



UNIVERSITÉ DU QUÉBEC
À CHICOUTIMI

Département d'Informatique et de Mathématique

Projet pratique 3 – Rapport

**8INF853 – Architecture des applications
d'entreprise**

–

MVP et preuve de concept pour l'application bancaire PrixBanque

–

Yann COTINEAU
Brayan SILLIAU
Aurélien TRICARD

Trimestre d'automne 2022

Sommaire

Sommaire	2
1. Introduction	3
1.1 Contexte	3
1.2 Problématique	3
2. Description du MVP	4
2.1 Technologies utilisées	4
2.2 Les microservices	5
2.2.1 <i>RegistrationService</i>	5
2.2.2 <i>LoginService</i>	5
2.2.3 <i>BalanceService</i>	6
2.2.4 <i>PaymentService</i>	7
2.3 Communication entre microservices	7
2.4 API Gateway et <i>DiscoveryService</i>	8
2.5 Diagramme d'architecture général	10
3. Preuve de concept	10
3.1 Objectifs	10
3.2 Déroulement de l'expérimentation	11
3.3 Résultats	12
4. Conclusion	12
4.1 Bilan du projet pratique 3	12
4.2 Bilan du cours 8INF853	13

1. Introduction

1.1 Contexte

Ce projet s'inscrit dans le contexte de l'application bancaire "PrixBanque", une solution déployée exclusivement en ligne, qui doit à la fois traiter un volume important de données :

- 1.5 M de clients au bout d'une année
- 100 M de transactions par année
- 2000 transactions par seconde

mais qui doit également répondre à d'autres exigences non fonctionnelles, comme la mise à l'échelle, la haute disponibilité, l'évolutivité, la facilité d'utilisation, etc.

1.2 Problématique

Dans ce contexte, notre problématique est de concevoir puis d'implémenter une architecture MVP basée sur des microservices, et ainsi qu'une preuve de concept permettant de répondre à l'exigence non-fonctionnelle suivante :

2000 transactions par seconde

Nous détaillons ainsi dans les parties suivantes les choix que nous avons fait en termes d'architecture du MVP, et comment ils nous permettent d'essayer de répondre à cette exigence à travers la preuve de concept.

De plus, les exigences fonctionnelles de l'application que nous allons construire sont détaillées dans la figure 1.

Exigence fonctionnelle	Priorité
Créer un compte	Moyenne
Se connecter	Basse
Consulter le relevé actuel	Moyenne
Envoyer un virement à un autre client	Haute

Figure 1 : Résumé des exigences fonctionnelles de l'application du PP3

Comme c'est indiqué dans le sujet, nous verrons par la suite que à chacune de ces exigences fonctionnelles correspond un microservice différent.

2. Description du MVP

Nous détaillons dans cette section les différentes composantes de l'architecture MVP que nous avons construite pour ce projet.

2.1 Technologies utilisées

Le présent projet est constitué de 3 composantes principales : une base de données pour stocker les informations utilisateur ; des microservices qui gèrent les requêtes et l'authentification ; un frontend qui permet d'accéder à l'application. Les technologies suivantes sont utilisées :

- base de données MongoDB : un système de gestion de bases de données MongoDB sur le port par défaut 27017 permet de stocker les informations utilisateurs. Elle contient pour ce projet une base de données appelée "**pp3-db**", laquelle contient une collection appelée "**users**". Dans cette collection, un utilisateur est décrit par un nom d'utilisateur, un mot de passe, ainsi qu'un solde de compte en \$ (il y a également un identifiant généré automatiquement par MongoDB, que nous n'utilisons pas). Un exemple est décrit dans la figure 2.

username (String)	password (String)	balance (long)
yann	yann12345	4269
brayan	brayan12345	1515
aurelien	aurelien12345	2023
...

Figure 2 : Aperçu de la collection "**users**" de la base de données "**pp3-db**"

- microservices : les microservices sont tous des projets individuels Maven utilisant le framework Spring Boot.
- frontend : le frontend de l'application permet aux utilisateurs d'accéder à une interface pour être en mesure de faire les tâches décrites dans les exigences fonctionnelles (créer un compte, etc.). Il est codé dans de simples fichiers en .html à l'aide d'HTML et de Javascript : l'utilisation de la méthode fetch() du Javascript permet alors d'envoyer des requêtes vers les microservices.
- Le projet utilise le contrôle de version avec Git, qui permet de centraliser tout le code source. Le projet est en accès libre à l'adresse suivante :
<https://github.com/yanncotineau/pp3>
Les instructions de déploiement sont disponibles dans le README.

