


République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université AMO de Bouira

Faculté des Sciences et des Sciences Appliquées

Département d'Informatique

Mémoire de Master

en Informatique

Spécialité : *Génie des Systèmes Informatiques*

Thème

Gestion des données pour les architectures microservises

Encadré par

— Mahfoud Zohra

Réalisé par

— Madani Amar

— Djaid Sami Mohammed Tahar

2021/2022

Remerciements

Nous remercions tout d'abord ***ALLAH****, le tout puissant de nous avoir donné la patience, la santé et la volonté pour réaliser ce mémoire.*

Un grand merci à nos chers parents et nos familles qui par leurs prières et leurs encouragements, on a pu surmonter tous les obstacles.

Nous tenons à adresser nos sincères remerciements et le grand respect à notre encadreur ***Mme. Mahfoud Zohra et Mme Aid Aicha*** *pour leur disponibilité, leur conseils, leur gentillesse et toute l'aide qu'ils nous ont rapportés.*

Nous adressons toutes nos sympathies à tous nos collègues et nos amis pour leurs encouragements et pour tous les moments agréables qu'on passés ensemble.
Nos plus vifs remerciements vont également à nos collègues de la spécialité Master Génie des Systèmes Informatiques et nos enseignants qui nous ont aidés durant notre parcours.

Un grand merci pour tous ceux qui ont contribuées de près ou de loin pour la réalisation de ce mémoire.

Dédicaces

travail est dédié à :

Nos familles, nos parents, nos frères, nos sœurs qui sont pour nous les personnes les plus précieuses.

Nos amis et personnes spéciales, Touati Djamel, Hamza, Salaheddine, Anis, Yahia,Hicham, Walid, Tarek, Amine, Zaki, Oualid, Kamel, Hamza, Abdou,Zahra.

Table des matières

Table des figures

Liste des abréviations

ACID : Atomicity, consistency, isolation, durability

API : Interface de programmation d'applications

AMQP : Advenced message Queuing protocol

BPEL : Business Process Execution Language

CQRS : Command and Query Responsibility Segregation

CI : continiuous integration

CD : continiuous delivery

CM : continuous monitoring

DevOPS : développement et opérations

EDA : Event-driven architecture

ESB : un bus de service d'enterprise

HTTP : Protocole de transfert hypertexte

IDC : International data corporation

ORM : Object-relational Mappers

MSA : microservices architecture

MQTT : Message queuing Telemetry transport

Rest : transfert d'état représentatif

Paas : platform as a service

SOA : architecture orientée service

TM : Transaction manager

UI : utiliseur interface

XML : Extensible Markup Language

2PC : two phase commit

2PL : two phase locking

3PC : three phase commit

Introduction générale

Les technologies logicielles ne cessent d'évoluer pour faciliter le développement, le déploiement et la maintenance d'applications dans différents domaines. En parallèle, ces applications évoluent en continu, avec la croissance potentielle des exigences et des besoins des clients, pour garantir une bonne qualité de service qui deviennent de plus en plus complexe. Cette évolution implique souvent des coûts de développement et de maintenance plus élevés, Réduire ces coûts et améliorer la qualité de ces applications sont actuellement des objectifs centraux du domaine du génie logiciel. Apparus il y a quelques années, une tendance à l'accélération, sous le nom des microservices au tant qu'exemple de technologie ou styles architectural favorisant d'atteindre ces objectifs. En effet, l'approche des microservices a été inventé pour résoudre certaines difficultés, notamment une augmentation de l'évolutivité, de la flexibilité et l'agilité. En remplaçant ainsi l'approche par défaut adopté dans le développement des applications, connue sous le nom « approche monolithique ». Le microservice alors est un style architectural d'applications logicielles en tant que ensemble de services modulables indépendants, dans lesquels chaque service exécute un processus qu'il communique à travers un mécanisme défini au préalable. Dans le domaine de la programmation par exemple, on forme des petites équipes qui ne s'occupent que d'un seul service. Et puis, dans le

domaine du management du projet, on se concentre sur l'équipe et son indépendance, c'est-à-dire au lieu d'avoir une administration centralisée, chaque équipe est entièrement responsable de son produit final. De ce fait, de nombreuse entreprise considèrent que suivre ce style est le moyen le plus efficace de développer leurs activités, en basculant leurs processus de développement vers cette nouvelle architecture.

Ce style architectural propose des nouvelles exigences concernant la gestion des bases de données .

les données sont décentralisées et distribuées entre les microservices constitutifs. Ceci contraste avec l'architecture d'application traditionnelle et la centralisation des données, généralement dans une base de données relationnelle. Cela pose un certain nombre de problèmes. Chaque microservice est une solution à une capacité opérationnelle dans un sous-domaine et fonctionne avec un modèle de données conceptuel pour ce sous-domaine. De plus, avec la décentralisation de la gouvernance, la conception et le développement du microservice sont confiés à une équipe. Une conséquence de la décentralisation serait l'absence d'un modèle de données unifié pour le système. Outre les différents modèles de données, Microservices peut opter pour différents systèmes de stockage de données, y compris des systèmes de bases de données relationnelles, des systèmes de fichiers, etc. La décentralisation des décisions et de la gestion des données a ses implications. Les bases de données distribuées utilisent les transactions pour gérer les mises à jour.

De plus, avec la décentralisation de la gouvernance, la conception et le développement du microservice sont confiés à une équipe. Une conséquence de la décentralisation serait l'absence d'un modèle de données unifié pour le système. Outre les différents modèles de données, Microservices peut opter

pour différents systèmes de stockage de données, y compris des systèmes de bases de données relationnelles, des systèmes de fichiers, etc. La décentralisation des décisions et de la gestion des données a ses implications. Les bases de données distribuées utilisent les transactions pour gérer les mises à jour.

Dans le cadre de ce mémoire, nous étudions la gestion des bases des données dans le cadre des microservices. Et nous explorons le développement d'applications dans ce domaine à travers un exemple de notre application. Nous effectuant ainsi des tests de performances.

Chapitre 1

Etude Préliminaire

1.1 Introduction

Dans ce premier chapitre, Nous définissons quelques types d'l'architecture logicielle, l'architecture microservice, ses avantages et inconvénients, et ses caractéristiques . Puis nous présenterons l'approche de développement DevOPS (developpeurs et opérationels) .

1.2 Les architectures monolithiques

Les architectures monolithiques regroupent toutes les fonctionnalités d'une application dans un seul code monolithique, les modules ne s'exécutent individuellement. Dans cette technique, toute la logique de traitement des demandes est encapsulée dans un seul processus étroitement lié. On utilise les capacités de base du langage, pour diviser le programme en classes, fonctions et espaces de noms. Et pour soigneusement exécuter et tester l'application sur l'ordinateur portable d'un développeur,[1] On utilise un processus de déploiement pour assurer que les changements sont bien testés et mis en production. on peut mettre le monolithique à l'échelle horizontale en exécutant de nombreuses instances derrière un équilibreur de charge [2].

Cette architecture a fonctionné avec succès pendant de nombreuses décennies. Ce qui fait que de nombreuses entreprises sont toujours entrain de l'utiliser. Mais avec l'évolution des besoins de les entreprises et l'émergence de nouvelles technologies, les applications monolithiques ont eu de sérieux problèmes. [2, 3] :

— Il peut être difficile de comprendre et de modifier le programme. Par conséquent, les progrès sont habituellement ralentis.

— Le déploiement continu est difficile ; un changement apporté à une section mineure de l'application nécessite la reconstruction et le déploiement de l'application complète.

— Le développement de l'application utilise une seule technologie et elle est fixée avant de commencer.

— Un dysfonctionnement sur une partie du système impacte tout le système.

la figure 1.1 présente un exemple d'application monolithique de commerce électronique.[4]

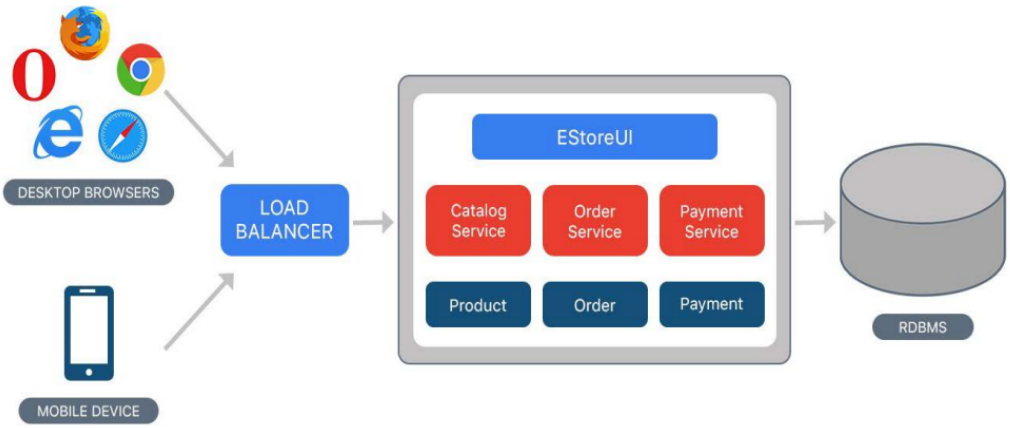

FIGURE 1.1 – Exemple d'application monolithique

Malgré la mise en œuvre de plusieurs fonctionnalités de l'ensemble du logiciel, l'application est déployée comme un seul programme autonome.

1.3 Les architectures orientées services (SOA)

Les problèmes rencontrés avec l'architecture monolithique ainsi que les problématiques d'interopérabilité concernant les technologies informatiques utilisées en entreprise ont posé des défis. Ce qui a donné naissance aux architectures orientées services (AOS ou SOA). Cette type d'architecture est considérée comme un modèle d'intéraction applicative distribuée qui expose des composants logiciels (services). Elle suit le principe de fournisseur et consommateur de service. L'adoption des principes SOA permet de décomposer des systèmes complexes et monolithiques en des applications composées d'un écosystème de composants plus simples et bien définis. L'utilisation d'interfaces communes et des protocoles standards donne une vue horizontale d'un système d'entreprise (Atzori et al., 2010). Cette architecture utilise des formats d'échange (XML ou JSON) et une couche d'interface interopérable connue sous le nom de service web.[5]

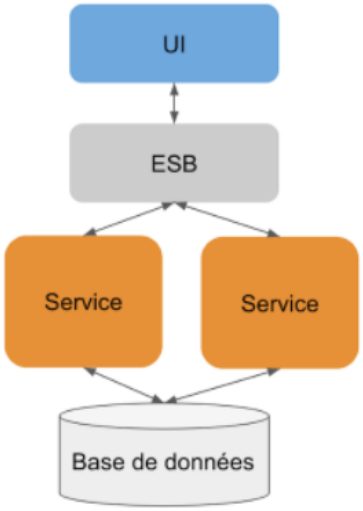

FIGURE 1.2 – Exemple d'une Architecture orientée service [6]

Un bus de service d'entreprise(ESB) : est une infrastructure d'intégration distribuée basée sur des normes ouvertes, des messages et qui fournit des

services de routage, d'invocation et de médiation pour faciliter les intéractions d'applications et de services distribués disparates de manière sécurisée et fiable. [6]

1.4 Les microservices

Les microservices est un style d'architecture inspiré du SOA qui met l'accent sur la division du système en petits services légers qui sont conçus expressément pour exécuter une fonction d'affaires très reliées, et est une évolution du style d'architecture traditionnel axé sur les services [5]. un microservice est défini dans [7] comme « le processus indépendant minimal qui interagit par messagerie ». Une architecture microservice est également définie comme « une application distribuée où tous ses modules sont des microservices ». Les avantages communément acceptés de ce style comprennent : l'augmentation de l'agilité, la productivité des développeurs, la résilience, l'évolutivité, la fiabilité, la maintenabilité, la séparation des préoccupations et la facilité de déploiement. Cependant, ces avantages s'accompagnent des défis, comme la découverte de services sur le réseau, la gestion de la sécurité, l'optimisation des communications, le partage des données et la performance. Lorsqu'ils sont abordés de façon appropriée, cependant, ces défis permettent à un système de bénéficier de la plupart des avantages susmentionnés [8].

L'architecture de microservices (MSA) a acquis une immense popularité au cours des dernières années. L'International Data Corporation (IDC) a prédit à la fin de 2021, 80 % des logiciels cloud seront développés à l'aide du style MSA [9]. les principaux des conseils en logiciels et des sociétés de conception de produits ont constaté que l'approche des microservices est une architecture attirante qui permet aux équipes et aux organisations de logi-

ciels d'être plus productives en général, et de construire des produits logiciels souvent plus réussis. De nombreuses organisations en dehors des entreprises de logiciels traditionnels ont également essayé et testé ce type d'architectures et ont trouvé cela très bénéfique. Les microservices sont également considérés comme une architecture appropriée pour les systèmes déployés sur les infrastructures cloud, car ils peuvent tirer parti de l'élasticité et des fonctionnalités de provisionnement à la demande du modèle cloud. Des entreprises comme Netflix et SoundCloud ont adopté le style des microservices dans le nuage et en ont tiré de nombreux avantages [10, 11] .

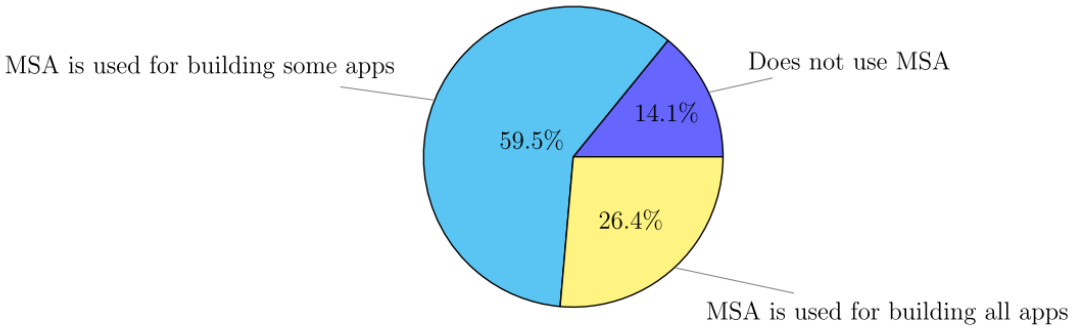

FIGURE 1.3 – Statut de l'adoption de MSA dans un sondage[9]

Selon une étude [9] concernant l'adoption de MSA dans un ensemble de organisation : la figure 1.3 représente 59,5% (63 sur 106) des développeurs ont déclaré qu'ils n'utilisaient l'ASM que pour développer des applications spécifiques dans leurs organisations respectives. 26,4% (28 sur 106) des répondants ont répondu qu'ils avaient utilisé MSA pour développer toutes les applications de leur organisation. Dans l'ensemble, près de 86% des répondants proviennent d'organisations qui ont adopté l'ASM pour élaborer des applications. Il est à noter que même si 14,1% (15 sur 106) des répondants ont répondu que leur organisation n'utilisait pas MSA pour élaborer des applications, ces répondants ont encore de l'expérience et de l'expertise avec

les systèmes fondés sur l'ASM, ce qui est l'un des critères pour répondre à ce sondage.

La figure 1.4 présente un exemple d'un application de e-commerce basée sur Microservice, qui fournit une logique d'affaires de commerce électronique, cet exemple est équivalente à la version monolithique

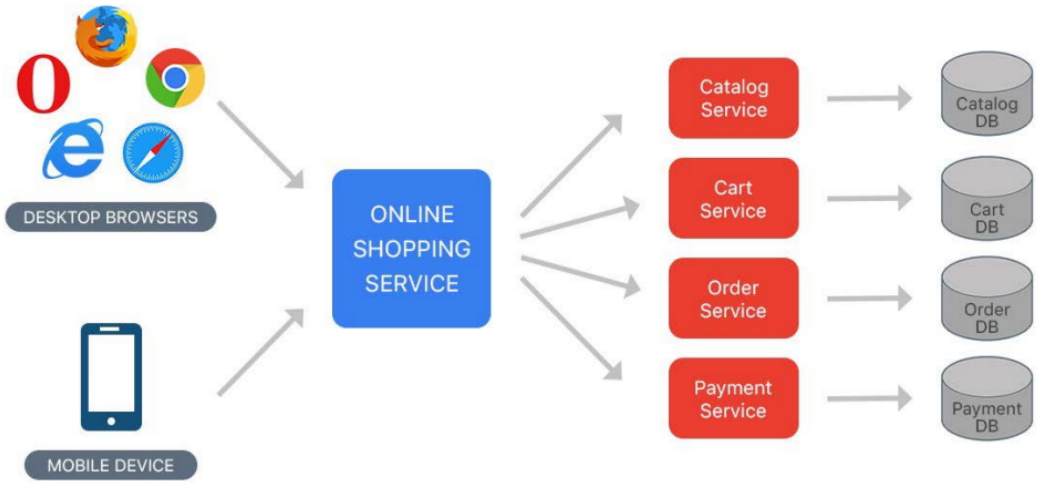

FIGURE 1.4 – Flux microservice sur l'exemple de l'application e-commerce [12]

L'application est composée de quatre microservices de base, qui fournissent la logique d'affaires et en raison d'un service supplémentaire elle expose les fonctionnalités comme si c'était une seule application.

1.5 L'architecture d'un microservice

L'architecture microservices développe une application comme un ensemble de petits services.[12] Chaque service fonctionne moyennant son propre processus qui communique avec des mécanismes légers. Les services sont développés autour des fonctonalitées métiers qui sont déployés d'une façon indépendante par un processus automatisé. Ils sont autonomes et isolés mais aussi ils peuvent communiquer entre eux pour pourvoir créer l'ensemble des fonctionnalités nécessaires.

