Web services: Présentation de leurs principaux types d'architecture et d'API et analyse comparée de leurs performances respectives par le prototype Stock&Co

Travail de Master

Kesigan Thavarajasingam

Mars 2023

Supervisé par:

Prof. Dr. Jacques Pasquier-Rocha
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Remerciements

Je tiens à remercier le professeur Pasquier pour la confiance qu'il m'a accordé, ainsi que pour la qualité de ses conseils et de son suivi. Dans ce travail, j'ai pu découvrir de nouvelles perspectives et approfondir mes connaissances, et je suis reconnaissant d'avoir eu l'opportunité de partager cette expérience avec le professeur Pasquier.

Abstract

Cette thèse de master étudie les architectures web, les workflows et les API pour déterminer ceux qui facilitent l'évolution d'une entreprise. Trois architectures web ont été comparées: monolithique, SOA et en microservices. La comparaison montre que l'architecture monolithique ne convient pas aux entreprises en expansion. L'architecture SOA est de moins en moins utilisée et l'architecture en microservices devient la norme. Pour faciliter la communication entre les différents services, deux workflows peuvent être mis en place: les workflows en orchestration et les workflows chorégraphiques. Le premier permet de visualiser la progression des tâches. Le deuxième fonctionne sur un système d'événements. La mise en place d'un workflow peut être complexe si de nombreux services sont impliqués. La spécification AsyncAPI peut faciliter cette tâche grâce à sa documentation et à sa capacité à générer du code. Trois technologies permettant le développement des API ont également été étudiées: REST, GraphQL et gPRC. La comparaison des trois à travers la littérature a montré que GraphQL convient pour de grandes quantités de données, gRPC est plus adapté pour les données lègères et les transmissions rapides, et enfin REST est recommandé pour la plupart des situations. Dans la dernière partie de ce travail, un prototype utilisant l'architecture en microservices, le workflow en orchestration et l'API GraphQL a été développé pour mettre en application les différentes notions apprises. L'application a été adaptée pour supporter une API REST, ce qui a permis d'effectuer des tests de performance pour comparer REST et GraphQL. Trois types de test ont été mis en place: test séquentiel, test de charge et test concurrent. Les résultats ne montrent pas de différence notable. GraphQL excelle pour de grandes quantités de données et les données imbriquées. Dans ce domaine, la différence est de 6% en faveur de GraphQL. Finalement, ces tests ont confirmé que REST convient dans la plupart des situations.

Keywords: Architecture web, Workflow, API, REST, GraphQL, gRPC.

Table des matières

1 Introduction **2**

2 Application web **4**

2.1 Qu'est ce qu'est une application web ? 4
2.2 Comment une application web fonctionne ? 4
2.3 Histoire 5
2.4 Avantages des applications web 6
2.5 Les défis 6
2.6 Eléments à considérer dans la création d'une application web 7

3 Services et microservices **8**

3.1 Service web 8
3.2 Principaux types d'architecture web 9
3.2.1 Architecture monolithique 9
3.2.2 Architecture SOA 9
3.2.3 Architectures en microservices 10
3.3 Comparaison 11
3.4 Quelle est l'architecture la plus adaptée pour mon entreprise ? 16
3.5 Communication entre les services 16
3.6 Workflow 16
3.6.1 Workflow engine 17
3.6.2 Workflow en orchestration 17
3.6.3 Worflow chorégraphique 18
3.6.4 Complexités liées aux workflows 19
3.6.5 Comparaison des workflows dans le cadre d'une entreprise 21
3.7 Synthèse 22