2.2 Les microservices

Nous détaillons dans cette section les 4 microservices principaux des exigences fonctionnelles de l'application. A chacun de ces microservices sont associés des endpoints de l'API de l'application. Dans la suite des explications, chaque endpoint de l'API sera présenté de la manière suivante :

Description de ce que fait l'endpoint

TYPE (GET, POST, etc.) /api/endpoint : **Type de retour**

Paramètres dans l'URL

Body de la requête

-> ce que l'endpoint retourne

2.2.1 RegistrationService

Ce microservice s'occupe de l'inscription des utilisateurs. Il comprend 1 endpoint qui permet aux utilisateurs, en spécifiant un nom d'utilisateur et un mot de passe, de créer un compte avec ces informations :

Créer un compte avec des identifiants donnés :

POST /api/registration : **boolean**

Paramètres : aucun

Body : un JSON de la forme : **{"username": "test", "password": "test"}**

-> retourne true si l'inscription s'est bien passée, false sinon.

Bien évidemment, on ne peut pas créer un compte avec un nom d'utilisateur déjà enregistré dans la base de données. Lors de la création de compte, le champ **"balance"** de la base de données est arbitrairement initialisé à \$1000.

2.2.2 LoginService

Ce microservice permet à la fois la connexion des utilisateurs et leur authentification :

- La connexion permet de générer un token de connexion JWT à partir d'identifiants donnés, pourvu qu'ils correspondent à un utilisateur déjà inscrit.
- L'authentification, au contraire, permet, étant donné un token de connexion JWT, de déchiffrer ce token et d'obtenir le nom d'utilisateur associé à ce token de connexion.

Ce microservice comprend donc 2 endpoints pour chacune de ces tâches :

Se connecter avec des identifiants donnés (obtenir un token de connexion) :

POST /api/login/tokenFromUser : **String**

Paramètres : aucun

Body : un JSON de la forme : **{"username": "test", "password": "test"}**

-> retourne un token de connexion si la connexion s'est bien passée, et la chaîne vide "" sinon.

Bien évidemment, il faut que le couple nom d'utilisateur/mot de passe fourni corresponde effectivement à un compte déjà existant pour que la requête soit un succès. Dans notre frontend, le token de connexion reçu après avoir fait cette requête est stocké dans le **localStorage** du navigateur. Puis, nous verrons par la suite qu'il sera fourni en paramètre de toutes les requêtes portant sur les informations utilisateur (obtenir le solde du compte, effectuer un virement, etc.) pour permettre l'authentification de l'utilisateur effectuant la requête.

S'authentifier (déchiffrer un token de connexion) :

GET /api/login/userFromToken?token={token} : **String**

Paramètres : un string "token", le token de connexion précédemment généré lors de la connexion

Body : aucun

-> retourne le nom d'utilisateur du compte associé au token si l'authentification s'est bien passée et la chaîne vide "" sinon.

Le token JWT fourni doit être valide car il est déchiffré de la même manière qu'il avait été généré (même algorithme de chiffrement, même code secret, etc.)

2.2.3 BalanceService

Ce microservice permet d'obtenir le solde d'un compte correspondant à un utilisateur donné. Il comprend 1 endpoint qui permet aux utilisateurs, en spécifiant un token de connexion, d'obtenir en temps réel le solde de leur compte :

Obtenir son solde (en donnant simplement son token de connexion) :

GET /api/balance?token={token} : **Map<String,Object>**

Paramètres : un string "token", le token de connexion précédemment généré lors de la connexion

Body : aucun

-> retourne un objet JSON de la forme **{"username": "test", "balance": 1000}**

(username correspond au nom d'utilisateur du compte dont on a obtenu le solde et balance correspond au solde du compte en \$) en cas de succès et un objet JSON vide {} sinon.

Bien évidemment, le token de connexion fourni doit être valide.