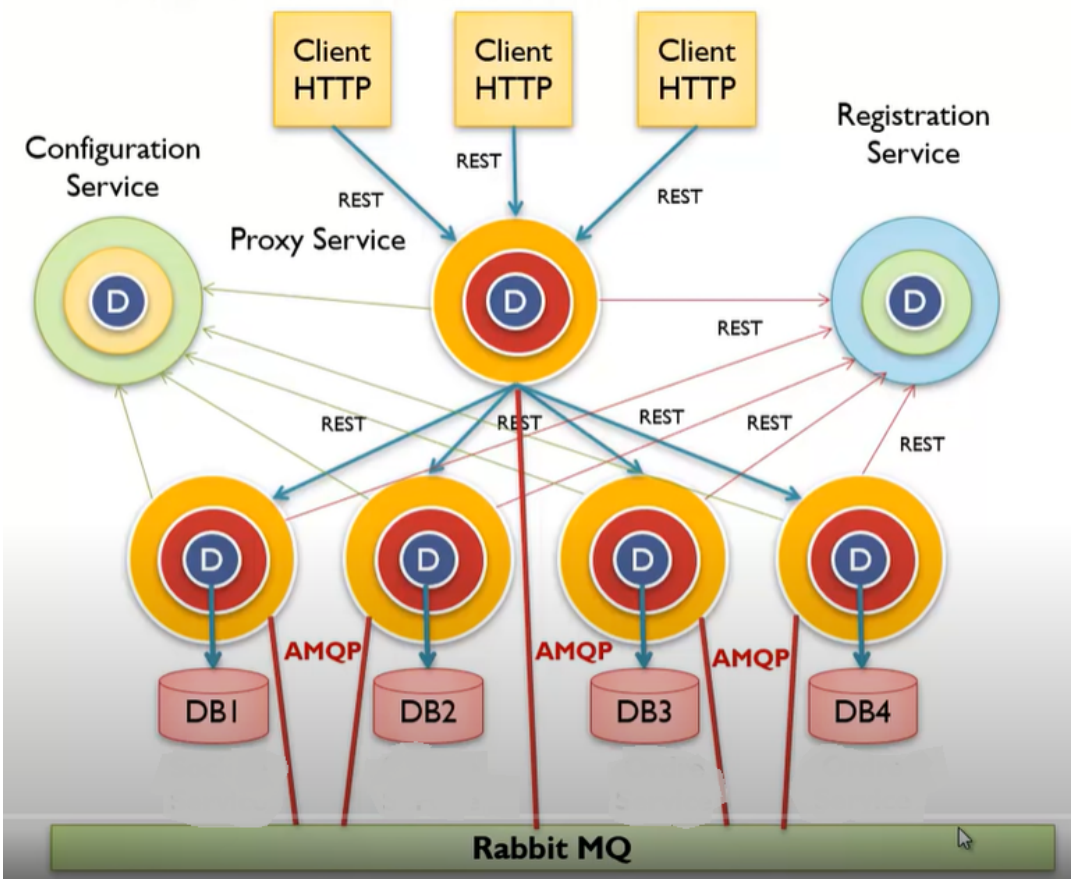

FIGURE 1.5 – Architecture de microservices

Les microservices sont généralement gérés par des petites équipes avec suffisament d'autonomie. Chaque équipe peut modifier ou supprimer l'implémentation d'un service dans un microservice avec un impact minimal sur le reste de système. Ils ont plusieurs avantages et inconvénients selon le cas dans lequel ils sont utilisés.[13, 14]

L'architecture microservices est composé principalement de :

1.5.1 Service de configuration

C'est un service de configuration, dont le rôle est de centraliser les fichiers de configuration des différents microservices dans un endroit unique.
Chaque Microservice que nous créons à un certain nombre de configuration. L'ensemble des fichiers de configurations pour cette architecture sont centralisés dans.
Considérons qu'après déploiement de nos Microservices, qu'on veut changer un ou des paramètres de configuration d'un Microservices qui a plusieurs instances. On sera obligé d'arrêter toutes les instances, ce qui n'est pas une bonne pratique. La solution est d'externaliser tous ces fichiers de configuration dans un serveur central. La figure 15 ci-dessous permet d'illustrer.[15]
La figure 1.6

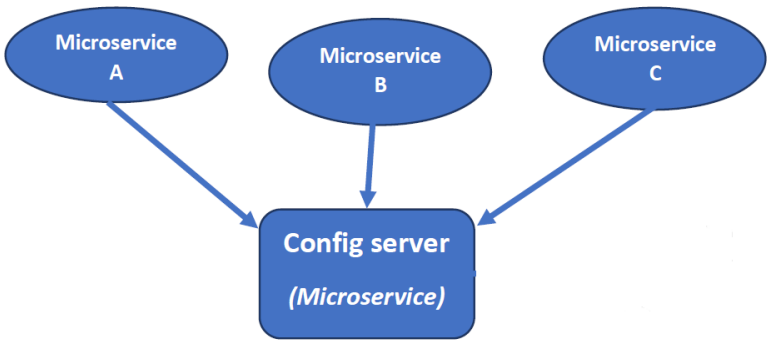

FIGURE 1.6 – Interaction entre le microservice discovery-server et les autres microservices

1.5.2 Service d'enregistrement

C'est un service qui permet aux instances du microservice d'être enregistrées pour la découverte par d'autres microservices. Lorsqu'un microservice est active, de nombreuses instances doivent être lancées pour répondre à la

demande accrue. Ce service permettra aux instances du microservice d'être enregistrées pour la découverte par d'autres microservices. Il est fortement recommandé d'utiliser un service de découverte pour empêcher une forte connectivité entre les microservices. Un service de découverte (la figure 1.7) permet d'enregistrer les attributs de plusieurs services et donc d'éviter d'avoir à appeler un service directement. Le service de découverte offrira dynamiquement les informations essentielles, soutenant la flexibilité et la dynamique de notre architecture de microservices.[15] .

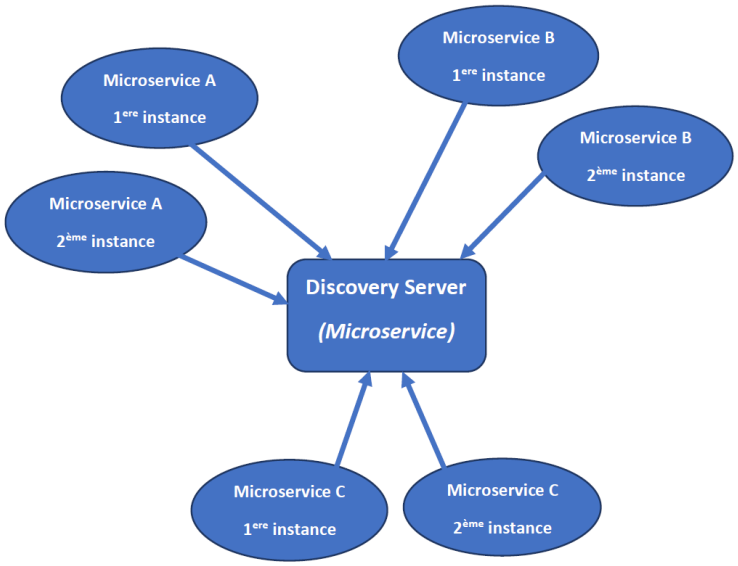

FIGURE 1.7 – Interaction entre le microservice discovery-server et les autres microservices

1.5.3 Serveur proxy

C'est une passerelle qui gère le routage des requêtes vers l'une des instances d'un microservice afin de gérer la distribution de charge automatiquement.fig 1.8 Si nous considérons une application avec de nombreux microservices, la mise en œuvre de la sécurité de tous les microservices pourrait être difficile si nous souhaitons modifier le protocole de sécurité de l'application à l'avenir. Le problème pourrait être résolu avec un seul point d'entrée.[16].

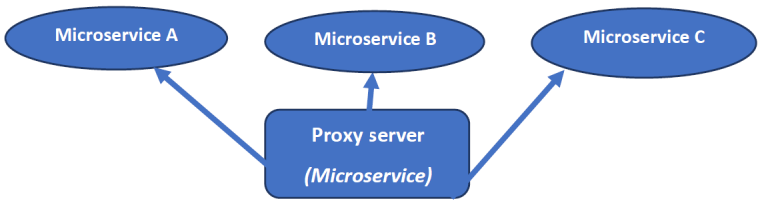

FIGURE 1.8 – Interaction entre le microservice serveur proxy et les autres microservices

1.6 Caractéristiques des microservices

1– **La division en composants via des services :** Le génie logiciel basé sur les composants a donné naissance à cette capacité. Les microservices sont conçus, testés et déployés individuellement. Une mise à jour du service nécessite simplement son déploiement et n'a aucune incidence sur le reste du système. Des méthodes d'appel avec des distances spécifiées sont utilisées par les services pour éviter une connexion sévère entre des composants distincts.[17]

2– **L'organisation autours des capacités métiers :** Les niveaux techniques sont souvent classés par ordre de priorité dans la ventilation des applications traditionnelles. La division est basée sur les compétences commerciales de l'entreprise. En terme de fonctionnement, chaque service est autonome. Il a son propre code, interface et système de gestion des données.[17]

3– **Un Produit, pas un Projet :** L'objectif de l'utilisation des microservices est de produire un logiciel complet en peu de temps. Une équipe est responsable d'un produit tout au long de son cycle de vie selon la vision des microservices. Le logiciel produit est entièrement sous le contrôle de l'équipe de développement.

4– **Les extrémités Intelligentes et les Canaux Stupides :** De nombreuses entreprises investissent dans des voies de communication sophistiquées entre les services, mais avec les microservices, la communication stu-

pide est préférée. Ces communications non-intelligentes ne font que porter les messages, laissant le reste au microservice. L'utilisation de protocoles ouverts pour communiquer entre les microservices est préférée, bien que beaucoup d'autres utilisent des API REST ou des mécanismes de file d'attente pour communiquer entre eux.

5– **Une Gouvernance Décentralisée :** Il est difficile d'identifier une seule technologie qui répond efficacement à toutes les difficultés. Par conséquent, il est préférable d'utiliser l'outil approprié au moment opportun. Chaque service dans une architecture de microservice sera en mesure d'utiliser la meilleure technologie, langue et plate-forme pour ses besoins.

6– **Gestion des Données Décentralisée :** Dans un programme monolithique, il n'y a qu'une seule base de données logique pour les choses permanentes, mais une application de microservices a son propre modèle conceptuel et contrôle ses propres bases de données.

7– **Automatisation de l'Infrastructure :** Ces dernières années, les approches d'automatisation des infrastructures ont considérablement progressé. La progression du nuage a réduit la complexité opérationnelle de la création, du déploiement et de l'exploitation des microservices. Les entreprises qui sont passées à une architecture de microservices ont acquis une expertise en matière de prestation et d'intégration continues. Des outils pour l'automatisation de l'infrastructure sont maintenant utilisés.

8– **Tolérance aux pannes :** Les microservices sont conçus pour résister aux pannes, ce qui est l'un de leurs principaux avantages. Si l'un des services d'une application de microservices tombe en panne, les autres services restent inchangés et adaptent leurs activités à l'état actuel du système.

9– **Une conception évolutive** : Les principes d'indépendance et d'évo-

lutivité sont des propriétés fondamentales d'un microservice, ce qui implique que nous pouvons en reconstruire un sans affecter les autres. En général, la croissance d'une application implique l'inclusion de nouvelles fonctionnalités, ce qui conduit à la construction de nouveaux microservices, ainsi que la mise à niveau des services existants, ce qui implique simplement la mise à jour et le redéploiement du microservice en question.[18, 19]

1.7 Les avantages et inconvénients des microservices

Avantages :

- Passage à l'échelle : pour industrialiser une application basée sur des microservices, il suffit de mettre à l'échelle les composants qui optimisent la consommation de ressources.[20]

- Faible couplage : Comme les composants du microservice ne sont pas interconnectés, ils peuvent être testés indépendamment, ce qui facilite également les changements au fil du temps.

- Les microservices peuvent compter sur l'hétérogénéité technologique, ce qui signifie que chaque service d'un système peut utiliser une technologie différente des autres services pour atteindre les objectifs et les performances souhaités [21].

- Si un composant du système tombe en panne, il n'affecte pas l'ensemble du système.

- Le processus de mise à l'échelle plus accessible que l'échelle d'application monolithique parce que seuls les services qui nécessitent une mise à l'échelle réelle sont mis à l'échelle dans l'architecture des microservices, contrairement à une application monolithique nécessite d'être mis à l'échelle comme une

unité entière qui peut conduire à une utilisation matérielle plus élevée [21].

- La facilité de déploiement, car avec les microservices, chaque service peut être déployé indépendamment sans affecter les performances des autres services.

- L'architecture des microservices aide les entreprises à harmoniser son architecture avec sa structure organisationnelle, ce qui les aidera à réduire au minimum le nombre de personnes qui travaillent sur une base de code spécifique. Par conséquent, les microservices permettent l'alignement organisationnel [21].

Inconvénients :

1–Au lieu d'une seule application, plusieurs composants (microservices) doivent être surveillés. Pour obtenir l'état de chaque composant et l'état général du programme, vous aurez besoin d'une console centrale. Par conséquent, une architecture avec des capacités de surveillance distribuées et de visualisation centralisée doit être construite.[22]

2– Comme tout composant peut cesser de fonctionner à tout moment, la présence de nombreux composants pose un problème de disponibilité.

3–Pour certains clients, un composant peut avoir besoin de contacter la version la plus récente d'un autre composant, tandis que pour d'autres, il peut avoir besoin d'appeler la version antérieure du même composant (gestion des versions).[23]

4– Un test d'intégration est plus complexe car il nécessite un environnement de test dans lequel tous les composants doivent fonctionner et communiquer entre eux.

5–Toutes les étapes pertinentes pour une administration sûre de l'API

doivent êtres exécutées lorsque les intéractions à l'intérieur d'une application basée sur les microservices sont définies comme des appels d'API[24].

1.8 Les étapes d'implémentation d'une architecture microservices

- Identification des microservices où nous pourrions trouver de la valeur.
- Décrire la portée de la responsabilité des microservices identifiés.
- Tenie en compte des types des renseignements qui seront transférés.
- Associer les processus opérationnels aux fonctionnalités pratiques définies.
- Associer les processus technologiques aux processus opérationnels
- Des compétences de recherche qui n'existent pas aujourd'hui, mais qui sont souhaitées.
- Concevoir le microservice en commençant par la définition de l'API et expliquer comment le service sera consommé.
- Élaborer un service ou effectuer une simulation.
- Déployer le microservice.

1.9 Les défis

La figure 1.9 montre qu'il y a quatre principaux défis auxquels les développeurs sont confrontés lorsqu'ils travaillent avec des microservices. la fig présente les réponses des participants à un enquête élaboré en [25]. Les participants conviennent que les transactions distribuées complexes constituent un défi très important. En fait, ce défi a le pourcentage le plus élevé pour la note 1 (32,8 %).environ de 57% et 49% considèrent que tester l'ensemble du

système et les défauts des services, respectivement, sont des défis importants (scores 1 et 2).En revanche, le défi des appels à distance coûteux est très important pour seulement 13,8% des participants . Les praticipants ne sont pas d'accord avec la littérature sur ce dernier défi car ils ne considèrent pas les appels à distance coûteux comme un défi très important dans le développement des microservices.En un mot, les développeurs devraient accorder une attention particulière aux transactions distribuées, aux tests de l'ensemble du système et aux défauts du service, car ceux-ci peuvent devenir de gros problèmes dans le système.[25, 26]

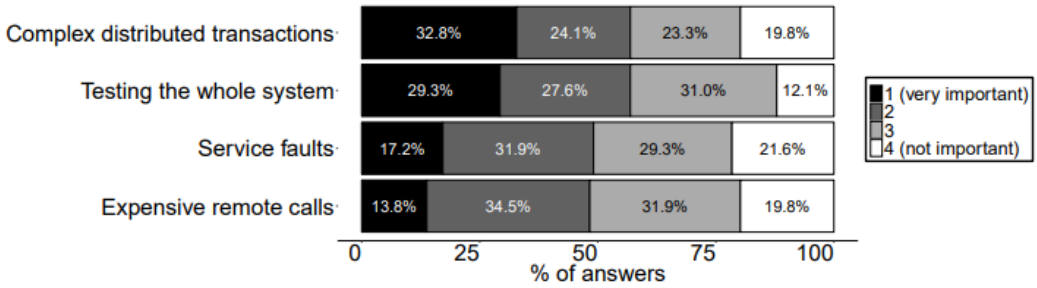

FIGURE 1.9 – Les défis des microservices[27]

Le prochain défi commun est la compétence et les connaissances. Il semble qu'il y ait des difficultés entre les clients et les équipes de développement pour "comprendre les MS des niveaux Top Management/Director", "obtenir les bons développeurs/ingénieurs", et "changer les esprits des gens qui sont habitués aux monolithes traditionnels".[27]

1.10 Architecture microservices vs architecture monolithique

Le tableau 1 présente une comparaison de ces deux types d'architecture. Comme nous pouvons le constater, les deux ont des avantages et des incon-

Catégorie	Microservices	Monolithique
Communication au sein de l'application	Pour communiquer entre eux, les microservices utilisent le modèle de communication demande-réponse. L'implémentation standard utilise des appels REST API basés sur le protocole HTTP .	Les procédures internes (appels de fonction) facilitent la communication entre les composants de l'application. Il n'est pas nécessaire de limiter le nombre d'appels des procédures internes.
Conception des unités	L'application est composée de services à configuration dispersée.	L'application entière est conçue, développée et déployée en une seule unité.
Utilisateurs de technologie	Chaque microservice peut être développé en utilisant un langage et une structure de programmation qui répondent le mieux au problème que le microservice est conçu pour résoudre	En règle générale, l'ensemble de la demande est rédigé dans un seul langage de programmation.
Gestion des données	Décentralisé : chaque microservice peut utiliser sa base de données.	Centralisé : toute l'application utilise les mêmes bases de données.
Installation et maintenance	Les microservices sont simples, ciblés et indépendants. L'application est donc plus facile à gérer	À mesure que la portée de l'application augmente, la maintenance des codes devient plus complexe.
Evolutivité	Chaque microservice peut être évolutif indépendamment des autres services.	L'application entière doit être mise à l'échelle.

Tableau 1. Comparaison entre monolithique et microservices [23, 28, 24]

vénients. Comme conclusion sur le tableau, nous pouvons remarquer que le type d'architecture microservice devient attrayant lorsque nous travaillons avec une grande base de code. Dans le cas des petits projets, les défis tech-

niques que soulèvent les microservices pourraient ne pas avoir suffisamment de temps pour rembourser. Aussi, si les compétences DevOps de l'équipe font défaut, il serait peut-être préférable de s'en tenir au monolithe au début.[29]

1.10.1 Systèmes distribués

La notion sous-jacente des limites des services soutient l'architecture par la distribution des services. Toutefois, les microservices doivent être à l'origine de quelques problèmes de communication à l'échelle du réseau. La première préoccupation importante concerne les pannes du réseau. Parce que c'est quelque chose que le service invoquant n'a aucun contrôle, chaque demande de microservice doit être traitée avec prudence (par exemple, en établissant un délai explicite). Par conséquent, chaque service devrait être conçu pour être robuste et tenir compte du risque de défaillance.

l'efficacité des appels à distance de communication, qui sont nécessaires pour les invocations inter-services, ajoutent de la complexité et du temps au système, qui n'est pas présent dans les systèmes monolithiques modulaires. Il existe une variété de stratégies différentes qui peuvent être utilisées pour augmenter la vitesse globale, comme réduire le nombre d'appels ou les rendre asynchrones. Les défauts mis en évidence ci-dessus ont été explorés dans les travaux de James Gosling, qui en 1997 a élargi sur un projet fourni par Peter Deutsch, qui a déclaré des hypothèses incorrectes sur les systèmes distribués qui sont régulièrement faites. Les huit erreurs du calcul distribué sont les suivantes :

— Le réseau est fiable.

— La latence est de zéro

— La bande passante est infinie.

— Le réseau est sécurisé.

— La topologie ne change pas.

— Il y a un administrateur

— Le coût du transport est nul

— Le réseau est homogène.

1.11 DevOps et Microservices

L'administration des services opérationnels est intrinsèquement plus difficile en raison de la conception des microservices. Étant donné que les services doivent être constamment générés et supprimés, mis à niveau, mis à l'échelle et déployés, il est essentiel que le système utilise des stratégies qui simplifient les opérations. L'automatisation, la livraison continue (CD) et l'intégration (CI), ainsi que peut-être des logiciels de surveillance et d'orchestration externes, sont des exemples de ces techniques. Bon nombre de ces techniques sont bénéfiques même pour les systèmes monolithiques, mais elles sont nécessaires lorsque le système utilise des microservices [30].

Platform comme Service (PaaS) simplifie considérablement les responsabilités opérationnelles dans la migration actuelle vers les infrastructures cloud. Les systèmes de conteneurisation comme OpenShift [30] et Kubernetes [31] facilitent le réseautage, l'évolutivité autonome, la réplication, la résilience, le traçage, la surveillance et une variété d'autres activités. Toutes les approches énumérées ci-dessus raccourcissent considérablement les intervalles de processus du cycle de vie du logiciel. Par conséquent, il est essentiel d'appliquer les concepts DevOps [32] (développement et opérations) qui encouragent une plus grande coopération et une responsabilité partagée entre les équipes. DevOps permet aux entreprises non seulement de produire des produits plus

rapidement tout en maintenant des niveaux élevés de fiabilité, mais aussi d'intégrer la qualité dans le processus de développement.

