactement dans chaque base de données. C'est un peu moins ergonomique mais pas moins efficace et suffisant.

Chaque repository présenté dans la partie sur les microservice hérite de la classe "MongoRepository" qui nous permet d'utiliser toutes les fonctionnalités nécessaires au fonctionnement de notre MVP.

2.4 Exigences de documentation

Dans cette dernière partie, nous présenterons dans un premier temps le guide d'installation de notre MVP, puis un guide d'utilisation.

2.4.1 Guide d'installation

Vous pouvez trouver le guide d'installation du MVP sur le dépôt GitHub.

2.4.2 Guide d'utilisation

Vous pouvez trouver le guide d'utilisation du MVP sur le dépôt GitHub.

3 Preuve de concept

Dans cette partie, nous allons expliquer la façon dont nous avons réalisé notre preuve de concept en présentant les outils utilisés et les résultats obtenus. Nous présenterons également le rôle de l'architecture dans l'obtention de nos résultats.

3.1 Démonstration du nombre de transactions par seconde

Pour tester l'application, nous avons fait deux tests différents : le premier consiste à calculer le temps nécessaire pour insérer 4000 informations dans la base de données. Pour rappel, l'objectif fixé est de pouvoir faire 2000 opérations par seconde. Ce test nous a montré que l'on était capable de faire cette tâche en 7 secondes, soit 3.5 secondes pour 2000 opérations. Le deuxième test consiste à réaliser autant d'opérations mais cette fois on souhaite exécuter un cycle complet pour actualiser le solde de deux clients lorsqu'un virement est effectué. L'application a mis 50 secondes, soit 25 secondes pour 2000 opérations. Ces résultats peuvent sembler peu convaincants. Cependant, il faut prendre en compte qu'il n'y a qu'une seule instance de chaque service qui est sollicitée et il n'y a pas de load balancer dans notre architecture. De plus, il a été vu avec le Product Owner qu'au vu de l'environnement dans lequel les tests ont été réalisés, le résultat reste acceptable.

Il faut savoir que ces tests ont été effectués sur une seule machine en locale et que le processus de virement sollicite tous les services que nous avons implantés : le service d'identification du client, du solde, et de virement. Il faut récupérer le client payeur et le client receveur dans la base de données, aller récupérer leurs soldes respectifs, extraire le montant puis le modifier et ensuite sauvegarder cette nouvelle instance de solde pour chaque client avant d'enfin ajouter le dit virement dans la base de données. On pourrait penser à modifier le modèle de la classe solde en lui donnant en attribut l'adresse mail du client au lieu d'une référence de son compte et il en va de même pour le modèle de la classe virement. Cela améliorerait grandement la rapidité des opérations avec, comme dit précédemment, l'ajout d'un load balancer dans le système.

Ces performances seront grandement améliorées lorsque ces correctifs seront apportés. Une vidéo illustrant ces tests est disponible sur la page GitHub du projet.

4 Conclusion

Dans ce document, nous avons commencé par présenter en détail les exigences fonctionnelles et non fonctionnelles auxquelles notre MVP devait répondre. Ensuite, nous avons déroulé les exigences d'un point de vue développement et architectural sur comment nous avons construit notre MVP. Le but ici était évidemment de présenter comment nous l'avons développé, tout en respectant les technologies et architectures choisies lors de notre dernière présentation. Pour finir, nous avons montré une preuve de conception qui doit répondre aux demandes formulées par PrixBanque dans le tout premier document fourni. La demande principale étant le nombre de transactions par seconde où le système doit être en mesure de supporter environ 2000 opérations par seconde. Les résultats que nous avons obtenus sont loin de ce qui est demandé mais suffisant aux dires du Product Owner, de l'environnement et les conditions dans lesquels ont été réalisés les tests.