2.2.4 PaymentService

Ce microservice permet de faire un virement de son propre compte à un compte donné. Il comprend 1 endpoint qui permet aux utilisateurs, en spécifiant un token de connexion, un nom d'utilisateur receveur et un montant en \$, d'effectuer le virement et ainsi mettre à jour les soldes des 2 comptes impliqués dans la transaction.

Faire un virement :

POST /api/payment : Map<String,Object>

Paramètres : aucun

Body : un JSON de la forme : {"token": "3fQS0kT66", "receiver": "test", "amount": 1000}

-> retourne un objet JSON de la forme {"username": "test", "balance": 1000}
(username correspond au nom d'utilisateur du compte qui a effectué le virement
balance correspond au nouveau solde du compte en \$ après la transaction) en cas de succès et un objet JSON vide {} sinon.

Bien évidemment, plusieurs conditions doivent être remplies pour que le virement se passe bien :

- le token de connexion fourni doit être valide
- le nom d'utilisateur du receveur doit correspondre à un compte qui existe véritablement
- le nom d'utilisateur du receveur ne peut être le même que celui de l'utilisateur qui effectue le virement (on ne peut s'envoyer de l'argent à soi-même)
- le montant du virement doit être un nombre positif
- le montant du virement doit être inférieur au solde du compte qui effectue le virement avant la transaction (on ne peut pas payer plus que ce que l'on a)

2.3 Communication entre microservices

Ces microservices que nous avons présenté ont également des connexions entre eux. Par exemple, considérons l'endpoint **GET** /api/balance?token={token} qui permet d'obtenir le solde d'un compte en donnant simplement un token de connexion pour ce compte : pour savoir, dans la base de données, pour quel compte il faut récupérer le solde, cet endpoint envoie en réalité une requête au microservice **LoginService**, en fournissant le token de connexion qu'il a reçu, et reçoit ainsi le nom d'utilisateur associé, comme illustré dans le code Java de cet endpoint ci-dessous :

```
String username = webClientBuilder.build().get()
    .uri("http://login-service/api/login/userFromToken?token="
+ token)
    .retrieve().bodyToMono(String.class).block();
```

Figure 3 : Implémentation en java de la communication entre services

On retrouve par ailleurs cette même communication entre microservices pour l'endpoint permettant de faire des virements. Cette communication entre microservices permet ainsi d'éviter de réécrire plusieurs fois le même code et également de bien séparer chaque niveau de l'application (ici, la couche sécurité et la couche métier)

2.4 API Gateway et DiscoveryService

Chacun des microservices présentés précédemment sont donc des projets Spring Boot totalement indépendants. Il nous est possible de lancer plusieurs instances de chaque microservice : chacune de ces instances sera alors déployée sur un port aléatoire de la machine, comme ce qui est indiqué, pour chaque microservice, dans son fichier de configuration **application.properties** :

```
server.port = 0
```

Cette ligne permet de ne pas spécifier un port précis, mais bien de laisser la machine choisir un port aléatoire sur lequel déployer l'instance.

Ce caractère aléatoire des ports est mis en évidence lorsque l'on lance les microservices depuis un IDE, comme IntelliJ :