DevOps est un ensemble de pratiques [33] qui visent non seulement à réduire le temps entre l'application des changements à un système et de les pousser à la production, mais insiste également sur l'amélioration de la qualité des logiciels, à la fois en termes de code et de maintenance. Le mécanisme de déploiement est l'un des éléments clés du processus de développement. Toutes les technologies qui atteignent les objectifs énoncés sont considérées comme des pratiques DevOps [33, 34] La livraison continue (CD) [35] est une pratique DevOps qui fournit des logiciels à la demande dans n'importe quel environnement au moyen de machines automatisées. À mesure que le nombre d'unités déployables augmente, le DC est une contrepartie essentielle aux microservices. Une autre pratique clé de DevOps est la surveillance continue (CM) [36], qui non seulement fournit aux développeurs une rétroaction liée au rendement, mais facilite également l'identification des anomalies opérationnelles [34] .

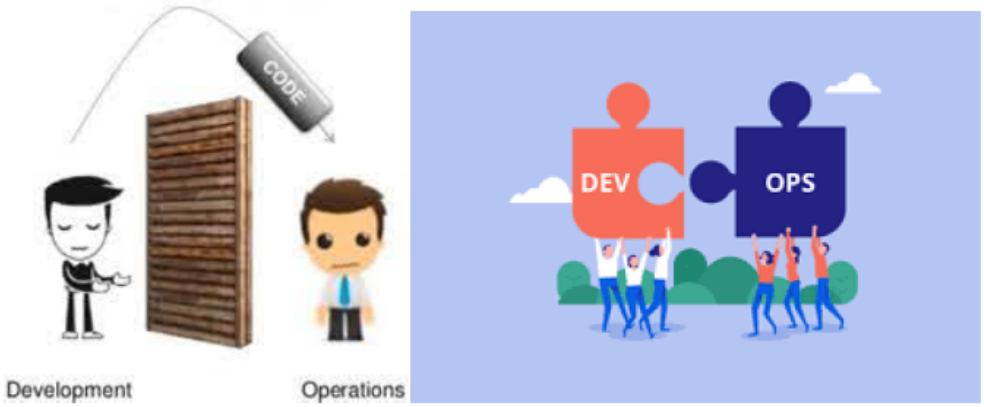

FIGURE 1.10 – Approche DevOps

1.12 Conclusion

Cela fait des années que les microservices suscitent un engouement important dans le développement d'applications. Dans ce chapitre nous avons parlé de l'émergence de l'architecteur des microservices comme une nouvelle architecture après les architectures monolithiques et les architectures orientées service et la comparaison entre eux.

Les microservices ont prouvé leur efficacité par rapport aux architecures précédentes en termes de rapidité, scalabilité et de maintenabilité, nous avons aussi parlé sur l'approche DevOPS .

Chapitre 2

Concepts de base sur les transactions

2.1 Introduction

Ce chapitre explique les concepts fondamentaux des transactions, leurs attributs et les défis fréquents de la gestion des transactions dans les systèmes centralisés et distribués, on a parlé aussi sur les protocoles consensuels les plus utilisées.

2.2 Transaction

Une transaction est une séquence d'opérations qui, lorsqu'elle est exécutée seule dans un environnement sans pannes, fait passer le système d'information d'un état cohérent initial à un état cohérent final. Les états intermédiaires peuvent être temporairement incohérents [37]. Une transaction est dite centralisée lorsqu'elle s'exécute entièrement sur un site. Elle est dite répartie lorsqu'elle met en jeu des actions sur des objets situés sur des sites distincts [38].

La transaction peut être effectuée de deux façons : engagée ou annulée. Un résultat positif est déterminé par le commit, qui indique que toutes les actions à l'intérieur de la transaction ont été achevées. L'abandon indique que toutes

les activités précédemment terminées ont été inversées et que le système est dans le même état que si la transaction n'avait jamais commencé.[39] Il est généralement nécessaire de commencer et de terminer la transaction du point de vue du promoteur. Le système de transaction [39] cache souvent tous les traitements sophistiqués requis pour accomplir les attributs de la transaction, ce qui permet aux développeurs de se concentrer sur les processus opérationnels de la transaction. En général, l'atteinte des qualités susmentionnées varie selon la portée et l'application des idées de transaction.

2.2.1 Propriétés ACID

Une transaction peut être considérée comme un ensemble d'énoncés de logique commerciale ayant des attributs communs [40]. L'atomicité, la consistance, l'isolation et la durabilité sont quelques-uns des critères qui sont couramment examinés. Ces quatre caractéristiques, communément appelées propriétés ACID [41], expliquent les principaux aspects des principes de transaction.

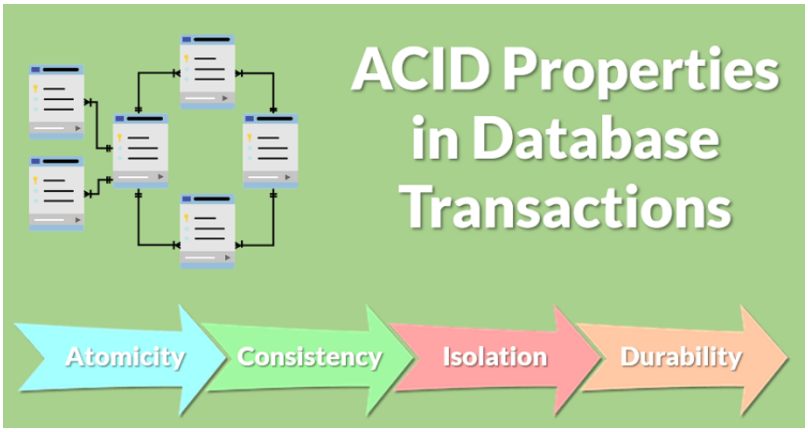

FIGURE 2.1 – Propriétés ACID dans les transactions de base de données[43]

Atomicité

La transaction est constituée d'une série d'activités sur plusieurs ressources. L'attribut atomicité spécifie que les activités de la transaction sont réalisées comme une seule unité. Parce que le terme atomicité est surutilisé dans de nombreuses disciplines de la recherche informatique, certains auteurs préfèrent se référer à elle comme la caractéristique d'abandon dans le contexte de l'ACID. La possibilité d'annuler une transaction par erreur et de faire effacer tous les écritures de cette transaction est connue sous le nom d'abandon [42]. Cela signifie que lorsqu'une transaction s'engage avec succès, toutes ses activités doivent également s'engager correctement. Lorsqu'une transaction échoue et doit être annulée, toutes les activités terminées et tous les effets doivent être annulés.

Isolation

La propriété isolation prend effet lorsque plusieurs transactions peuvent être exécutées simultanément sur les mêmes ressources. Elle garantit que les transactions parallèles ne peuvent pas interférer les unes avec les autres. Par conséquent, chaque exécution simultanée sur la ressource partagée doit être équivalente à un ordre de série de transactions contenues. C'est pourquoi l'isolement est souvent appelé une sérialité. Du point de vue d'une vue externe, la propriété d'isolement signifie que la transaction apparaît telle qu'elle a été exécutée dans le système entièrement seul. Cela signifie que même s'il y a plusieurs transactions effectuées simultanément, ce fait est caché du point de vue externe. Comme une extension instinctive de la propriété de cohérence, l'exécution en série des transactions préserve l'état de cohérence. L'exécution des opérations en parallèle ne peut donc pas donner lieu à un système

incohérent.

Durabilité

Cette propriété spécifie que toutes les modifications des transactions doivent être persistantes, ce qui signifie que toute modification d'état effectuée lors d'une transaction validée avec succès doit être préservée en cas de défaillance du système. La manière dont l'état est conservé est généralement déterminée par la mise en œuvre du système de transaction. Dans la plupart des cas, le stockage persistant, comme un disque dur ou le nuage, est suffisant pour accomplir cette capacité. Même si ce niveau de stockage est acceptable, il ne protégera pas les données en cas de défaillance catastrophique plus grave.[43]

Cohérence

Le mot cohérence fait référence aux restrictions imposées aux changements de données qui ne peuvent se produire que de manière autorisée. Lorsque les données sont persistées, elles doivent être valides selon toutes les règles définies qui répondent aux invariants de l'application. La propriété consistency décrit que la transaction maintient la cohérence du système et des ressources sur lesquels elle est effectuée. Lorsque la transaction est lancée dans le système cohérent, c e système doit demeurer uniforme lorsque la transaction prend fin – il passe d'un état cohérent à un autre. Contrairement aux autres propriétés transactionnelles (A, I, D), la cohérence ne peut pas être réalisée par le système de transaction car il ne détient aucune connaissance sémantique sur les ressources qu'il manipule [44]. Par conséquent, la réalisation de ce bien est la responsabilité du code d'application. La propriété de cohérence d'une transaction est assurée par le contrôle d'intégrité, et est généralement

du ressort du concepteur de l'application (même si certains systèmes intègrent un vérificateur de cohérence). Les propriétés d'atomicité et de durabilité sont assurées grâce à des techniques de validation et reprise ou recouvrement.

2.2.2 Contrôle de concurrence

Le but de ces techniques est d'assurer la propriété de sérialisation. Le principe consiste à ordonnancer les actions des transactions concurrentes de telle sorte que le résultat de leur exécution soit équivalent à celui d'une exécution séquentielle. L'ordonnancement est basé sur la détection des conflits d'accès aux objets partagés par des transactions concurrentes. Il y a conflit d'accès lorsque deux transactions exécutent des actions incompatibles sur le même objet (exemple lire et écrire). L'apparition des conflits crée des relations de dépendance qui peuvent être représentées à l'aide d'un graphe dans lequel les noeuds représentent les transactions et les arcs représentent les dépendances engendrées par les conflits d'accès. Un cycle dans le graphe des dépendances indique une exécution non sérialisable. Il existe deux groupes de techniques de contrôle : le contrôle continu et la certification. Le contrôle continu assure la sérialisation tout le long de la vie des transactions, celles qui provoquent des exécutions non sérialisables sont soit abandonnées soit mises en attente. Ces méthodes dites pessimistes supposent une fréquence élevée de conflits et sont bâties autour de deux techniques de base : le verrouillage à deux phases et l'estampillage.

Ordonnancement par estampillage

Le principe de ces techniques consiste, lorsqu'il y a conflit, à forcer les transactions à s'exécuter selon un ordre de sérialisation pré-établi. En cas

de conflit, le système vérifie que les accès concurrents sont effectués dans l'ordre préétabli entre les transactions à l'aide de l'estampille. Si c'est le cas, l'exécution est acceptée, sinon, la transaction est annulée et reprise. C'est une technique adaptée à la mise en cohérence des copies multiples d'une donnée.

Verrouillage à deux phases

Le principe de fonctionnement du verrouillage à deux phases ou 2PL (Two Phase Locking) consiste à protéger les objets partagés par des verrous. Un conflit a lieu lorsqu'une demande de verrouillage, émise par une transaction, est incompatible avec le verrou existant sur l'objet. La relation de dépendance est définie par l'ordre chronologique des opérations sur l'objet partagé. Pour traduire cette dépendance en termes d'exécution séquentielle, on force la transaction à l'origine du conflit à attendre la libération de l'objet par la transaction concurrente.

Méthodes des contrôles par certification

Cette approche diffère les contrôles à la fin de la transaction Si une action est jugée incompatible avec celles d'opérations concurrentes, l'opération est interrompue et reprise. Sinon, la transaction est terminée. En conséquence, la transaction est effectuée en trois (03) étapes [45] :

— La phase de lecture.

— La phase de validation.

— La phase d'écriture.

2.2.3 Validation et reprise

Cela implique d'assurer les qualités d'atomicité et de permanence en présence de défaillances telles que celles qui provoquent l'arrêt de la transaction avant l'achèvement et celles qui font disparaître l'information de la mémoire secondaire. Pour atteindre le contrôle de l'atomicité en utilisant les deux processus d'annulation et de révision du gestionnaire de transaction, une transaction est terminée en deux phases, le calcul et la validation, et un journal est conservé en mémoire permanente. Les modifications apportées aux objets sont enregistrées dans le journal lors de l'étape de calcul qui correspond à l'exécution de la transaction. Le système consolide les actions effectuées sur les objets pendant la phase de calcul dans la phase de validation. Les deux formes de journaux les plus répandus sont le journal-avant qui stocke les valeurs des objets avant leur modification par une transaction et le journal-après qui stocke les valeurs des objets modifiés par une transaction. Il est utilisé pour refaire les actions d'une transaction. La validation consiste à propager les modifications enregistrées dans l'espace de travail de la transaction vers la mémoire secondaire. Les journaux permettent de fiabiliser cette opération.

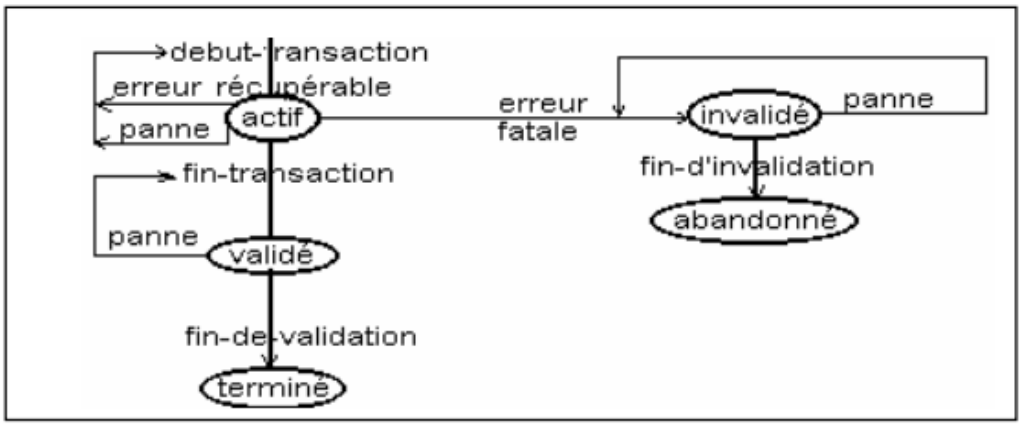

FIGURE 2.2 – Diagramme d'état d'une transaction

Lors d'un redémarrage après échec, le système de transaction effectue

l'action associée au statut rapporté dans le journal pour chaque transaction :

— état actif : la transaction est annulée et ses actions sont défaites,

— état validé : la procédure de validation est refaite,

— état invalidé : la procédure d'invalidation est refaite,

— état terminé ou abandonné (annulé) : pas d'action.

2.3 Protocoles consensuels

Le problème du consensus représente la procédure à suivre pour parvenir à un accord sur la valeur des données partagées entre plusieurs composantes. Il a son application dans de nombreux environnements, y compris les transactions où le TM doit conclure si une transaction peut être engagée en fonction du consensus des participants. Un protocole de consensus décrit une série d'étapes qui résolvent le problème de consensus. Ces étapes peuvent généralement être divisées en trois phases : la sélection des valeurs des candidats, l'échange de valeurs entre les participants et l'accord. La décision finale de chaque participant est irréversible. Le protocole de consensus est correct s'il respecte ces conditions [46] :

— **Accord** tous les noeuds non-faulty décident sur le même simple valeur

— **Validité** si tous les noeuds non communs ont la même valeur initiale, alors le consensus doit être atteint sur cette valeur

— **Résiliation** tous les nœuds non défectueux finissent par décider

Dans les architectures d'entreprise, il existe deux grands protocoles consensuels, à savoir :

— **Two-phase commit (2PC)** protocol

— **Three-phase commit (3PC)** protocol

2.3.1 Two-phase commit (2PC)

Dans les systèmes de bases de données, le commit en deux phases est un schéma bien connu. Cette approche peut également être utilisée pour construire des transactions distribuées dans les microservices. Un commit à deux phases a un noeud de contrôle qui détient la majorité des noeuds logiques et participants (microservices) qui conduisent les actions. Il fonctionne en deux étapes :[47, 48]

— **Phase de préparation (phase 1) :** Tous les noeuds participants sont priés de confirmer s'ils sont prêts à commettre par le noeud de contrôle. Les réponses par oui ou non sont envoyées par les noeuds participants.

— **Phase d'engagement (phase 2) :**Si tous les noeuds ont dit oui, le noeud de contrôle leur demandera de valider. Même si l'un des nœuds répond négativement, le nœud de contrôle demande à ce qu'il revienne en arrière.

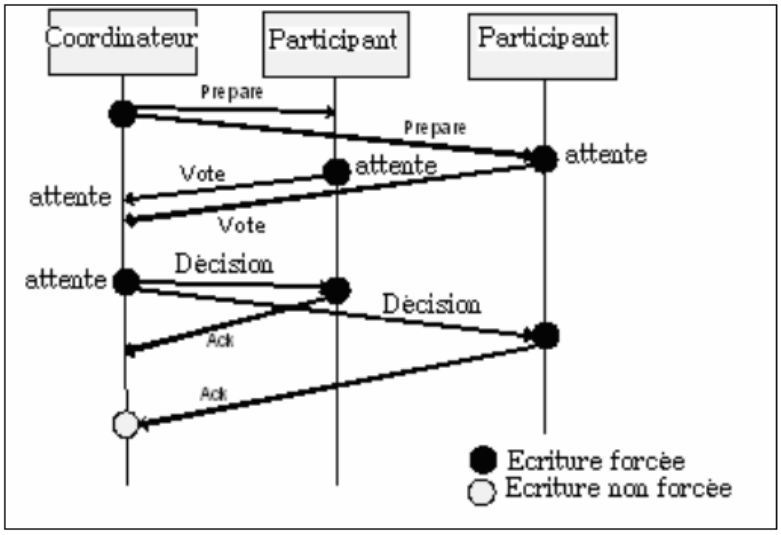

FIGURE 2.3 – Le protocole 2PC[50]

Même si 2PC peut aider dans la gestion des transactions dans un système distribué, il crée également un point d'échec unique parce que le coordinateur

est responsable de toutes les transactions. Le rendement total dépend du nombre d'étapes. En raison du chatinage du coordinateur, l'ensemble du système est contraint par les ressources les plus lentes, car chaque nœud prêt doit attendre la confirmation d'un nœud plus lent. De plus, la plupart des mises en œuvre d'un tel coordonnateur sont synchronisées, ce qui pourrait entraîner une baisse des performances à l'avenir. 2PC continue d'avoir les défauts suivants :

— Il n'y a aucun moyen de faire reculer l'autre transaction si un microservice tombe pendant l'étape de validation.

— Les autres services doivent attendre que la confirmation du service le plus lent soit terminée avant de poursuivre. Les ressources des services sont verrouillées jusqu'à la fin de la transaction.

— En raison de leur dépendance au coordinateur de transaction, les engagements en deux phases sont intrinsèquement lents. Cela pourrait poser des problèmes d'évolutivité, en particulier dans un programme de microservices ou dans un scénario de rétrogradation à grande échelle.