4 REST, GraphQL et gRPC **23**

4.1 Définition d'une API 23
4.2 REST 24

4.3 Les fondements de REST 24
4.3.1 Endpoints 25
4.3.2 Interface http 25
4.3.3 Contraintes de développement 26
4.3.4 Hypermedia et HATEOAS 26
4.3.5 Les codes des états pour REST 27
4.4 GraphQL 28
4.4.1 Problématique de REST et fondement de GraphQL 28
4.4.2 Notion de Graphe 28
4.4.3 Principe de conception de GraphQL 29
4.4.4 Caratéristiques de GraphQL 29
4.5 gRPC 34
4.5.1 Protocole RPC 34
4.5.2 Principe de conception de gRPC 36
4.5.3 Type de communication client-serveur 36
4.5.4 gRPC et HTTP/2 37
4.6 Synthèse 38

5 Analyse comparative des API 39
5.1 Cas d'utilisation 40
5.2 Exposition des API 41
5.3 Comparaison des API selon les critères 3 à 10 46
5.4 Analyse de la performance: temps de réponse des API 49

6 Présentation du prototype 52
6.1 Architecture du serveur 52
6.1.1 Gateway 53
6.1.2 Workflow 55
6.1.3 Stock&Co et le processus de paiement 57
6.2 Le client 59

7 Evaluation pratique des performances 64
7.1 Introduction 64
7.2 Méthodologie 65
7.2.1 Types de test 65
7.2.2 Echantillons 66
7.2.3 Récapitulatif 66
7.2.4 Matériel 66
7.3 Test séquentiel 67
7.4 Test de charge 69
7.4.1 Autocannon 69

7.4.2 JMeter 71
7.5 Test concurrent 74
7.6 Synthèse 78

8 Conclusion **79**

9 Annexe **81**
9.1 Informations 81
9.2 Détails techniques 81
9.2.1 Les services 81
9.2.2 Script de lancement 82
9.2.3 Données fictives 82
9.3 Téléchargement et installation du project Stock&Co 83
9.4 Architecture des services 83
9.5 Liste des Queries et Mutations 85
9.6 Benchmark 86
9.6.1 Les échantillons 86
9.6.2 Les endpoints 87
9.6.3 Benchmark JMeter 87
9.7 Graphiques et analyses 88

Bibliographie **90**

Sites Web **92**

Liste des figures

2.1 Communication entre des clients et un serveur à travers internet - [83] . . 5

3.1 Architecture d'une application monolithique - inspiré de [51] 9
3.2 Entreprise A - Découpage du processus en service: manque de transparence au sein de l'entreprise [8] 10
3.3 Entreprise B - Découpage du processus en service: transparence et partage au sein de l'entreprise (SOA) - [8] 10
3.4 Architecture d'une application composée de microservices - inspiré de [51] 11
3.5 Nombre de requêtes par seconde en fonction du nombre d'utilisateurs - Load testing - [1] 15
3.6 Temps de réponse en fonction du nombre d'utilisateurs - Load testing - [1] 15
3.7 Nombre de requêtes par seconde en fonction du nombre d'utilisateurs - Concurrency testing - [1] 15
3.8 Temps de réponse en fonction du nombre d'utilisateurs - Concurrency testing - [1] 15
3.9 processus de traitement des commandes - [94] 18
3.10 Fonctionnement d'un workflow chorégraphique avec RabbitMQ - [71] . . 19
3.11 Documentation AsyncApi - [90] 20
3.12 Commande pour générer un microservice Spring cloud stream à partir de la documentation AsyncAPI - [90] 20
3.13 Structure du projet créé par le générateur de code AsyncAPI - [90] . . . 21
3.14 Contenu du fichier Application.java - [90] 21

4.1 Structure d'un endpoint 25
4.2 Exemple de réponse sans HATEOAS 27
4.3 Exemple de réponse HATEOAS 27
4.4 Graphe de Stock&Co 29
4.5 Schéma Order et User 30
4.6 Schéma d'une query GraphQL 30
4.7 Query GraphQL 31