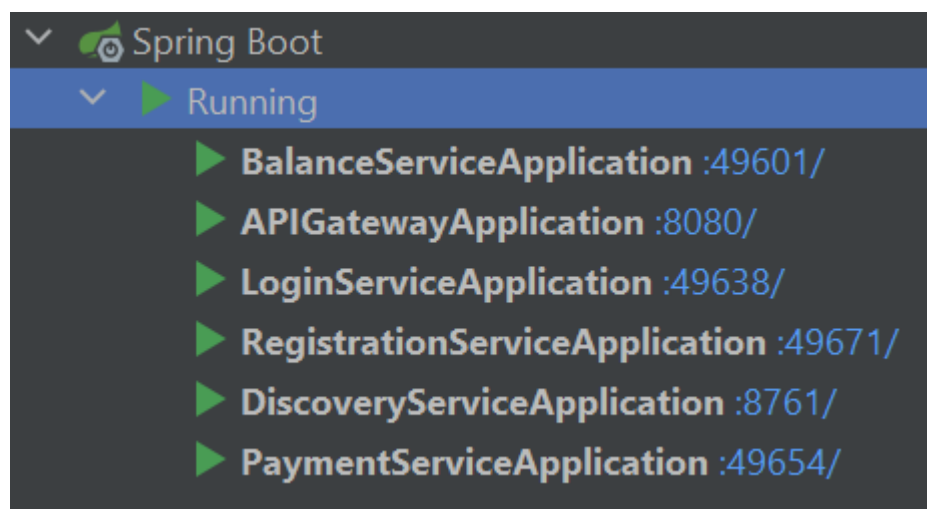


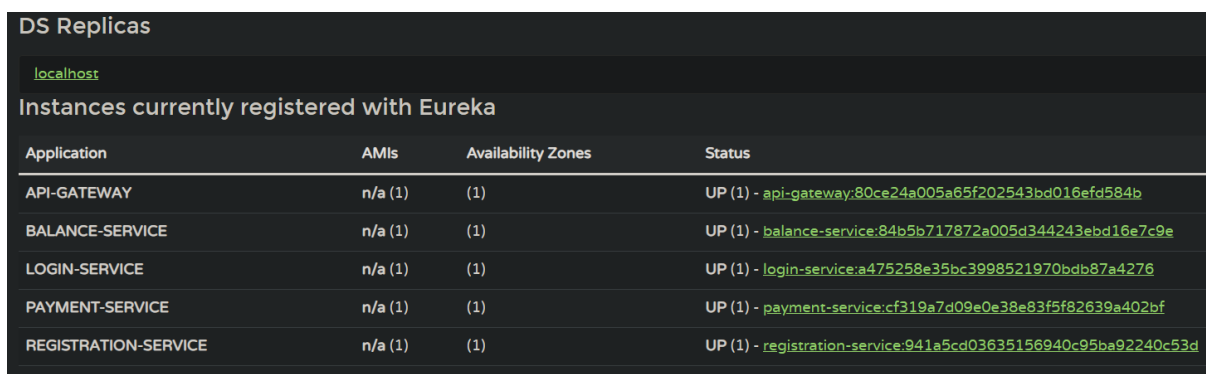
Figure 4 : Démonstration du caractère aléatoire des ports des services

Ainsi, puisque les microservices sont déployés sur des ports aléatoires mais qu'ils doivent néanmoins communiquer entre eux, ils ne peuvent pas savoir a priori sur quels

ports se trouvent ces autres services, d'où l'utilité d'un service de découverte des ports, le **DiscoveryService** : c'est un serveur qui recense toutes les instances de tous les microservices de l'application, avec leur port. Pour simplifier, dès qu'une instance est créée, elle envoie son port à ce service qui le stocke dans une table ; et dès qu'un service veut communiquer avec un autre, il demande au DiscoveryService le port de cet autre service.

Dans notre application, nous utilisons Spring Cloud Eureka, qui permet également de garantir la répartition de charges, à l'aide de l'algorithme dit "Round Robin", lorsqu'un service A doit communiquer avec un service B dont il existe plusieurs instances.

Sur le port 8761, Spring Cloud Eureka fournit de plus une interface utilisateur permettant de suivre, en temps réel, la liste des services déployés et le nombre de leurs instances