2.3.2 The Three-phase commit (3PC)

Le protocole 3PC est une extension du protocole 2PC, et son objectif est de dépasser ses limites, qui sont liées à sa nature bloquante. L'une des principales caractéristiques de ce protocole de consensus est qu'il est non bloquant ; cela ne signifie pas que les participants ne sont pas bloqués pendant le traitement, mais cela signifie que le protocole peut se poursuivre en dépit des échecs. Il se compose de trois phases ; la phase de préparation et la phase de validation (ou d'abandon) sont les mêmes que celles du protocole 2PC. Cependant, il introduit une nouvelle phase, que nous pouvons définir comme un état

préparé, où tous les participants de la transaction définiront leurs statuts. Le statut peut être en attente ou pré-engagement. Cela signifie que la phase finale ne peut avoir qu'un seul état final - annulé, si la phase précédente était en attente, ou commit, si la phase précédente était pré-amorcée. 3PC atténue la nature bloquante de 2PC. Le problème est qu'il l'obtient avec un protocole plus compliqué, et avec un besoin d'envoyer un autre message avant de décider de commettre ou d'annuler la transaction. En outre, il ne résout pas le problème avec les partitions réseau. 3PC n'est pas largement utilisé dans les environnements de production.[49]

2.4 Conclusion

Nous avons présente dans ce chapitre les concepts fondamentaux des transactions distribuées , et comment améliorer la transactions de données distribuées dans plusieur applications ,enfin Nous avons discuté 2PC et 3PC Comme l'une des meilleures Protocoles consensuels actuellement utilisées.

.

Chapitre 3

Gestion des BDD dans les architectures microservices

3.1 Introduction

Dans ce chapitre, nous discutons la gestion les bases de données pour les micro-services, présentons les patterns spécifiques et quelques travaux de recherche connexes . Enfin, nous parlons de notre contribution sur la mise en place d'une application de microservices basée sur le modèle event-driven.

3.2 Les bases de données

La base de données est un ensemble de données structurées ou d'informations stockées dans un système informatique, de sorte qu'un programme informatique ou une personne peut utiliser un langage de recherche pour récupérer cette information. Les informations ainsi obtenues peuvent être utilisées dans un processus décisionnel. Le programme informatique utilisé pour la gestion des données et les requêtes est appelé SRBD (Database Management System). Les informaticiens peuvent classer le système de gestion de base de données en fonction des modèles de base de données qu'ils sou-

tiennent. Les premières bases de données mentionnées (relationnelles) sont devenues dominantes dans les années 1980. [50] Elles appuient l'utilisation de lignes et de colonnes dans une série de tableaux. La majorité la plus significative d'entre eux utilisent également Sql pour écrire et interroger des données. Les bases de données non relationnelles sont devenues populaires dans les années 2000, appelées NoSql parce qu'elles utilisent un langage de requête différent.[51]

3.2.1 NoSQL

Les bases de données NoSQL sont conçues pour répondre aux exigences de performance et d'évolutivité des applications Web qui ne peuvent pas être traitées par les bases de données relationnelles traditionnelles. En raison de leur contraste entre les priorités et l'architecture des bases de données relationnelles conventionnelles utilisant SQL, ces bases de données sont appelées bases de données « NoSQL » parce qu'elles intègrent de nombreuses fonctionnalités supplémentaires en plus des caractéristiques des bases de données conventionnelles. Les bases de données relationnelles suivent fortement les propriétés ACID (Atomicity, Consistency, Isolation, and Durability) tandis que les bases de données NoSQL suivent les principes BASE (Basically Available, Soft State, Eventual consistency).[52]

3.3 Problématique

Les entreprises s'intéressent de plus en plus aux microservices, comme ces derniers offrent une approche légère, indépendante, réutilisable et rapide du déploiement des services qui réduit les risques liés à l'infrastructure.

Ces avantages soulèvent un problème concernant le partage de la base de

données entre les microservices constituant l'application. En effet, Le partager la base de données entre permet de créer une association forte entre les différents microservices. Cela est contradictoire avec les principes des architectures microservices ! Les efforts de synchronisation et de connectivité seront toujours substantiels, même pour une mise à jour ou une modification minimale de la base de données.[53]

A l'inverse, si chaque microservices dispose de sa propre base de données, les échanges de données entre microservices ouvrent la porte à d'autres problématiques. C'est pourtant en théorie un choix naturel pour les microservices qui utilisent un modèle de données différent pour chaque service. C'est ce qu'on appelle la persistance polygotte fig 3.1 .

Un service peut s'appuyer sur une base de données relationnelle, une autre sur une base de données orientée colonne (Cassandra par ex) puis un autre sur une base clé-valeur, un autre sur un entrepôt de type Document, etc. On peut également se retrouver dans un cas de figure où un service repose sur plusieurs bases de données.[54]

dans ce qui suit nous présentons les approches utilisés pour gérer les bases des données dans les AMS

FIGURE 3.1 – Persistance polyglotte dans les microservices[56]

3.4 Microservices Patterns

Il ya plusieurs directions pour régler les problèmes causés par la décentralisation des données, où les services peuvent stocker des données sans sacrifier leur autonomie, utilisant plusieurs modèles de bases de données de microservices . Ci-dessous quelques Patterns et concepts pour concevoir une architecture en microservices.

3.4.1 Database per Microservice Pattern

L'architecture des bases de données est en constante évolution, et la conception d'une solution basée sur les microservices présente divers défis. L'architecture des bases de données est l'un des éléments les plus importants des microservices.

Lors de la mise en œuvre de l'architecture de microservice, il devrait y avoir deux alternatives de base pour l'organisation des bases de données :

— Database per service

— Shared database

Database per service

Le principe est facile à comprendre. Chaque microservice possède son propre stockage de données (schéma complet ou tableau). D'autres services ne peuvent pas accéder aux dépôts de données qu'ils ne possèdent pas. Ce type de solution offre plusieurs avantages.

D'autre part, le stockage de données individuelles est simple à développer. En outre, les données du domaine sont encapsulées par le microservice. Par conséquent, il est beaucoup plus facile de comprendre le service et ses données dans leur ensemble. C'est particulièrement important pour les nouveaux membres de l'équipe de développement.[55]

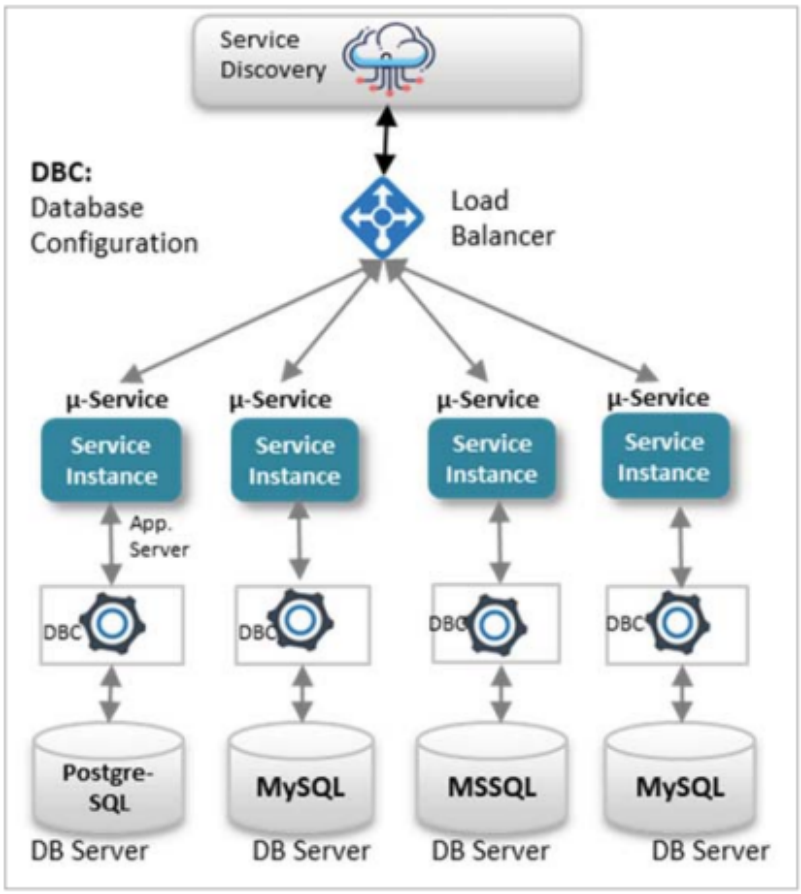

FIGURE 3.2 – The Per Service Pattern for (DBPS)

Ils passeront moins de temps et d'efforts à comprendre la région dont ils sont responsables. L'inconvénient fondamental de ce service de base de données est qu'il nécessite une solution de sécurité en cas de panne de communication.

Shared Database

L'utilisation d'une base de données partagée est une tendance qui devrait être évitée. Toutefois, elle est discutable. Le problème est que lorsque les microservices utilisent une base de données commune, ils perdent leur évolutivité, leur fiabilité et leur indépendance. Par conséquent, les bases de données partagées sont rarement utilisées dans les microservices.

Nous devrions examiner si les microservices sont réellement essentiels lorsqu'une base de données partagée semble être le meilleur choix pour un projet

de microservices. Le monolithe peut être l'option supérieure. Regardons comment fonctionne une base de données partagée.

Il n'est pas courant d'utiliser une base de données partagée avec des microservices. Lors de la migration d'un monolithe vers des microservices, un état temporaire peut être produit. Le principal avantage d'une base de données partagée par rapport à une base de données par service est la gestion des transactions. Il n'est pas nécessaire de répartir les transactions entre de nombreuses transactions.

3.4.2 Event Sourcing

Le sourcing événementiel est en charge de générer une nouvelle séquence séquentielle d'événements. L'état de l'application peut être recréé en interrogeant les données, mais nous devons réimager chaque changement à l'état du programme pour le faire. Le principe de l'approvisionnement en événements est que tout changement dans l'état d'une entité devrait être pris en charge par le système.[56]

La persistance d'un article est obtenue en stockant une succession d'événements changeant l'état. Chaque fois que l'état d'un objet change, un nouvel événement est ajouté à la séquence des événements. Parce que c'est une seule opération, c'est pratiquement atomique. L'état actuel d'une entité peut être recréé en rejouant ses occurrences.

Une boutique événementielle est un endroit où vous pouvez garder une trace de tous vos événements à venir. Le magasin d'événements fonctionne à la fois comme un courtier de messages et une base de données d'événements. Il permet aux services de s'abonner à des événements en utilisant une API. Le magasin d'événements transmet des informations sur chaque événement

entré dans la base de données à tous les abonnés intéressés. Le magasin d'événements est au cœur d'une architecture de microservices axée sur les événements.[57]

Ce modèle peut être utilisé dans les scénarios suivants :

1-Il est essentiel de préserver le stockage actuel des données.

2-La base de code de la couche de données actuelle ne doit pas être modifiée

3-Le succès de l'application dépend des transactions.

Comme le montre l'explication précédente, le sourcing événementiel aborde la question de l'établissement d'une architecture événementielle. Les micro-services qui partagent des bases de données sont difficiles à développer. En outre, la base de données servira de point d'échec unique. Divers services pourraient être touchés par les modifications apportées à la base de données.

3.4.3 Ségrégation des responsabilités de requête de commande (CQRS)

La ségrégation des responsabilités de commande et de requête (CQRS) est un modèle architectural où l'objectif principal est de séparer la manière de lire et d'écrire les données. Ce modèle utilise deux modèles distincts : [57]

— **Commandes** : Modifier l'état de l'objet ou de l'entité.

— **Requêtes** : Retourner l'état de l'entité et ne changera rien.

Les méthodes traditionnelles de gestion des données comportent certaines lacunes.

— 1. La possibilité d'altération des données.

— 2. Comme les objets sont accessibles aux applications de lecture et d'écriture, il est difficile de gérer les performances et la sécurité.

Par conséquent, le CQRS examine la situation dans son ensemble afin de

régler ces problèmes. Le CQRS est chargé soit de changer l'état de l'entité, soit de restituer le résultat.[58]

Les avantages CQRS

1. Comme les modèles de requête et les instructions sont séparés, la complexité du système est réduite au minimum.
2. Peut donner diverses perspectives d'interrogation.
3. Peut optimiser les côtés lecture et écriture du système individuellement.

Le côté écriture du modèle est responsable de la persistance des événements et sert de source de données pour le côté lecture. Le modèle de lecture du système crée des représentations matérialisées des données, qui sont souvent largement dénormalisées.

3.4.4 SAGA

Sans les principes ACID, SAGA est l'une des meilleures options pour maintenir la cohérence des données dans l'architecture distribuée. SAGA est en charge de réaliser plusieurs transactions de commentaires tout en permettant des rollbacks.[59]

Il y a deux options pour compléter l'histoire :

— **Chorégraphie** : dans laquelle chaque service peut déclencher l'événement d'un autre service sans coordinateur central.

— **Orchestration** : dans laquelle un coordinateur central prend la décision de déclencher les événements pertinents de la saga.

chorégraphie d'événement vs. orchestration

Dans l'approche chorégraphie événementielle, lorsqu'un micro service exécute une transaction locale, il publie un événement qui peut être souscrit par l'un ou l'autre des micro services pour déclencher leurs transactions locales [60]. Ce processus se poursuit jusqu'au dernier service qui ne publie plus d'événements, en marquant là la fin de la transaction, Dans cette approche, il n'y a pas de coordinateur central qui écoute les événements et déclenche la transaction locale du micro service concerné. L'autre technique pour implémenter les sagas s'appelle l'orchestration. En cela, il y a un coordinateur central qui écoute tous les événements émis par n'importe quel micro service de transaction locale. En fonction de l'événement entrant, il déclenche la prochaine transaction locale dans un ou plusieurs micro services différents. Les scénarios mentionnés ci-dessus sont représentés à l'aide d'une seule entité à chaque niveau de transaction locale. Cela peut être complexe s'il existe des collections dépendantes dans chacune de ces sources de données. Lorsqu'une transaction doit être annulée, l'état des collections dépendantes doit également être annulé. Les deux techniques mentionnées ci-dessus ont des avantages et des inconvénients en fonction du scénario à mettre en œuvre. Dans la section suivante, nous allons simuler les différents scénarios et comprendre la pertinence de ces techniques.

Les avantages de SAGA

Peut être utilisé pour maintenir la cohérence des données entre de nombreux services sans les lier étroitement.

Les inconvénients de SAGA

Du point de vue du programmeur, le modèle de conception de SAGA est

complexe, et les développeurs ne sont pas habitués à créer des sagas comme des transactions traditionnelles.

3.4.5 API Gateway

Ce modèle d'architecture de microservice est idéal pour les gros systèmes avec diverses applications clientes, car il fournit un point d'entrée unique pour un ensemble de microservices.[61]

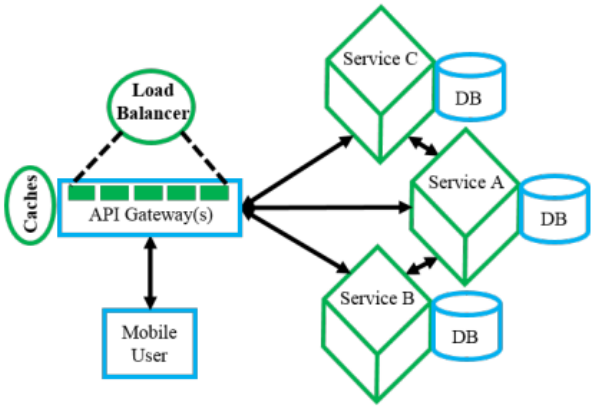

FIGURE 3.3 – API Gateway

La passerelle API est un proxy inverse qui se situe entre les applications clients et les microservices, transmettant les demandes des clients aux services. D'autres services transversaux qu'il peut fournir comprennent l'authentification, la terminaison SSL et la mise en cache.[62]

Les exemples suivants definie Pourquoi l'architecture API Gateway est-elle envisagée au lieu d'une communication client-microservice directe :

1. Problèmes de sécurité : Sans passerelle, tous les microservices doivent être exposés au « monde externe », ce qui augmente la surface d'attaque comparativement à la dissimulation de microservices internes qui ne sont pas directement accessibles par les applications clientes.

2. Questions générales : Chaque microservice accessible au public doit

traiter l'autorisation et le protocole SSL. Dans de nombreuses situations, ces problèmes peuvent être résolus en une seule couche, réduisant ainsi le nombre de microservices internes.

3. Couplage : Sans la conception de la passerelle API, les applications clientes sont connectées aux microservices internes. Les microservices doivent être compris par les applications clientes afin qu'elles puissent ventiler les multiples parties de l'application.

Enfin, la passerelle de l'API des microservices doit pouvoir faire face à des défaillances partielles. La défaillance d'un seul microservice sans réponse ne devrait pas entraîner l'échec de l'ensemble de la demande.

Les défaillances partielles peuvent être traitées de diverses façons par une passerelle d'API de microservices, notamment :
1-Utilisez les données d'une requête précédente qui a été mise en cache.
2-Pour les données sensibles au temps qui constituent l'objectif principal de la demande, retournez un code d'erreur.
3-Fournir une valeur vide
4-Compter sur le matériel haut 10 valeur.

3.4.6 Event-Driven Architecture

Dans les applications monolithiques, la gestion des données est relativement facile – il n'existe qu'une seule base de données et elle est généralement relationnelle. Lors de l'utilisation de bases de données relationnelles, les applications peuvent utiliser le paradigme ACID (Atomicité, Consistance, Isolation et Durabilité) pour modifier de nombreuses lignes et tables en une seule transaction [63, 40]. Dans une architecture de microservices, cependant, les données de chaque service sont privées pour ce service et ne peuvent être

consultées que via l'API du service [64].Il présente les principaux défis suivants en matière de données distribuées :

• Extraire les données de plusieurs services.

• Mettre en œuvre des transactions commerciales qui maintiennent l'uniformité des données pour plusieurs services.

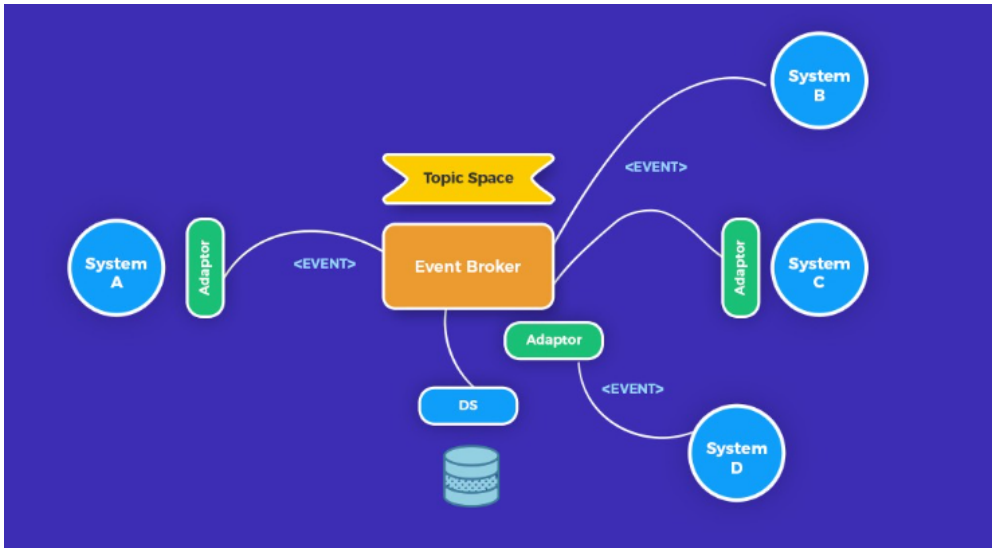

FIGURE 3.4 – Event Driven Architecture

L'architecture pilotée par les événements (EDA) est un modèle d'architecture asynchrone distribué populaire, qui peut être utilisé pour surmonter les défis des données distribuées. Il est hautement évolutif et flexible [65]. Dans EDA, chaque microservice publie un événement lorsque quelque chose de notable se produit, c'est-à-dire que le service de commande publie un nouvel événement lorsque la commande a été créée ou modifiée. Les autres microservices s'abonnent aux événements qui les intéressent. Un événement peut être défini comme "un changement significatif d'état" [66]. Les événements peuvent être utilisés pour implémenter des transactions commerciales qui couvrent plusieurs services. Les transactions peuvent être représentées par une série d'étapes où chaque étape est un microservice, qui met à jour ou crée une entité commerciale et publie un événement qui déclenche l'étape

suivante. EDA a deux topologies principales - médiateur et courtier :

Topologie du médiateur : utile pour les événements avec plusieurs étapes. La topologie de médiateur a quatre composants principaux : les files d'attente d'événement, le médiateur d'événement, les canaux d'événement et les processeurs d'événement. Le flux d'événements commence par un client envoyant un événement à une file d'attente, qui le transporte au médiateur d'événements. Ensuite, le médiateur envoie des événements asynchrones supplémentaires aux canaux d'événements pour exécuter chaque étape du processus. Les processeurs d'événements écoutent les canaux d'événements et exécutent une logique commerciale spécifique pour traiter l'événement [65].