4.8 Réponse de la figure 4.7 31
4.9 Obtention des informations sur les commandes de l'utilisateur 31
4.10 Réponse de la figure 4.9 32
4.11 Schéma d'une Mutation GraphQL 32
4.12 Exemple Mutation 33
4.13 Exemple Mutation non autorisée 33
4.14 Schéma d'une Subscription GraphQL 34
4.15 Exemple subscription 34
4.16 HTTP REST - Action caché derrière la ressource 35
4.17 RPC - Données cachées derrière les actions 35
4.18 Exemple de fichier .proto 35
4.19 Fichier .proto compilé vers d'autres langages - [74] 36
4.20 Types de communication client-serveur avec gRPC - [5] 37

5.1 Documentation swagger - [95] 42
5.2 Documentation GraphQL - [61] 43
5.3 Query permettant l'introspection 43
5.4 Réponse de la Query 5.3 44
5.5 Exposition des fonctions entre un client et un serveur GPRC - [33] 45
5.6 Reverse Proxy gRPC - [48] 45
5.7 Temps de réponse de REST et GraphQL [12] 50
5.8 Temps de réponse moyen entre REST et GraphQL - [28] 50
5.9 Temps de réponse moyen entre REST et gRPC sans chiffrement (HTTP) et avec chiffrement (HTTPS) - [3] 50
5.10 Temps de réponse pour cent éléments [13] 51

6.1 Architecture de Stock&Co 53
6.2 Définitions des endpoints dans Mesh 53
6.3 Extension du schéma utilisateur 54
6.4 Liaison entre le microservice utilisateur et le microservice commande . . 54
6.5 Query combiné avec Mesh 54
6.6 Réponse de la figure 6.5 55
6.7 L'interface de camunda modeler 56
6.8 Paramètres pour la tâche "Mise à jour des points" 56
6.9 Topic update_point 57
6.10 Interface de Camunda 57
6.11 Workflow paiement 58
6.12 Interface du client 60
6.13 Page d'un produit 60
6.14 Détails du panier 61
6.15 Page de paiement 61

6.16 Page de profil 62
6.17 Commande en attente 62
6.18 Mis à jour du statut de la commande 63
6.19 Historique du workflow 63

7.1 Temps de réponse en fonction du nombre de requêtes - addProduct (POST) 68
7.2 Temps de réponse en fonction du nombre de requêtes - getUsers (GET) . 68
7.3 Temps de réponse en fonction du nombre de requêtes - products-users (GET) 68
7.4 Répartition des requêtes réussies et échouées en fonction du nombre d'utilisateurs concurrents sur l'échantillon few 70
7.5 Temps de réponse moyen et écart type 70
7.6 Paramétrage de JMeter pour le test de charge 71
7.7 Temps de réponse en fonction du temps - test de charge 73
7.8 Distribution du temps de réponse en fonction du nombre d'utilisateur concurrent - test de charge 73
7.9 Paramétrage de JMeter pour le test concurrent 74
7.10 Temps de réponse en fonction du temps - test concurrent 76
7.11 Temps de réponse en fonction du temps et par la fonction - test concurrent 76
7.12 Temps de réponse en fonction du nombre d'utilisateurs concurrents - test concurrent 77
7.13 Temps de réponse en fonction du nombre d'utilisateurs concurrents et par fonction - test concurrent 77

9.1 Architecture du client 83
9.2 Architecture du microservice utilisateur 84
9.3 Architecture du microservice produit 84
9.4 Architecture du microservice commande 84
9.5 Architecture du microservice benchmark (sequentiel + concurrent/autocannon) 84
9.6 Activer/désactiver un benchmark 88
9.7 Chemin d'accès JMeter 88