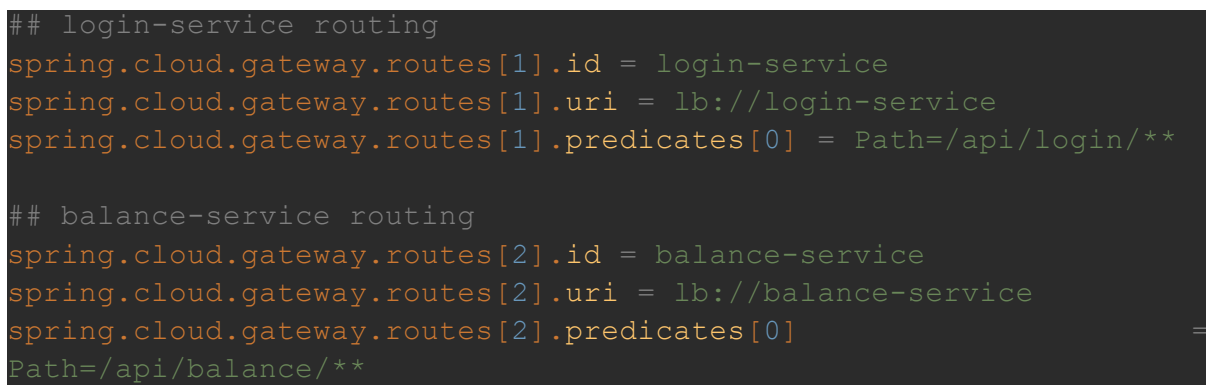


The screenshot shows the 'DS Replicas' interface of Spring Cloud Eureka. It displays a table titled 'Instances currently registered with Eureka' with the following columns: Application, AMIs, Availability Zones, and Status. The table lists five services: API-GATEWAY, BALANCE-SERVICE, LOGIN-SERVICE, PAYMENT-SERVICE, and REGISTRATION-SERVICE. Each service has one instance (n/a (1)) in one availability zone (1) and is in an 'UP' status. The status column also includes a unique ID for each instance.

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - api-gateway:80ce24a005a65f202543bd016efd584b
BALANCE-SERVICE	n/a (1)	(1)	UP (1) - balance-service:84b5b717872a005d344243ebd16e7c9e
LOGIN-SERVICE	n/a (1)	(1)	UP (1) - login-service:a475258e35bc3998521970bdb87a4276
PAYMENT-SERVICE	n/a (1)	(1)	UP (1) - payment-service:cf319a7d09e0e38e83f5f82639a402bf
REGISTRATION-SERVICE	n/a (1)	(1)	UP (1) - registration-service:941a5cd03635156940c95ba92240c53d

Figure 5 : Interface de suivi des services de Spring Cloud Eureka

Enfin, nous utilisons un dernier service qui est une simple API Gateway, déployée sur le port 8080, qui permet d'avoir un seul point d'entrée pour l'API, ce qui facilite la consommation de l'API dans le frontend. Cette gateway ne fait que faire transiter les requêtes vers le bon service à l'aide de filtres, dont des exemples sont présentés ci-dessous :



```
## login-service routing
spring.cloud.gateway.routes[1].id = login-service
spring.cloud.gateway.routes[1].uri = lb://login-service
spring.cloud.gateway.routes[1].predicates[0] = Path=/api/login/**

## balance-service routing
spring.cloud.gateway.routes[2].id = balance-service
spring.cloud.gateway.routes[2].uri = lb://balance-service
spring.cloud.gateway.routes[2].predicates[0] = Path=/api/balance/**
```

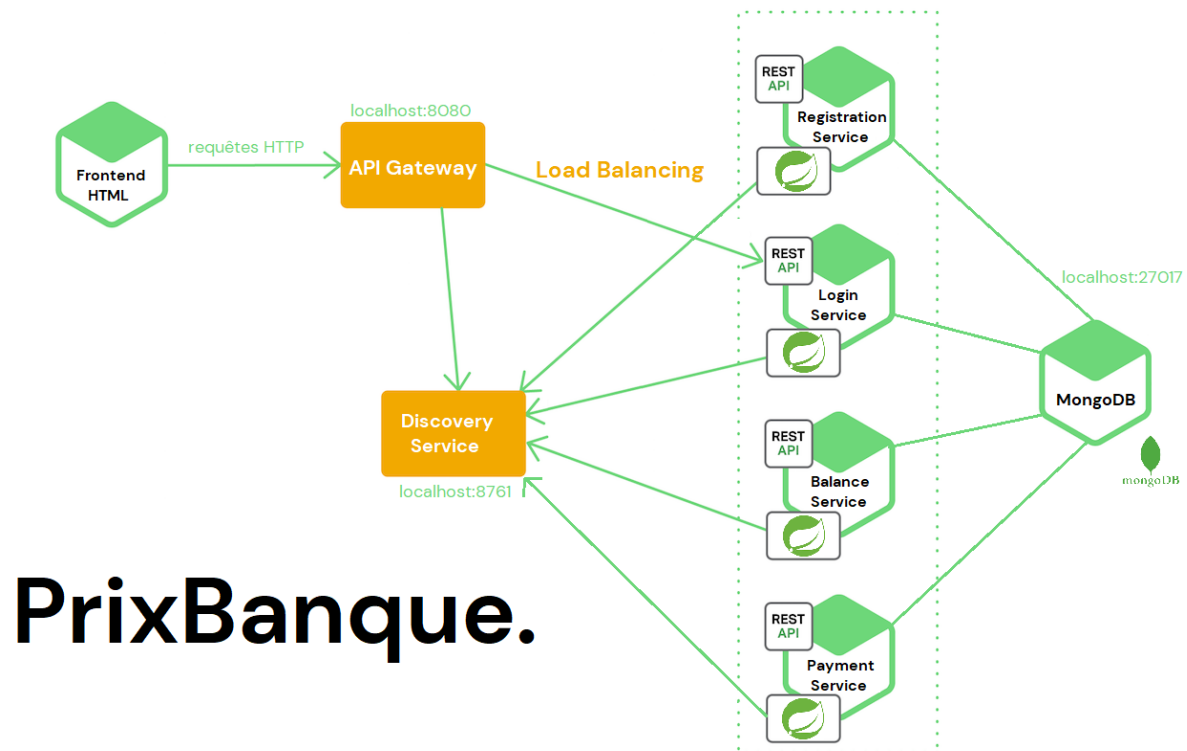
Figure 6 : Implémentation dans le code des filtres de l'API Gateway