Topologie de courtier : Dans cette topologie, il n'y a pas de médiateur d'événement central pour orchestrer l'événement initial. Il existe deux types de composantes : les courtiers et les transformateurs d'événements. Chaque processeur d'événement est responsable du traitement d'un événement et de la publication d'un nouvel événement pour informer les autres

‘

Travaux de recherches connexes

Beaucoup de travaux de recherche ont traité les problèmes de gestion de base de donnée dans les microservice Nous citons ci-dessous quelques-uns :

3.5 Comparaison des architectures microservices et des architectures monolithique

En Lviv et all [67],ont réalisé une comparaison entre l'architecture monolithique et une architecture basée sur les microservice et monolithique archi-

tecture.

Pourcela deux types d'architectures présentées deux applications aux fonctionnalités identiques ont été développés. Les applications ont été conçues pourgérer la location des voitures. La première implémentation était une version monolithique créée à l'aide de Spring Boot. La base de données choisie était PostgreSQL. La connexion entre l'application et la couche persistante a été fournie par Hibernate - un module utilisé pour le mappaging objet-relationnel en Java. L'ensemble du projet a été conteneurisé avec Docker en utilisant l'image "openjdk : 8-jre-alpine". L'application Microservice est une version modularisée de la variante monolithique. Il a été créé à l'aide des technologies Spring Boot et Spring Cloud. Le dernier fournit des bibliothèques prépa- rées et ajustées (certaines d'entre elles basées sur le logiciel Open Source Netflix) qui ont été utilisées pour implémenter la découverte de services (Eureka) et une passerelle (Zuul). Eureka aide à découvrir un microservice disponible qui est déjà en cours d'exécution. Zuul est une implémentation du modèle API Gateway qui masque principalement tous les microservices d'un client et partage les points de terminaison comme une seule appli- cation. Tous les microservices doivent être autonomes et indépendants, c'est pourquoi un courtier de messages (Apache Kafka) est utilisé. La couche base de données est comme dans la partie monolithique (Hibernate avec PostgreSQL). Pour simplifier le processus de déploiement, chaque microservice a été mis sous un conteneur.

A fin de realisé les experiances, un serveur puisant de Disque dur - Samsung SSD 840 EVO 120 Go ,ils ont traité 30000 requetes pour la premier fois ensuite avec 300000 requêtes.

FIGURE 3.5 – Performances architecturales et le Temps de réponse (HTTP GET)[68]

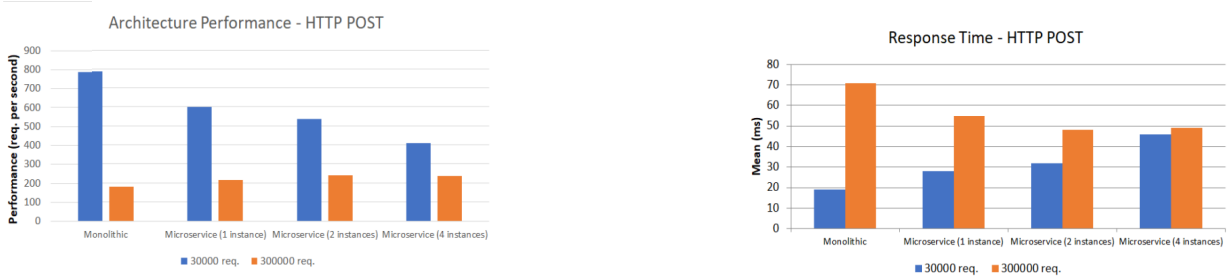

FIGURE 3.6 – Performances architecturales et le Temps de réponse (HTTP POST)[68]

Conclusion

Les résultats sont évalués en termes de nombre de requêtes exécutées en seconde et aussi en termes de latence.

Les tests de charge effectués ont indiqué que l'architecture microservice donne des mauvais si la charge est petite, la réplication des microservices aggrave la situation. Donc, l'architecture monolithique n'est pas mauvaise. Il est plus efficace à faible charge et elle est facile à développer. Il ne présente pas tellement de problèmes d'intégration, de connexion et de configuration.

L'architecteur microservices est très bénéfique lorsque de nombre de requêtes est important. L'augmentation du nombre des réplicas n'augmente pas toujours les performances. Dans ce cas, l'AMS présente de nombreux avantages qui permettent de créer un logiciel de haute qualité, facile à mettre à l'échelle, plus fiable et à long terme plus pratique à entretenir.

Le choix de la bonne architecture doit être défini par les objectifs de l'entreprise afin que l'investisseur obtienne le produit qui répondra à ses attentes.

3.6 Comparaison de la chorégraphie d'événements et des techniques d'orchestration

En [59], Chaitanya K. Rudrabhatla ,ont fait une analyse quantitative des performances des techniques de chorégraphie d'événements et d'orchestration utilisées pour implémenter le modèle de conception saga pour gérer les transactions distribuées dans des bases de données isolées NOSQL dans une architecture de micro services.

— 1 Les microservices ont utilisé la technologie Spring Boot. Un composant de découverte de service appelé Eureka est utilisé pour enregistrer et découvrir les microservices en cours d'exécution. Les entités sont représentées sous forme de collections dans une base de données open source no-SQL Mongo.

— 2 Chaque micro service MS1, Ms2, ...MSn a une instance isolée de Mongo DB- DB1, DB2.DBn, respectivement avec une collection C1, C2,.. Cn, s'exécutant respectivement sur chacune de ces instances de base de données. Ces microservices et instances de base de données s'exécutent sur des machines virtuelles basées sur Linux. Tout d'abord, la technique de chorégraphie d'événements est exécutée avecdeux microservices, MS1 et MS2 ayant DB1 et DB2 comme bases de données pour chaque microservice avec C1 et C2 comme collections dans chaque base de données respectivement.

— 3 Chaque collection a un attribut appelé état qui décrit l'état de l'entité avec les valeurs possibles de S1 et S2 et un attribut appelé horodatage qui enregistre l'horodatage lorsque le changement d'état s'est produit.

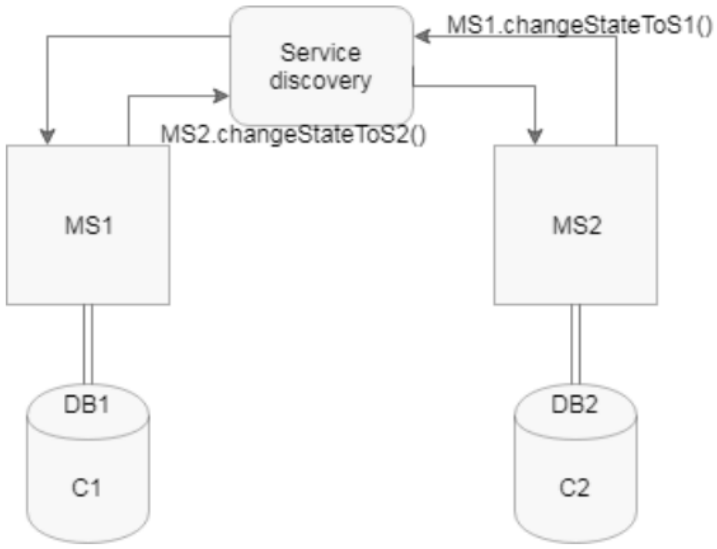

FIGURE 3.7 – Chorégraphie événementielle avec 2 micro services.[60]

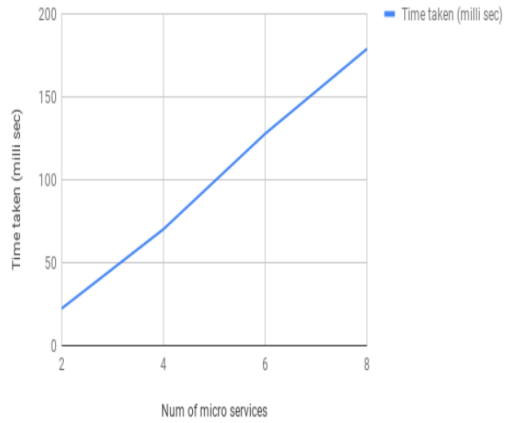

FIGURE 3.8 – Corrélation du temps pris par rapport aux microservices dans la chorégraphie d'événements[60]

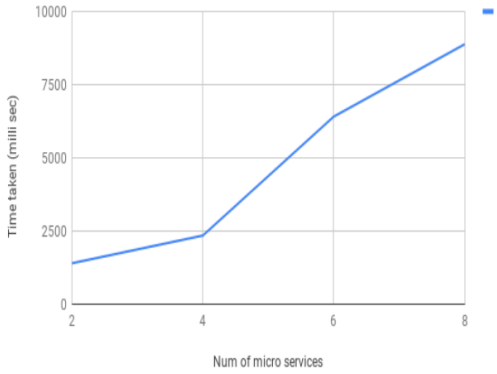

FIGURE 3.9 – Corrélation du temps pris par rapport aux microservices dans l'orchestration[60]

L'étude a permet déterminer clairement que la chorégraphie d'événement est beaucoup plus rapide en performance par rapport à l'orchestration. Cependant, la chorégraphie des événements devient très com- plexe à coder et à gérer si plusieurs événements sont déclenchés à partir de chaque micro service. L'étude a prouvée aussi que la gestion de plusieurs actions pour les déclencheurs sans orchestrateur central est difficile car un développeur ou une équipe travaillant sur un micro service peut ne pas être au courant de

l'autre.

L'étude a conclu que la chorégraphie des événements est une approche suggérée lorsqu'il y a :

— Un nombre réduit des micro services participant à la transaction distribuée. le nombre de déclencheurs d'événements n'est pas trop élevé .

— lorsque les actions de déclenchement ne sont pas trop complexes.

— L'orchestration est lente, mais elle est utile lorsque les scénarios de transaction sont com- plexes.

3.7 Mécanisme De Sélection Des Microservices Orchestration Vs Chorégraphie

En [68], Neha Singhalet all, ont effectué une analyse de la façon dont les techniques de chorégraphie et d'orchestration des microservices sont utilisées pour mettre en œuvre l'ar- chitecture des microservices. Une application médicale a été utilisé pour tester et réaliser les expériences pour rendre l'application plus efficace et impression- nante. L'étude a effectué une composition dynamique des microserviceset a évaluée le processus.

La modélisation des processus de microservice soulève de nouvelles questions difficiles, c.-à-d. quel mécanisme de service est bon pour la collaboration en matière de service. C'est un grand défi d'identifier quelle approche de composition est la meilleure parmi l'orchestration et la chorégraphie. Le but de l'étude est d'effectuer une analyse sur divers paramètres considérables tels que la consommation de temps, d'énergie et de mémoire dans le processus d'orchestration et de chorégraphie. La pertinence de l'orchestration et de la chorégraphie des microservices est abordée dans divers scénarios de modélisation.

L'étude est basée sur la création d'un processus opérationnel BPEL exécutable pour l'application médicale. Le code de l'application est écrit en Java et déployé sur le serveur Web Apache. Dans les scénarios, le processus métier BPEL reçoit une

demande de l'utilisateur. Pour y répondre, le processus appelle l'utilisateur concerné pour lui demander des services connexes à partir du registre des services, puis répond au client demandeur. L'invocation est gérée par le service composite.

Pour expliquer BPEL, on défini un processus métier simplifié pour une application de santé. Le client invoque la requête selon le centre de santé et donne la solution appropriée. on supposons que l'application de soins de santé fournit un service par l'intermédiaire duquel on recherchons des médecins et des services liés à la prise de rendez-vous. Enfin, le processus BPEL fournit la liste des services demandés au client. Nous construisons un processus BPEL synchrone. Le nouveau microservice composite de BPEL utilise un ensemble de ports différents à travers lesquels il fournit des fonctionnalités comme tous les autres services nouvellement conçus. Nous exécutons la composition du service de deux façons différentes, soit l'orchestration et la chorégraphie. La méthodologie du modèle proposé est décrite ci-dessous :

1- Les clients/consommateurs de services envoient des demandes pour disposer d'un ou plusieurs services au référentiel de microservices UDDI

2- Mise en place d'un registre de microservices comme UDDI pour les micro services.

3- Mise en place d'un hub en nuage pour le stockage des microservices participant à l'écosystème de services.

4- Les critères de sélection tels que les attributs typiques de la qualité de service (extensibilité, disponibilité,performance/ débit, sécurité, extensibilité, capacité de composition, etc.) pour sélectionner les différents microservices.

5- L'implication de la technologie de l'ontologie pour la sélection et l'utilisation automatiques de microservices adéquats et pertinents pour les processus d'affaires. et pertinents pour les processus métier. L'intervention humaine,l'interprétation et l'instruction ne sont pas nécessaires pour choisir les services appropriés à être composés.

6- Orchestration ou chorégraphie pour composer les microservices choisis afin

de créer des applications/charges de travail/processus composites.

7- La composition de plusieurs microservices conduit à des applications pilotées par les événements.

8- L'évaluation des performances du cadre composite.

Dans le présent document, diverses analyses comparatives sont effectuées pour la chorégraphie des microservices et les méthodes d'orchestration des microservices en fonction des trois paramètres suivants :

• Consommation de temps

• Utilisation de la mémoire.

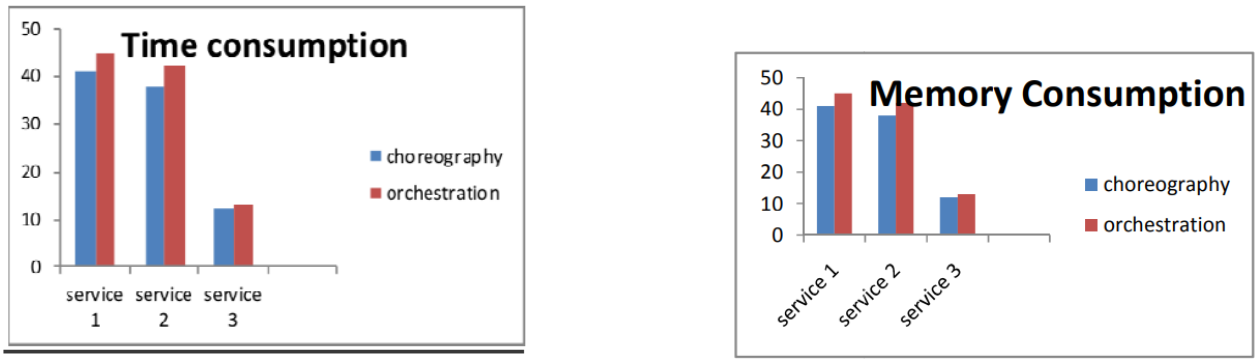

FIGURE 3.10 – Consommation de temps et de mémoire des différents services

L'analyse montre que le temps et la consommation de mémoire pour la chorégraphie microservice est moins comparable à la méthode d'orchestration

3.7.1 conclusion

la présente étude a mis en œuvre des approches dynamiques de composition de microservices, utilisant l'orchestration et la choréogrgraphie. Ce travail à effectue l'analyse entre l'or- chestration et la chorégraphie. L'analyse est effectuée en fonction de la consommation de temps, de mémoire. L'analyses des résultats a permet d', identifier clairement que la choré- graphie

événementielle est beaucoup plus rapide en performance fig 3.8 et la fig 3.9 Par rapport à l'orchestration

3.8 Un cadre d'interaction entre les bases de données et l'architecture des microservices

En [69], Mohamed El Kholy et all, ont fait une étude sur La migration d'applications monolithiques vers une architecture microservices qui apporte sur un certain nombre de défis ; le fractionnement et le partage des bases de données entre les microservices est un cas ?. L'article présente un cadre pour gérer le fractionnement de bases de données entre les Micorservices d'une application. pour maintenir la cohérence des données et assure un déploiement indépendant de Micro services tout en améliorant le temps de réponse du système.

Cette section présente différentes solutions :

3.8.1 Cloner La Base De Données

Cette approche crée un clone de la base de données d'origine pour chaque service .[70] Chaque microservice interagit avec sa base de données de manière isolée. L'approche de clonage nécessite plus de ressources pour stocker les données. La mise à l'échelle d'une partie de la base de données affecte tous les autres clones. La base de données de chaque microservice comprend toutes les tables et requêtes. Cependant, de nombreuses tables peuvent ne pas être associées à la logique métier de ce microservice. Maintenir la cohésion des données est un autre défi. Chaque champ mis à jour doit être diffusé à tous les autres clones. Toute nouvelle instance doit également être créée dans

toutes les tables de tous les clones.

3.8.2 Base De Données Privée (Base De Données ParService)

Dans la deuxième approche, principal défi consiste à assurer la cohérence des données tout en joignant les données de différentes bases de données appartenant à différents services. Un autre défi consiste à prendre en charge une interaction efficace entre un microservice et les données associées à un autre service. Dès lors, ce n'est pas applicable pour utiliser une seule interface d'application (API) pour agréger les bases de données pour tous les services. Richardson[28] a introduit un modèle de conception qui facilite l'interaction entre les différentes bases de données contenues dans leurs microservices. La base de données privée de chaque service est accessible par d'autres microservices uniquement via l'API du service. Dans une telle approche, toute mise à jour ou création d'une nouvelle instance d'un microservice base de données n'a aucun impact sur les autres services. Cependant, un tel environnement affecte l'autonomie des microservices car chaque service doit interagir avec la (passerelle API) pour exécuter sa fonctionnalité.[71] De plus, une telle conception pattern augmente la charge de l'application pour gérer la base de données.17 L'approche comprend plusieurs problèmes de sécurité pour authentifier chaque service avec l'API. Joindre des données à partir de données distribuées bases de données hétérogènes est un problème difficile. Viennot et al. introduit Synaps comme solution pour l'interaction entre différents microcervices.[72] Synaps prend en charge le partage de données entre différents bases de données associées à différents services. Cependant, synaps dépend d'un nœud central qui gère l'encapsulation et la gestion des données provenant de différents services. Un tel nœud central crée un seul point de

défaillance et a un effet négatif de l'efficacité du système.

Solution MDMA

La solution proposée divise une grande base de données complexe associée à une ap- plication monolithique en bases de données simples plus petites correspondant aux limites commerciales de chaque microservice. Les avantages de la solution proposée de ce travail par rapport aux solutions rapportées dans la littérature sont les suivants :
Permettre à chaque service d'interagir avec les données associées à sa fonctionnalité tout en conservant l'esprit du Microservice pour déploiement indépendant.

Chaque service est déployé avec sa base de données séparément sans aucune dépendance vis-à-vis d'autres services.

Cependant, après le déploiement, un service publie les informations qui permettent à tout autre service d'interagir avec sa base de données. Permettre une interaction croisée entre un microservice et des bases de données associées à d'autres microservices tout en maintenant l'efficacité et la cohérence des données. La solution proposée prend en charge l'interaction des microservices avec plusieurs bases de données en évitant la dépendance à une passerelle API centrale. Cela conduit à une exécution plus efficace car les demandes d'interaction ne sont pas regroupées dans un nœud central.