Intuitivement, toutes les requêtes dont l'URL est de la forme /api/login/** seront redirigés vers une instance de **LoginService**. Cette gateway garantit ainsi la répartition

de charges s'il existe plusieurs instances d'un service donné, à l'aide de l'algorithme dit "Round Robin".

2.5 Diagramme d'architecture général

Un diagramme résumant l'architecture générale de l'application que nous avons construite est présentée ci-dessous.



PrixBanque.

Figure 7 : Diagramme de déploiement général de l'application

3. Preuve de concept

Cette section détaille les objectifs, le déroulement et les résultats de la preuve de concept de ce projet, portant sur le nombre de transactions par seconde.

3.1 Objectifs

Il s'agit dans cette preuve de concept d'essayer d'approcher 2000 transactions par seconde sur l'application.

3.2 Déroulement de l'expérimentation

Pour faire cette expérimentation, il nous faut trouver un moyen de compter le nombre de requêtes réalisées en un temps donné. Pour cela, nous avons établi le protocole suivant :

- nous considérons un temps d'expérience donné, disons 10 secondes.
- Pendant ces 10 secondes, alors que nous sommes connectés à un compte, nous envoyons, plusieurs milliers de fois par seconde, des requêtes pour effectuer un virement d'un montant de 1\$ vers un autre compte donné.
- A la fin des 10 secondes, la différence entre le montant initial et le montant final nous renseignera alors directement sur le nombre de requêtes qui ont été correctement effectuées.
- En divisant ce nombre de requêtes par la durée de l'expérimentation, nous pouvons alors obtenir un nombre de requêtes par seconde.
- Enfin, nous reproduisons cette expérience plusieurs fois (pour obtenir une moyenne), et ce, pour plusieurs nombres d'instances de **LoginService** et **PaymentService** (les 2 services les plus sollicités lors de cette expérimentation)

L'image suivante montre à quoi ressemble l'interface utilisateur sur le frontend lorsqu'on est connecté à un compte. Cela nous montre comment il est possible de lancer une expérience depuis le frontend. Il est possible de préciser la durée de l'expérimentation puis, à la fin, ses résultats s'affichent directement.

The screenshot displays a web interface for a benchmarking tool. At the top, there is a 'Se déconnecter' button. Below it, the heading 'Page d'accueil' is followed by a personalized greeting 'Bienvenue yann !'. The user's balance is shown as 'Votre solde : \$999999171781'. There are two input fields: 'Montant (en \$):' and 'Destinataire :', each with a corresponding 'Submit' button. Below these, there is a 'Durée du benchmark (en s):' input field with the value '10' and a 'Lancer le benchmark' button. At the bottom, the 'Résultats du benchmark :' are displayed, showing 'Temps écoulé : 10s', 'Nombre de requêtes effectuées : 7265', and 'Requêtes par seconde : 726.5'.