Le processus d'évaluation examine l'amélioration de l'efficacité sans enfreindre les concepts de microservices tels que le « déploiement de services indépendants ». Les mé- triques choisies et les critères de choix sont listés ci-dessous.

Temps De Réponse

Le temps de réponse mesure l'intervalle de temps entre la demande du client et la réponse de la candidature. Le temps de réponse est divisée intervalles. Tout d'abord, le message est passé de le client à la passerelle d'application. Seconde, la requête est transférée de la passerelle vers le côté application.Dans le cas d'un nœud central approche, le message est dirigé vers le centre nœud où la demande est gérée et traitée. Dans l'approche proposée, la demande est adressée de la passerelle au service requis qui est capable d'exécuter la fonctionnalité demandée. La séquence de messages est illustrée à la Figure 3.11, L'amélioration se concentre sur le backend de l'application où les services effectuent les calculs requis et communiquent entre eux.

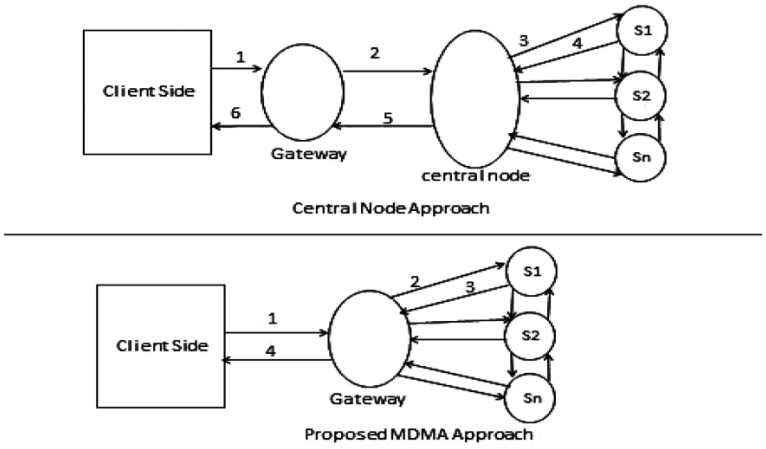

FIGURE 3.11 – message path

Taille Des Messages Transférés

Microservices l'application est une application basée sur le réseau. Moins le réseau est utilisé, plus l'efficacité des applications est élevée. Le cadre proposé est évalué en calculant la taille totale des messages pour les différentes fonctionnalités offertes par l'application proposée. Ensuite, la taille des messages est comparée à celle de l'approche du nœud central qui est rapportée dans les travaux connexes.

3.9 Contribution

La récente croissance de la popularité et de l'acceptation de l'architecture des micro- services dans l'industrie a fait de ce style architectural souvent décrit comme une forme de modernisation des applications qui résout tous les inconvénients des applications mo- nolithiques traditionnelles.

Notre contribution consiste é développer une application simple basée sur l'architecture de microservices avec un modèle évent-driven construit avec des frameworks flask et django, et hébergé dans des conteneurs docker. L'application est évaluée sous une forte charge de demandes et une évaluation des performances est élaborée. les résultats présentés.

3.10 Conclusion

Dans ce chapitre, nous avons discuté les différents concepts autour des bases de données dans les architectures microservices. Nous avons présenté les différents patterns utilisés, nous avons détaillé et examiné quelques travaux de recherche connexes et nous avons parlé de notre contribution dans le domaine.

Chapitre 4

Implementation et Tests

4.1 Introduction

Dans ce chapitre, nous présenterons la mise en œuvre de notre application basée sur une architecture de microservices, y compris les outils que nous avons utilisés pour la créer, nous examinerons également les performances de notre application en la déployant sur un serveur local et en la testant avec une charge élevée de requêtes. Ensuite, nous discuterons les résultats avant de conclure.

4.2 Environnements de travail

Afin de développer notre application , différentes bibliothèques, environnement de développement et langage ont été utilisés.

4.2.1 Environnement matèriel

Pour réaliser ce travail, nous avonsa utilisé un ordinateur Hp Elitebook 840 G5 Intel(R) Core(TM) i5-7300U caractérisé par :
- CPU @ 2.60GHz 2.71 GHz

- 8 Go de RAM

- 256 Go de Disque dure

- Un système d'exploitation Windows 11 64 bits .

4.2.2 Environnement logiciel

Visual Studio Code

Visual Studio Code est un éditeur de texte qui intègre la coloration syntaxique du code source pour les langages et fichiers C++,Python, C, JavaScript, HTML, PHP,CSS,...etc. comme il nous offre :

— Plusieurs onglets pour ´editer plusieurs fichiers dans la meme fenetre.

— Choix du Codage de caractères (ANSI, UTF-8, UCS-2)

— Glisser-déposer.

— Numérotation des lignes,...etc.

GitHub

GitHub est une plateforme de controle de version et de collaboration basée sur le web pour les développeurs de logiciels. De type SaaS (Software as a Service) , ce service en ligne est utilisé pour stocker le code source d'un projet et suivre l'historique complet de toutes les modifications apportées à ce code. Il permet aux développeurs de collaborer plus efficacement sur un projet en fournissant des outils pour gérer les changements éventuellement contradictoires de plusieurs développeurs.

Docker

Docker nous permet de travailler dans des environnements isolés appelés conteneurs, capables de communiquer entre eux et pouvant être déployés

facilement sur nos serveurs. [73] Dans cas de conteneurs docker nous disposons d'un système d'exploitation et des ressources qui sont partagés entre les conteneurs. Docker est une plateforme logicielle, destinée aux développeurs et administrateurs systèmes. Dans le but de faciliter le développement, la diffusion et le déploiement d'applications. Il repose sur le noyau linux et ses fonctionnalités de virtualisation par conteneurs,[74] il s'appuie notamment sur :

— Groupe qui est un composant pour Contrôler et limiter l'utilisation des ressources pour un ou plusieurs processus (utilisation de la RAM CPU entre autres).

— Espace de noms (namespace) permet de créer des environnements sécurisés d'une manière à isoler les conteneurs.

La figure 4.1 présente l'architecture de docker

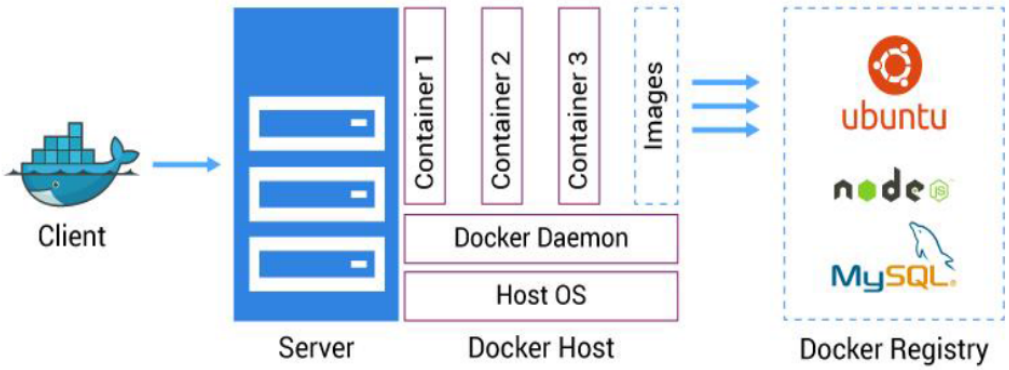

FIGURE 4.1 – L'architecture docker

MQrabbit

La solution de courtage la plus répandue et probablement la plus connue, RabbitMQ, a été publiée en 2007 et a depuis été utilisée par de nombreuses entreprises comme un courtier de message classique et comme middleware orienté message. Le grand nombre de langages de programmation et de

protocoles de messagerie pris en charge a certainement contribué à sa popularité : RabbitMQ implémente le Advanced Messaging Queuing Protocol (AMQP), mais prend également en charge Message Queuing Telemetry Transport (MQTT) et WebSockets. Il y a des bibliothèques pour Java, . NET, Python et beaucoup d'autres qui sont absolument nécessaires pour pouvoir communiquer via le courtier. En tant que courtier classique, RabbitMQ est conçu pour traiter les opérations à haute disponibilité, mais il est clair que RabbitMQ n'a pas été conçu nativement pour les opérations à évolutivité élastique : une seule instance est déjà très Les instances prennent quelques minutes pour devenir disponibles , RabbitMQ prend en charge les garanties de livraison au plus une fois et au moins une fois, ainsi que la conservation des commandes de messages [75]. Ce support étendu des protocoles et des

FIGURE 4.2 – MQrabbit

langages de programmation est un net avantage de RabbitMQ. La mise en œuvre de l'AMQP rend le courtier une solution puriste qui est très proche de l'idée conceptuelle du médiateur , Néanmoins, l'installation est simple et le fonctionnement léger est possible, faire de RabbitMQ un candidat appro-

prié pour fournir l'infrastructure pour le nouveau style d'interaction entre les services.

React js

React est une bibliothèque JavaScript déclarative open-source, efficace et flexible pour construire des interfaces utilisateur. React permet de créer des interfaces utilisateur complexes avec de petits ensembles de code isolés appelés "composants". React JS est utilisé pour gérer la couche de vue dans les applications à page unique et le développement d'applications mobiles. React JS est géré par facebook, instagram, communauté de développeurs et les entreprises. React s'efforce de fournir la vitesse, la simplicité et l'évolutivité. Certaines des caractéristiques les plus frappantes sont JSX, Stateful Components, Virtual Document Object Model. l'utilisation de React JS en développement facilite le processus de création d'interfaces utilisateur interactives. L'écriture de la syntaxe déclarative React JS rend le flux de code plus prévisible et plus facile à déboguer. React JS est basé sur des composants qui peuvent créer plusieurs composants encapsulés qui définissent leur propre état, puis ces composants sont combinés pour former des interfaces utilisateur plus complexes. Cela rend également plus facile pour les développeurs lors de la maintenance lorsque les choses vont mal parce que les développeurs vont directement à la composante problématique sans déranger d'autres composants de sorte qu'il est non seulement facile, mais aussi plus rapide. En outre, les composants qui ont été écrits peuvent être réutilisés lorsque le développeur en a besoin, ce qui permettra d'économiser du temps et de l'efficacité du code afin qu'il ne soit pas écrit à plusieurs reprises.[76]

Langage HTML

HTML (HyperText Markup Language) a fait son apparition des 1991 lors du lancement du Web. Son role est de gérer et organiser le contenu des pages web. C'est donc en HTML que écrirez ce que vous souhaitez que la page affiche du texte, des liens, des images. Nous l'utiliserons pour l'élaboration des pages statiques qui constituent l'interface utilisateur.

Langage CSS

CSS (Cascading Style Sheets, aussi appelées Feuilles de style) joue un role important dans la gestion et l'apparence de la page web (agencement, positionnement, décoration, couleur et taille du texte). Nous utiliserons ce langage pour rendre les pages programmées en HTML plus agréables et leurs donner un style spécifique.

Langage JavaScript

JavaScript est un langage de programmation de scripts principalement employé dans les pages web.[77] Par exemple, JavaScript permet de tester que les champs obligatoires sont bien remplis sans avoir besoin d'envoyer le formulaire au serveur pour qu'il le vérifie. L'avantage dans ce cas là est de limiter le nombre de requetes envoyées au serveur.

Langage JSON

JSON (JavaScript Objet Notation) est un langage léger d'échange de données textuelles. Pour les ordinateurs, ce format se génère et s'analyse facilement . Pour les humains, il est pratique à écrire et à lire grace à une syntaxe simple et à une structure en arborescence. JSON permet de représenter des

données structurées (comme XML par exemple).

Python

En soi, Python est un excellent langage de "pilotage" pour les codes scientifiques écrits dans d'autres langages. Cependant, avec des outils de base supplémentaires, Python se transforme en un langage de haut niveau adapté au code scientifique et technique qui est souvent assez rapide pour être immédiatement utile mais aussi assez flexible pour être accéléré avec des extensions supplémentaires, il possède des fonctionnalités uniques offrant un environnement qui rend C'est un meilleur choix pour les scientifiques et les ingénieurs à la recherche d'un langage de haut niveau pour l'écriture d'applications scientifiques. il y a une multitude de raisons pour lesquelles Python excelle en tant que plateforme de calcul scientifique. et cette liste de fonctionnalités générales : [78]

Une licence open source libérale permet de vendre, d'utiliser ou de distribuer votre application basée sur Python comme bon vous semble, sans autorisation supplémentaire.

• Le fait que Python s'exécute sur autant de plates-formes signifie que vous n'avez pas à vous soucier d'écrire une application avec une portabilité limitée, ce qui permet également d'éviter la dépendance vis-à-vis d'un fournisseur.

• La syntaxe propre du langage mais ses constructions sophistiquées vous permettent d'écrire de manière procédurale ou entièrement orientée objet, selon la situation.

• Un interpréteur interactif puissant permet le développement de code en temps réel et l'expérimentation en direct, éliminant ainsi l'étape de compilation chronophage et gourmande en productivité du processus de développe-

ment codethen-test.

• La possibilité d'étendre Python avec votre propre code compilé signifie que Python peut apprendre à faire n'importe quoi aussi vite que votre matériel le permet.

• Vous pouvez intégrer Python dans une application existante, ce qui signifie que vous pouvez instantanément ajouter un placage facile à utiliser au-dessus d'une application plus ancienne et fiable.

• La possibilité d'interagir avec une grande variété d'autres logiciels sur votre système vous aide à tirer parti des compétences logicielles que vous avez déjà acquises.

StarUML

StarUML est un logiciel de modèlisation UML rapide, flexible, disponible en open source. Objectif du projet StartUML est de construire un outil de modélisation de logiciels et aussi la plate- forme qui est un remplacement convaincante d'outils UML commerciaux.

4.2.3 Frameworks

Django rest framwork

est un framework Python qui étend Django. DRF ajoute des sérialisateurs d'objets, plus d'options d'authentification et des ensembles de vues améliorés pour aider les développeurs à concevoir des API RESTful (vues combinées). En plus des fichiers Django, DRF utilise ces fichiers supplémentaires : Répertoire des applications : – serializers.py définit les conversions entre des données complexes, telles que des requêtes ou des résultats JSON et des types de données Python natifs.

SQLAlchemy

Un framework ORM est couramment utilisé dans les applications Python lorsque vous travaillez avec des bases de données. D'autres frameworks légers doivent nécessiter une bibliothèque autonome, mais Django inclut un outil ORM Django intégré qui peut être intégré en douceur dans les microservices Django. SQLAlchemy [79] est un framework Python ORM populaire qui prend en charge un large éventail de fonctionnalités. SQLAlchemy convertit les requêtes Python sur les classes en instructions SQL en utilisant des classes Python qui sont mappées à des tables de bases de données SQL. SQLAlchemy utilise des pilotes tiers pour un accès direct à la base de données afin de fournir une interface uniforme pour de nombreux formats de base de données. Le pilote sqlite39, par exemple, peut gérer les bases de données SQLite, tandis que psycopg210 peut contrôler les bases de données PostgreSQL. Toutes les opérations de base de données dans les microservices Starlette ont été effectuées avec SQLAlchemy.

Flask-RESTful Web Service Framework

Les microservices RESTful sont construits en utilisant une variété de frameworks, y compris Node.js Java Script, Flask[80] en Python, et Spring en Java. Nous avons choisi le cadre Flask-RESTful pour la mise en œuvre des services AGT parce qu'il a une intégration au niveau natif avec le cadre d'intégration d'outils Siemens, qui est également écrit en Python. Pour utiliser les modules d'un cadre d'intégration d'outils existant dans la construction du service d'exécution de flux de travail, un cadre de développement de services Web écrit en Python est nécessaire. Flask [81] est un cadre Python populaire pour le développement de services en ligne. Il est conçu pour faciliter

le démarrage tout en vous permettant de grandir pour créer des applications web complexes. Flask-RESTful est un emballage léger pour le cadre Flask qui simplifie la création de services web RESTful avec Flask.

4.3 Présentation de l'application

L'application développée consiste à un réseau social simple pour le partage des images. L'administrateur partage des images et les utilisateurs réagissent par des mentions « J'aime »

L'architecture générale de notre application basée sur les microservices, contient deux parties principales :
côté administrateur et côté utilisateur, chaque côté est construit sur 3 micro-services :
un backend, des base de données et une file d'attente.
le backend est responsable de l'envoi et de la réception des requêtes (j'aime) la base de données est responsable de la sauvegarde des données produit , get , update ,create , supprimer la file d'attente est responsable de la gestion des événements envoyés et reçus sur rabbit MQ le cas d'utilisation de notre programme est que l'administrateur peut créer des produits , modifier , consulter le nombre de Jaime ou supprimer un produit , l'utilisateur peut obtenir les produits créés par l'administrateur , et les aime un utilisateur ne peut pas aimer le même produit deux fois
Le service responsable de la gestion des événements entre la base de données de l'utilisateur et de la base de données de l'administrateur est le service de file d'attente « rabbit MQ ».

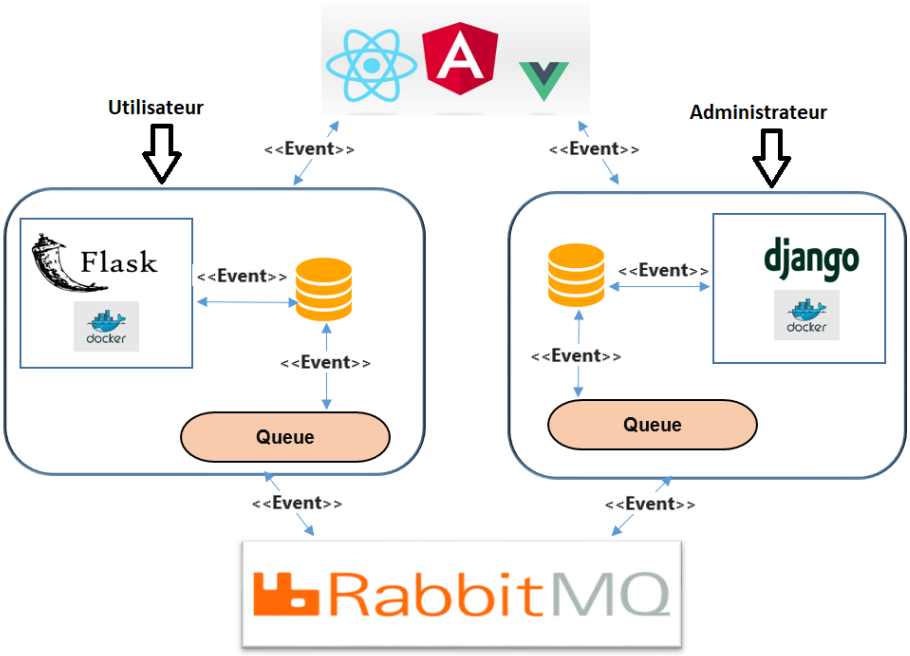

FIGURE 4.3 – Architecture de l'application dévelopée

4.3.1 Conception est modélisation du système

Pour concevoir notre application nous avons suivi le Processus et pour la modélisation nous avons opté par les diagrammes d'UML.

Le Processus Unifié (UnifiedProcess) UP :

Est un processus itératif et incrémental de développement logiciel, il est centré sur l'architecture, conduit par les cas d'utilisation et piloté par les risques. L'objectif d'untel processus, est de maitriser la complexité des projets informatiques en diminuant les risques. **La modélisation UML de notre système :**

Afin de modéliser notre système, nous avons implémenté le diag de cas d'utilisation et le diag de séquence :

Diagrame de cas d'utilisation

Un cas d'utilisation représente un ensemble de séquence d'actions qui sont réalisé par le système dont la finalité est de rendre un service à un acteur.

L'administrateur fig 4.4 joué un rôle très important dans la gestion de notre système, cette gestion est représentée en :

— Ajoute produit

— Modifier produit

— Supprime produit

— Consulte la liste des produits

L'utilisateur a le droit de :

— Consulte la liste des produits

— J'aime les produits

— Création de compte

Le cas d'utilisation « Authentification » : permet à l'administrateur et aux utilisateurs de s'authentifier pour accéder à l'application selon leurs profils.

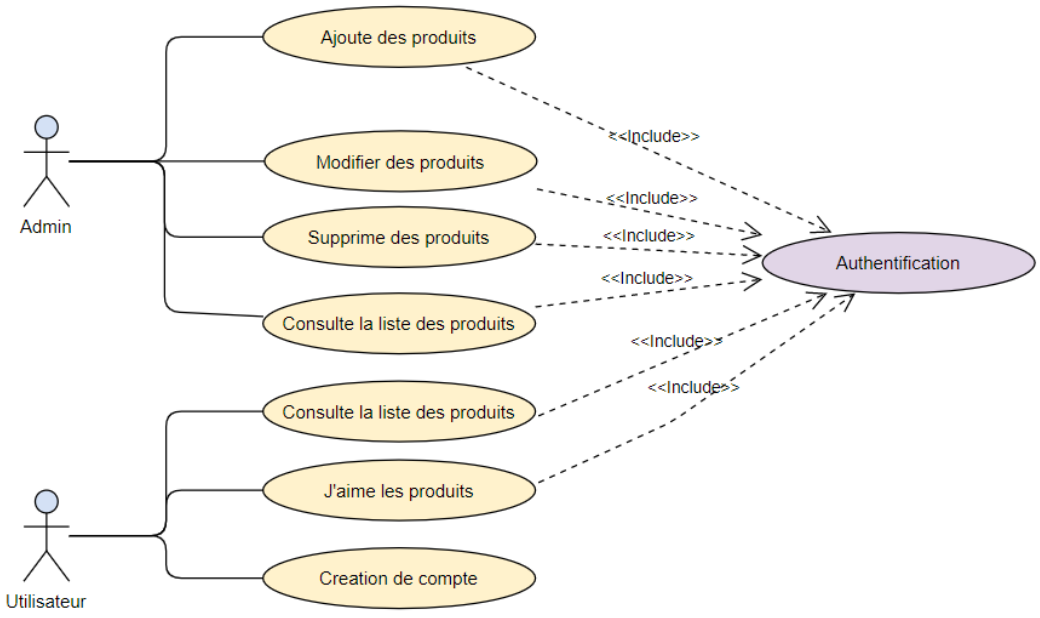

FIGURE 4.4 – Diagramme du cas d'utilisation

Diagramme de séquence

Les diagrammes de séquence montrent des interactions entre objets selon un point de vue temporel. Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique,La fig 4.5 présente le diag de séquence de notre systeme.

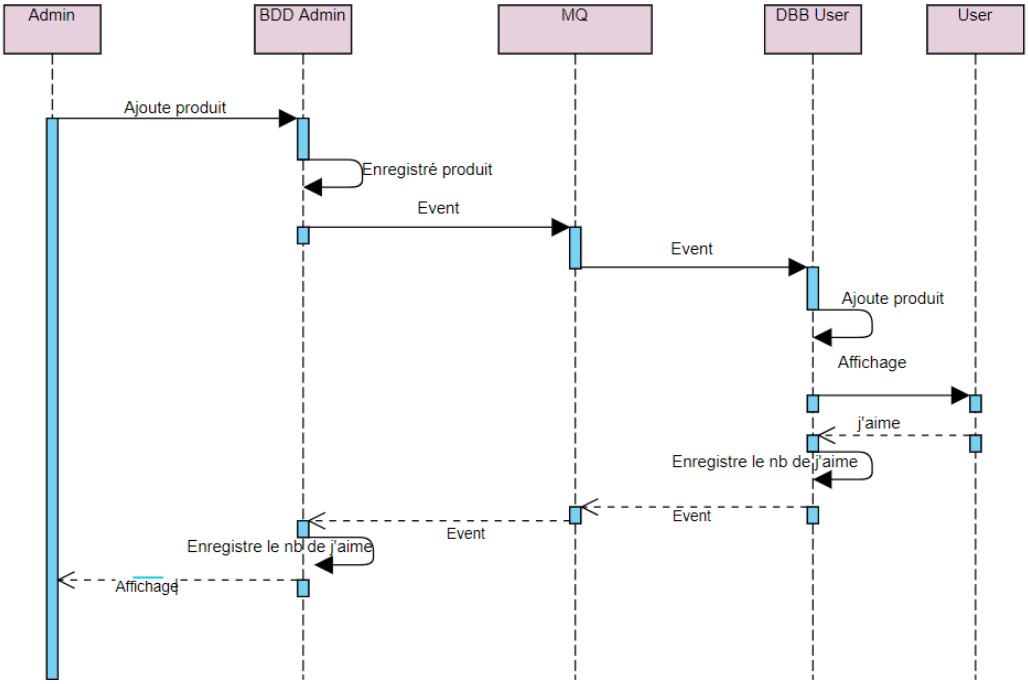

FIGURE 4.5 – diagramme de séquence

4.3.2 Scenario de fonctionnement et présentation graphique

— **L'interface de connexion**

Cette interface permet aux différents utilisateurs de se connecter. Si l'utilisateur donne un email et un mot de passe correct, il accède à sa page d'accueil, sinon un message d'erreur lui est envoyé. La figure 4.6 ci-dessous illustre la page de connexion de notre application.

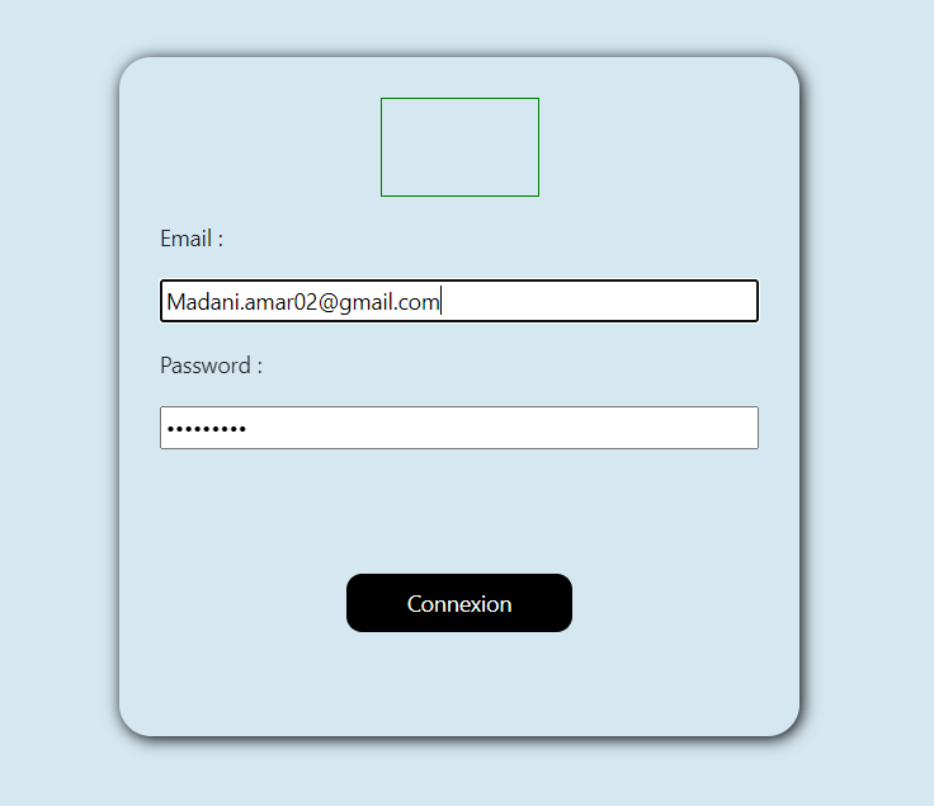

FIGURE 4.6 – Interface de connexion

— **L'interface page d'accueil**

Cette page contient les images publiées par l'admin et permet à l'utilisateur d'ajouter des mention « J'aime » pour les différentes images. La figure 4.7 ci-dessous illustre cet interface.

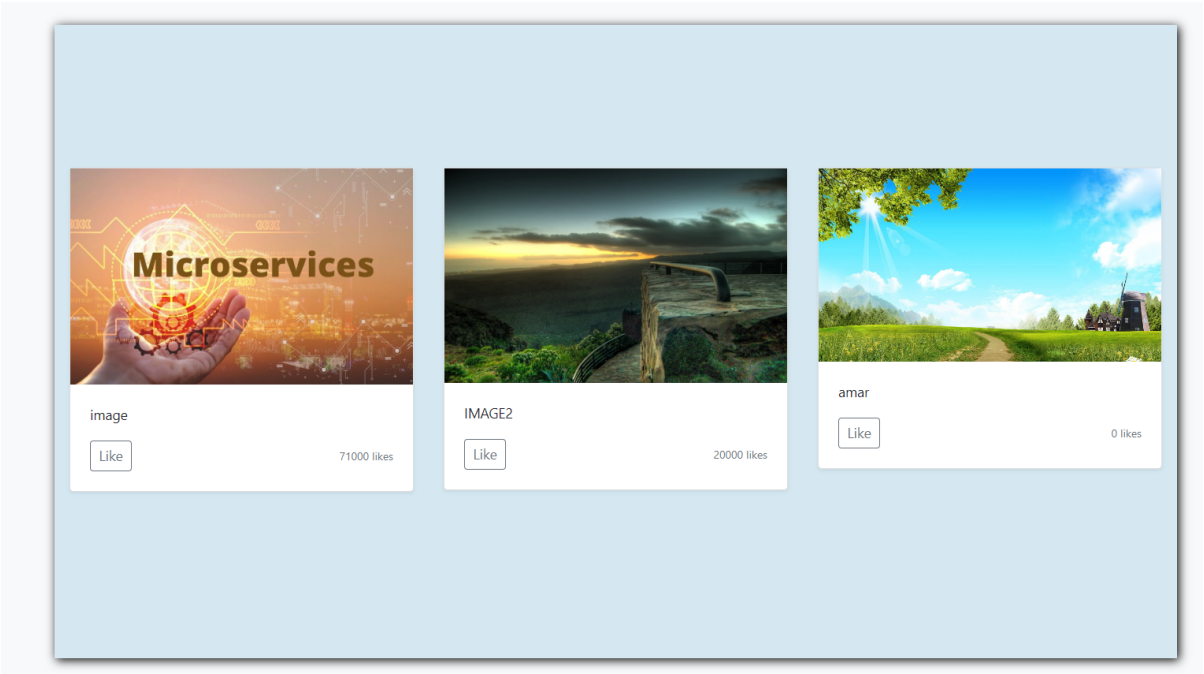

FIGURE 4.7 – Page d'accueil de l'utilisateur

— **L'interface pour l'administrateur du système**

A partir de cette interface, l'administrateur peut ajouter, modifier et supprimer des produits. La figure 4.8 ci-dessous permet d'illustrer cette interface.

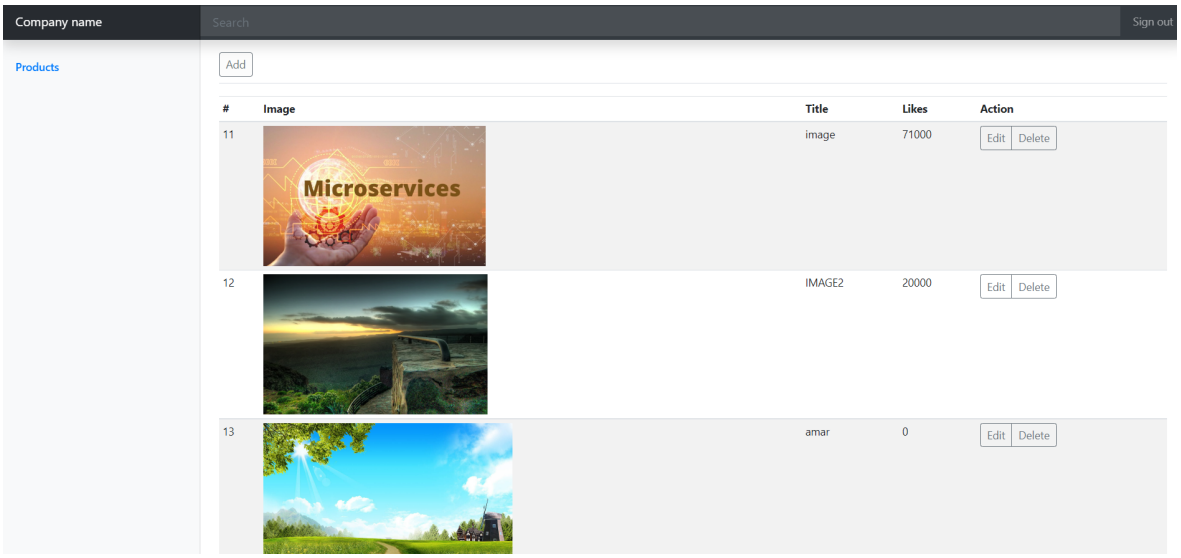

FIGURE 4.8 – Page de l'administrateur

— **Ajoute des produits**

Cette interface permet l'administrateur à ajouter une image d'un produit et à la nommer. La figure 4.9 ci-dessous permet d'illustrer cette interface.

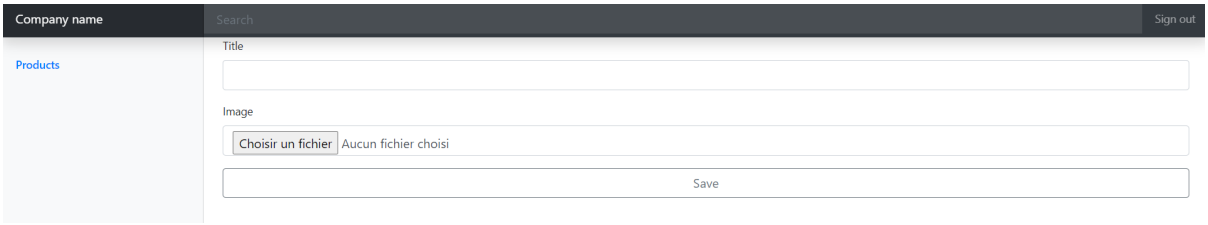

FIGURE 4.9 – interface pour ajout des produits

4.3.3 Code de source

Nous présentons ci-dessous les codes sources de notre application

Dockerfiles

contient un ensemble d'instructions qui spécifient l'environnement à utiliser et les commandes à exécuter.

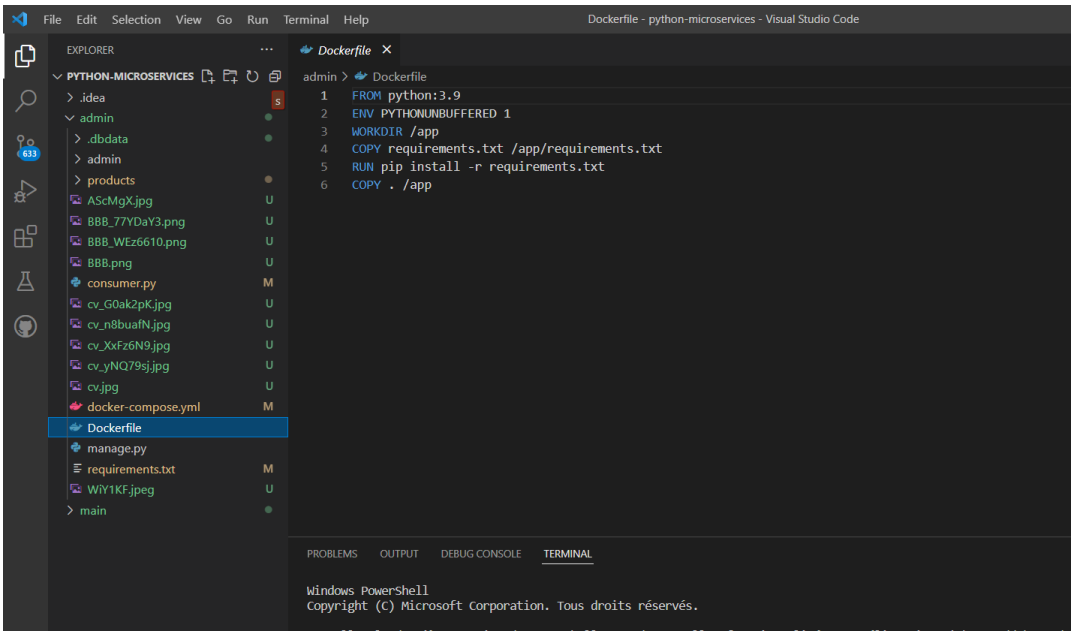

FIGURE 4.10 – dokerfile

cette ligne **FROM python :3.9** , pour l'utilisation de l'image de base de Python 3 comme point de lancement.

Doker compose

Compose est un outil pour définir et exécuter des applications Docker multi-conteneurs. Avec Compose, nous utilisons un fichier YAML pour configurer les services de votre application. Ensuite, avec une seule commande, vous créez et démarrez tous les services à partir de votre configuration.

```yaml
db:
  image: mysql:5.7.22
  restart: always
  environment:
    MYSQL_DATABASE: 'admin'
    MYSQL_USER: 'root'
    MYSQL_PASSWORD: 'root'
    MYSQL_ROOT_PASSWORD: 'root'
  volumes:
    - .dbdata:/var/lib/mysql
  # command: "mysql -u root -p root -host 127.0.0.1 -P 3306 --protocol=tcp"
  ports:
    - '3306:3306'
  expose:
    - '3306'
```

FIGURE 4.11 – docker compose

db utilise une image MySQL officielle de DockerHub. Nous attachons le port local 3306 au port 3306 de ce conteneur. Nous définissons ensuite un ensemble de variables d'environnement requises pour MySQL.
./db :/var/lib/mysql crée un dossier db local dans votre racine de projet afin que les informations de la base de données puissent être enregistrées si le service db est détruisé .

Tableau de product

Nous avons créé un modèle de base de donnees appelé Produit , ajoutons quelques champs au modèle Produit (image ,titre ,like)

image = models.ImageField() permet de associer des images à notre modèles.

FIGURE 4.12 – product models

Utilisateur

db.create-all() est une instruction demandant à l'application de créer toutes les tables dans la base de données spécifiée(user). Ainsi, dès l'exécution de l'application, toutes les tables seront créées si elles ne sont pas déjà là.

@app.route('/api/products') permet de cree un lien (endpoint) pour récupérer tout les produits de base de donnée de l'admin.

@app.route('/api/..',methods=['POST']) cette instruction permet de cree un lien (endpoint) pour faire des j'aimes à les produits .

```python
db.create_all()

@app.route('/api/products')
def index():
    return jsonify(Product.query.all())


@app.route('/api/users/<userid>/products/<int:id>/like', methods=['POST'])
def like(userid,id):


        check_exist = db.session.query(ProductUser).filter(ProductUser.user_id == userid, ProductUser.product_id == id).first()
        if check_exist is not None:
            return jsonify({
            'message': 'already liked'
        })
        productUser = ProductUser(user_id=userid, product_id=id)

        db.session.add(productUser)
        db.session.commit()
        publish('product_liked', id)
        print(userid)


        return jsonify({
            'message': 'success'
        })
```

FIGURE 4.13 – user main

4.3.4 Tests et résultats

Tests

Nous avons lancé six cas de tests. Les différents cas sont réalisés par l'envoi des requêtes asynchrones à une seule image

— 1 000 requêtes effectuées à la fois.