Se déconnecter

Page d'accueil

Bienvenue yann !

Votre solde : \$999999171781

Montant (en \$):

Destinataire :

Submit

Durée du benchmark (en s):

Lancer le benchmark

Résultats du benchmark :
Temps écoulé : 10s
Nombre de requêtes effectuées : 7265
Requêtes par seconde : 726.5

Figure 8 : Interface utilisateur du frontend

3.3 Résultats

Nos expériences montrent qu'avec 3 instances de **LoginService** et **PaymentService**, nous sommes en mesure d'obtenir entre **600 et 800 requêtes par seconde**. Cela est bien inférieur au 2000 souhaités, mais cela nous semble relativement correct.

L'ajout d'instances supplémentaires (nous avons essayé avec jusqu'à 10 instances) ne semble pas amener à des résultats significativement meilleurs : cela peut être s'expliquer par la limitation de nos machines, notamment en termes de RAM (déployer une vingtaine de services est en effet très consommateur en RAM), mais également en termes de vitesse de lectures/écritures sur le disque (il est possible que MongoDB ne puisse tout simplement pas gérer davantage de requêtes que cela).

Puisque toutes les expériences ont été conduites en local sur la même machine, il est également à noter que le navigateur utilisé pour conduire ces expériences semble avoir un impact très fort sur les performances.

Pour résoudre ce problème, nous aurions aimé pouvoir déployer notre application et y accéder à partir de plusieurs machines, mais nous n'avons cependant pas pu le faire par manque de temps.

Toutefois, ces résultats (**entre 600 et 800 opérations par seconde**) sont satisfaisants et démontrent la réelle pertinence d'une architecture microservices dans ce type d'application bancaire.

4. Conclusion

4.1 Bilan du projet pratique 3

En conclusion, ce projet nous a permis de nous familiariser de manière très concrète avec le concept d'architectures microservices, que nous avons déjà tous rencontré par le passé mais qui nous paraissait très abstrait. Maintenant, nous avons une vue d'ensemble d'un projet avec une architecture microservices :

- Nous comprenons sa structure, les éléments qui la composent et pourquoi ils sont pertinents dans ce type d'architecture (API gateway, DiscoveryService, avoir plusieurs instances de chaque service, la gestion de la communication entre services, etc.)
- Nous comprenons l'avantage de ce type d'architecture par rapport à une approche plus monolithique : meilleure séparation des composants de l'application, scalabilité, durabilité, etc.
- Nous avons découvert des technologies permettant de mettre en place cette architecture, notamment Spring Cloud Eureka, qui sauront nous être utiles dans notre vie professionnelle future.

Nous avons également apprécié la liberté qui nous a été laissée durant ce projet en ce qui concerne l'architecture, la réalisation de la preuve de concept, etc. Cela nous a permis de découvrir tous ces concepts par nous même et ainsi de nous les approprier de manière plus efficace.

4.2 Bilan du cours 8INF853

Yann : Ce cours m'a permis de me rendre compte que le développement logiciel doit être structuré pour être efficace en termes de temps et d'argent. Cela passe par de nombreux processus, architectures et outils pour simplifier et accélérer le développement d'applications dont je n'avais pas nécessairement connaissance. Tous ces outils me permettront, dans ma vie professionnelle, de développer des applications avec une vue plus globale.

Aurélien : J'avais déjà une bonne approche du développement logiciel mais le cours m'a permis d'apprendre de nouveaux aspects du développement logiciel. Il m'a également permis de nouvelles technologies et de revoir certains aspects de la conception de logiciel.

Pour finir, ce cours m'a permis de confirmer mon choix dans mon parcours professionnel.

Brayan : Ce cours m'a permis de mieux comprendre les différentes étapes et processus nécessaires pour développer une application de manière efficace. J'ai appris de nouvelles techniques et technologies qui me seront utiles dans mon parcours professionnel. Je suis maintenant mieux préparé pour affronter les défis qui se présenteront dans mon travail en tant que développeur logiciel. Je suis également plus confiant dans mon choix de carrière et je sais que ce cours m'aidera à atteindre mes objectifs professionnels.