— 10 000 requêtes effectuées à la fois.

— 20 000 requêtes effectuées à la fois.

— 40 000 requêtes effectuées à la fois.

— 60 000 requêtes effectuées à la fois.

— 80 000 requêtes effectuées à la fois.

Les tests ont été effectués sur ordinateur avec le système d'exploitation windows 11. Les applications ont été déployées sur Docker. L'ordinateur a les paramètres suivants :

— Processor G5 Intel(R) Core(TM) i5-7300U

— Ram 2.60GHz 2.71 GHz RAM 8 Go

— Browsers : Opera,google chrome.

Résultats

Après lancement des différents tests, nous avons récupérer le temps de réponse (latence du système). Les résultats sont présentés sur la fig 4.14 . La figure 4.14 représente l'évolution du la latence en fonction du nombre de requêtes.

Nous remarquons qu'il y a deux parties dans l'évolution de la latence, La première partie correspond au nombre de requêtes [1000, 20000] où la latence

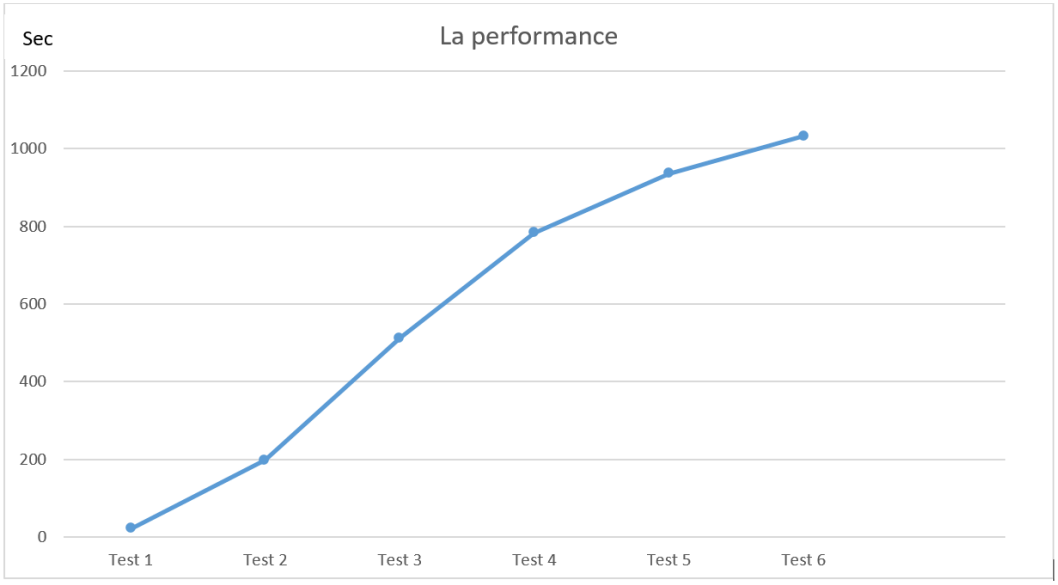

FIGURE 4.14 – Performances de l'application développée

N° de Test	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
N° de Requêtes	1000	10000	20000	40000	60000	80000
Temps de traitements (sec)	21.6	198.3	511.5	783.2	936.1	1032.5

Tableau 2. Temps de réponse - résultats

évolue linéairement avec le nombre de requêtes,dans cette partie plus ce que le nombre de requêtes plus,le temps de traitement est long. La deuxième partie, correspond au nombre de requêtes [20000, ...],nous remarquons dans cette partie que la courbe tend à être stable. Même si le nombre de requêtes est devient important, elles sont traitées dans un délai raisonnable.

Conclusion

Afin de tester les performances de l'architecture que nous avons présentée, six cas de test représentés dans le tableau 2 ont été réalisés, qui consiste à envoyer un tas de requêtes asynchrones a une seule image et le nombre de ces requêtes est doublé à chaque test.

Test de performance sont illustrés sur la figure 4.14 qui montre que l'archi-

tecture de microservices est scalable, elle peut faire face à une grande quantité de trafic d'événements, plus le nombre de requêtes est important, plus elles sont traitées dans le moins de temps possible et acceptable, contrairement l'architecture monolithique où la latence augment selon l'augmentation du nombre de requête.

4.4 Conclusion

L'objectif de ce chapitre a été implémentation de toutes les méthodes définis et étudiées dans les chapitres précédents. Dans la phase "Implémentation" nous avons montré les différentes étapes de réalisation de notre système ainsi que le résultat finals.Nous avons également montré le code sources, et nous avons teste notre application en le mettant sous charge des requêtes asynchrones,Les résultats prouve la scalabilité des architecture microservices, plus il y a de charge de requêtes, la latence raisonnable et l'efficacité est meilleure, et c'est l'un des avantages des microservices dont nous avons parlé précédemment.

Conclusion générale

Au terme de nos travaux, nous espérons avoir atteint l'objectif que nous nous étions fixé, qui est de réaliser une application basée sur l'architecture de microservices supportée par le style Event Driven. Cette structure permet essentiellement aux programmeurs de faciliter la maintenance en cas de dysfonctionnements, ainsi que moins de temps de traitement des informations en cas de compression des informations,surtout que les géants du web comme Netflix et Amazon la trouvent efficace et fiable. En effet entreprises doivent désormais comprendre que le choix architectural est un enjeu très important qui conditionne la performance de leurs activités Il garantit la crédibilité de son travail de programmation et répond également à ses exigences. Nous espérons que cette application provence tout les avantage et comment foctionne cette architecture on à travers cette étude les bases exigées dans le style architectural microservice sont :

— Chaque microservice est déployé séparément. En cas de modification impactant le périmètre d'un microservice on le redéployait seul,c est a dire que chaque service est indépendant en soi et est responsable de ses résultats et de son efficacité.
— Chaque microservice est dédié à réaliser une tâche bien spécifique, donc

la durée des tests est minimale.il traite un grand nombre de demandes en un temps record.

— Pour chaque microservice nous choisissons les technologies qui conviennent mieux à sa réalisation selon le besoin.

Bibliographie

[1] M. Fowler. “monolithfirst”. [Online ; accessed May 29, 2022].

[2] J. Lewis and M. Fowler. “microservices. a definition of this new architectural term.”. [Online ; accessed May 29, 2022].

[3] C. Richardson. “ “pattern : Monolithic architecture”. [Online ; accessed May 29, 2022].

[4] Ciera Jaspan, Matthew Jorde, Andrea Knight, Caitlin Sadowski, Edward K Smith, Collin Winter, and Emerson Murphy-Hill. Advantages and disadvantages of a monolithic repository : a case study at google. In *Proceedings of the 40th International Conference on Software Engineering : Software Engineering in Practice*, pages 225–234, 2018.

[5] Judith Hurwitz Carol Baroudi Robin Bloor, Marcia Kaufman. what is soa. In *Service Oriented Architecture for Dummies*, pages 5–9, 2006.

[6] Falko Menge. Enterprise service bus. In *Free and open source software conference*, volume 2, pages 1–6, 2007.

[7] James Lewis and Martin Fowler. Microservices. Elsevier, march 2014.

[8] James Lewis and Martin Fowler. trends. Elsevier.

[9] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. Design, monitoring, and testing of microservices

systems : The practitioners' perspective. volume 182, page 111061. Elsevier, 2021.

[10] Sudhir Tonse. Microservices at netflix-challenges of scale. Elsevier, August 2014.

[11] Phil Calcado. Building products at soundcloud—part iii : Microservices in scala and finagle. Elsevier, june 2014.

[12] Microservices. Elsevier, (visité le 30/10/2019).

[13] J. Thones. "microservices," in ieee software, vol32. pages pp.115–116. Elsevier, Oct. 15, 2018.

[14] Olaf Zimmermann. Microservices tenets. volume 32, pages 301–310. Springer, 2017.

[15] Chongkun Xia, Yunzhou Zhang, Lei Wang, Sonya Coleman, and Yanbo Liu. Microservice-based cloud robotics system for intelligent space. volume 110, pages 139–150. Elsevier, 2018.

[16] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and api gateways in microservices. 2016.

[17] M. Fowler J. Lewis. "microservices a definition of this new architectural term". Elsevier, 2014.

[18] S.Larsenz M.Mazzara N. Dragoni, S.Dustdary. "microservices :migration of a mission critical system". Elsevier, 2017.

[19] lien. Elsevier.

[20] Sam Newman. Advantages and disadvantages of a microservise. In *Monolith to Microservices Evolutionary Patterns to Transform Your Monolith*, pages 06–09, 2018.

[21] S. Newman. Bulding microservices : Design finegrained systems. pages 35–40, 2016.

[22] Moisés Macero García. Learn microservices with spring boot : A practical approach to restful services using an event-driven architecture, cloud-native patterns, and containerization. page 149–214, November 7, 2020.

[23] Sam Newman. Advantages and disadvantages of a microservise. In *Monolith to Microservices Evolutionary Patterns to Transform Your Monolith*, pages 06–09, 2018.

[24] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[25] Markos Viggiato, Ricardo Terra, Henrique Rocha, Marco Tulio Valente, and Eduardo Figueiredo. Microservices in practice : A survey study. 2018.

[26] Claus Pahl and Pooyan Jamshidi. Microservices : A systematic mapping study. pages 137–146, 2016.

[27] Javad Ghofrani and Daniel Lübke. Challenges of microservices architecture : A survey on the state of the practice. volume 2018, pages 1–8, 2018.

[28] Chris Richardson. "microservices — pattern : Monolithic architecture". Mars,2017.

[29] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices : State of the practice, challenges, and research directions. 2021.

[30] M. Fowler. “microservice trade-offs,” . 2018.

[31] Kubernetes. ““kubernetes,”. 2018.

[32] J. Willis G. Kim, P. Debois and J. Humble. “the devops handbook : How to create world-class agility, reliability, and security in technology organizations”. 2016.

[33] I. Weber L. Bass and L Zhu. Devops : A software architect’s perspective. In *Addison-Wesley Professional,*, pages 4–8. IEEE, 2015.

[34] F. Willnecker A. Danciu W. Hasselbring C. Heger N. R. Herbst P. Jamshidi R. Jung J. von Kistowski A. Koziolek J. Kroß S. Spinner C. V¨ogele J. Walter A. Brunnert, A. van Hoorn and A. Wert. “performance-oriented devops : A research agenda,” corr, vol. In *abs/1508.04752*. IEEE, 2015.

[35] J. Humble and D. Farley. Continuous delivery : Reliable software releases through build, test, anddeployment automation. In *Addison-Wesley Professional*. IEEE, 2010.

[36] W. Hasselbring J. Waller J. Ehlers S. Frey and D. Kieselhorst A. van Hoorn, M. Rohr. “continuousmonitoring of software services : Design and application of the kieker framework,”. In *research report, KielUniversity*. IEEE, novembre 2015.

[37] Gray J. The transaction concept : Virtues and limitations, in proc. In *the 7th VLDB,Cannes*, pages 144–154.

[38] Krakowiak S. Balter R., Banâtre J.P. Construction des systèmes d’exploitation répartis. In *Collection didactique éditée par INRIA, France*, 1991.

[39] J. Maron M. Little and G. Pavlik. Java transaction processing. In *Prentice Hall*, pages 20–25, 2014.

[40] M. Musgrove. “narayana + wildfly,”. 2015.

[41] T. Haerder and A. Reuter. “principles of transaction-oriented database recovery,”. page 287–317.

[42] M. Kleppmann. Designing data-intensive applications : The big ideas behind reliable, scalable, and maintainable systems. 2017.

[43] Cyrille Ponchateau. Conception et exploitation d’une base de modèles : application aux data sciences. 2018.

[44] J. Maron M. Little and G. Pavlik. Java transaction processing. prentice hall. 2014.

[45] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. volume 13, pages 185–221. ACM New York, NY, USA, 1981.

[46] A. D. Kshemkalyani and M. Singhal. Distributed computing : Principles, algorithms, and systems. 2016.

[47] João Carlos Ribeiro Dias Neves. Technical challenges of microservices migration. 2019.

[48] Nadia Nouali, Habiba Drias, and Anne Doucet. Revisiting distributed protocols for mobility at the application layer. In *WEC (5)*, pages 45–48. Citeseer, 2005.

[49] Mauro Vocale Luigi Fugaro. In *hands-on cloud-native microservices with jakarta ee*, January 31, 2019.

[50] Tariq N Khasawneh, Mahmoud H AL-Sahlee, and Ali A Safia. Sql, newsql, and nosql databases : A comparative survey. In *2020 11th*

International Conference on Information and Communication Systems (ICICS), pages 013–021. IEEE, 2020.

[51] Roman Čerešňák and Michal Kvet. Comparison of query performance in relational a non-relation databases. volume 40, pages 170–177. Elsevier, 2019.

[52] Deka Ganesh Chandra. Base analysis of nosql database. volume 52, pages 13–21, 2015. Special Section : Cloud Computing : Security, Privacy and Practice.

[53] Chris Richardson. Microservices patterns : With examples in java. October 27, 2018.

[54] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, and Sebastian Meixner. Supporting architectural decision making on data management in microservice architectures. In *European Conference on Software Architecture*, pages 20–36. Springer, 2019.

[55] S Suresh Kumar and PM Mallikarjuna Shastry. Database-per-service for e-learning system with micro-service architecture. In *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pages 705–708. IEEE, 2017.

[56] Aaron P Matthews. Microservice pattern identification from recovered architectures of orchestrated systems. University of California, Irvine, 2021.

[57] Ronnie Mitra and Irakli Nadareishvili. Dealing with the data. In *Microservices : Up and Running*, pages 085–095. O'Reilly Media, 2020.

[58] Binildas Christudas. Cqrs command query responsibility segregation. In *Practical Microservices Architectural Patterns : Event-Based Java*

Microservices with Spring Boot and Spring Cloud. Apress, 2019.

[59] Chaitanya K Rudrabhatla. Comparison of event choreography and orchestration techniques in microservice architecture. volume 9, pages 18–22, 2018.

[60] J. P. Macker and I. Taylor. Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures. 2017.

[61] Wasson M. "design patterns for microservices,". 2017, Accessed : 05-May-2021.[Online]. Available :.

[62] Woolf B. Brown K. "implementation patterns for microservice architectures,". In *Conference on Pattern Languages of Programs*, pages 099–115, 2016.

[63] N. De Greef D. Delhumeau D. . Dillenberger H. Potter A. L. Kooijmans, E. Ramos and N. Williams. Transaction processing : Past, present, and future, ibm redguide publication, isbn-13 : 9780738450780. September 2012.

[64] Event-driven data management for microservices.

[65] M. Richards. Software architecture patterns, o'reilly media. 2015.

[66] K. M. Chandy. Event-driven applications : Costs, benefits and design approaches. pages 25–34, 2018.

[67] Konrad Gos and Wojciech Zabierowski. The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 150–153. IEEE, 2020.

[68] Neha Singhal, Usha Sakthivel, and Pethuru Raj. Selection mechanism of micro-services orchestration vs. choreography. volume 10, page 25, 2019.

[69] Mohamed El Kholy and Ahmed El Fatatry. Framework for interaction between databases and microservice architecture. volume 21, pages 57–63. IEEE, 2019.

[70] M. Cardarelli G. Granchelli and P. D. Francesco. "microart : A software architecture recovery tool for maintaining microservice-based systems". 2017.

[71] C. Fetzer. "building critical applications using microservices,". page pp. 86–89, 2016 nov.

[72] J. Bell R. Geambasu N. Viennot, M. Lecuyer and J. Nieh. "synapse : A microservices architecture for heterogeneous-database web applications". 2015.

[73] Joe Stubbs, Walter Moreira, and Rion Dooley. Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39. IEEE, 2015.

[74] Hong Zhu and Ian Bayley. If docker is the answer, what is the question ? In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 152–163. IEEE, 2018.

[75] 13 :37 h Letzter Zugriffam, 25.10.2021. Vmvare inc. documentation – rabbitmq.

[76] Lizda Iswari et al. Penerapan react js pada pengembangan frontend aplikasi startup ubaform. volume 2, 2021.

[77] C. VIGOUROUX. Apprendre ‘a d´evelopper avec javascript, editions-eni. Avril 2014.

[78] Travis E Oliphant. Python for scientific computing. volume 9, pages 10–20. IEEE, 2007.

[79] Michael Bayer et al. Sqlalchemy.

[80] Nitin Kumar Karma. Web services - the python flask way : Build restful apis using python and flask-restful.

[81] Kunal Relan. Building rest apis with flask : Create python web services with mysql.

[82] Johannes Thones. Microservices. Elsevier, 2015.

[83] J. Lewis M. Fowler. Microservices. Elsevier, Nov. 1, 2018.

[84] lien. Elsevier.

[85] Ramaswamy Chandramouli. Microservices-based application systems. volume 800, pages 800–204, 2019.

[86] M. Mazzara F. Montesi C. Guidi, I. Lanese. Microservices : A language-based approach. page 217–225. IEEE, Springer, Cham, 2017.

[87] Mars,2020.

[88] D . Moshkovich Y . Nardi H. Ship P. Bak, R . Melamed and A . Yaeli. "location and context-based microservices for mobile and internet of things workloads,". pages 1–8, 2015.

[89] Chan Yeob Yeun Mahmoud AI-Qutayri Yousof AI-Hammadi Tasneem Salah, M. Jamal Zemerly. The evolution of distributed systems towards microservices architecture.

[90]

Résumé

Au cours des dernières années, les architectures microservices sont devenues une partie essentielle dans le développement d'applications. Comme ces derniers offrent une approche légère, indépendante, réutilisable et rapide du déploiement des services qui réduit les risques liés à l'infrastructure.

Ces avantages soulèvent un problème concernant le partage de la base de données entre les microservices constituant l'application.

Dans le cadre de ce travail, nous avons développé une application basée sur l'architecture de microservices.

Les tests de performances ont montré la scalabilté de notre application, Malgré l'augmentation du nombre des requêtes, la latence du système tend à se stabiliser, contrairement aux architecteurs monolithiques.

|**Mots clés :** microservices, BDDS, Patterns.

Abstract

In recent years, microservice architectures have become an essential part of application development. As the latter offer a lightweight, independent, reusable and fast approach to service deployment that reduces infrastructure risks.

These advantages raise a problem regarding the sharing of the database between the microservices constituting the application.

As part of this work, we developed an application based on microservice architecture.

The performance tests showed the scalability of our application, Despite the increase in the number of requests, the latency of the system tends to stabilize, unlike monolithic architectors.

Keywords : microservices, DB, Patterns.

الملخص

في السنوات الأخيرة، أصبحت بنى الخدمة الدقيقة جزءًا أساسيًا من تطوير التطبيقات. نظرًا لأن الأخير يوفر نهجًا خفيفًا ومستقلًا وقابل لإعادة الاستخدام وسريعًا لنشر الخدمات مما يقلل من مخاطر البنية التحتية.

تثير هذه المزايا مشكلة فيما يتعلق بتقاسم قاعدة البيانات بين الخدمات الصغيرة التي تشكل التطبيق.

كجزء من هذا العمل، قمنا بتطوير تطبيق يعتمد على بنية الخدمة الدقيقة.

أظهرت اختبارات الأداء قابلية التوسع في تطبيقنا، على الرغم من الزيادة في عدد الطلبات، فإن زمن انتقال النظام يميل إلى الاستقرار، على عكس المعماريين المتجانسين.

الكلمات المفتاحية: الخدمات الدقيقة، الأنماط، قواعد البيانات.