1 Introduction

Le modèle économique global actuel est influencé par la digitalisation. La majorité des activités humaines sont touchées par ce changement, que ce soit le travail, les relations sociales ou l'éducation [86]. En effet, au travers des applications web, les individus peuvent communiquer et collaborer à distance, indépendamment de leur emplacement géographique [26]. Cette digitalisation a également modifié le fonctionnement des entreprises. D'une part, elles ont simplifié et accéléré leurs processus, et d'autre part, cela a permis de stimuler l'innovation et de créer de nouveaux marchés et nouvelles industries [77]. De nombreuses entreprises sont forcées de se digitaliser pour répondre aux nouveaux modes de consommation [4]. Les applications web sont devenues le point de contact entre les entreprises et les clients [98]. L'adoption et l'utilisation des applications web apportent de nouveaux défis. Leurs développements doivent correspondre aux caractéristiques de l'entreprise et à son évolution. En effet, l'architecture des applications web est devenue un enjeu majeur pour les sociétés. Une application web bien conçue permettra une bonne qualité et performance de celle-ci. Ainsi, l'entreprise réduira ses coûts et la maintenance sera effective [80].

Ce travail de master étudie la partie serveur des applications web. Les différentes structures et architectures des serveurs et des API sont analysées. Pour comprendre l'importance de l'architecture d'une application web, un cas concert servira de mise en situation:

L'entreprise Stock&Co est une entreprise de vente en ligne. Depuis l'année 2020, les ventes de l'entreprise ont quadruplé. La structure informatique a de plus en plus de mal à suivre la demande et c'est pourquoi une nouvelle structure doit être envisagée. Son chiffre d'affaire annuel ayant également fortement augmenté, le directeur a engagé plusieurs développeurs et architectes en informatique pour développer une application web solide et évolutive. Le directeur a quelques notions en informatique. Il souhaiterait être conseillé sur l'architecture à adopter et, si possible, que les processus (commandes, paiements, etc,...) puissent être suivis.

Cette problématique peut être reformulée de façon plus générale: **"Quelles sont les architectures et API à privilégier par les entreprises pour s'adapter à leur croissance et à l'évolution technologique ?"**

Cette thèse est divisée en plusieurs chapitres. Le premier chapitre introduit le travail de master. Le deuxième chapitre présente l'application web et ses challenges. Le troisième chapitre s'intéresse à la structure du serveur. La notion de service est présentée et les différentes façons existantes pour structurer le serveur sont étudiées. Les différents

outils facilitant l'interaction entre services indépendants sont explorés. Le quatrième chapitre s'intéresse aux technologies facilitant le développement des API. REST, GraphQL et gRPC sont présentés. Le cinquième chapitre compare ces technologies sur plusieurs points. Le sixième chapitre est consacré au prototype Stock&Co. Le prototype et les diverses technologies utilisées pour son développement sont commentées. Dans le septième chapitre, des tests de performance sont effectués sur le prototype. La méthodologie choisie pour appliquer les tests de performance est présentée. Les résultats sont analysés. Finalement, le dernier chapitre conclue le travail.

2
Application web

2.1 Qu'est ce qu'est une application web ? **4**
2.2 Comment une application web fonctionne ? **4**
2.3 Histoire **5**
2.4 Avantages des applications web **6**
2.5 Les défis **6**
2.6 Eléments à considérer dans la création d'une application web **7**

Ce chapitre explique la définition d'une application web et comment elle fonctionne. Nous y parlerons également de son histoire et de son évolution au fil des années.

2.1 Qu'est ce qu'est une application web ?

Une application web est une application stockée sur un serveur distant et rendue disponible à l'utilisateur à travers l'interface du navigateur web. Les applications web répondent à divers besoins. Les plus connus sont les sites e-commerce, portails web ou les applications type messageries [66].

Les applications web ne sont pas de simples sites. Il ne s'agit pas seulement d'afficher des données stockées sur un serveur, mais bien d'interagir avec un utilisateur [64]. Plusieurs architectures sont disponibles pour structurer ces applications. La plus utilisée est la structure client-serveur [27]. Cette thèse sera consacrée à cette structure.

2.2 Comment une application web fonctionne ?

Deux composants sont essentiels pour le fonctionnement d'une application web: un client et un serveur. Le client communique avec le serveur à travers le réseau internet. Il effectue des requêtes auprès du serveur et reçoit des réponses de celui-ci. Il adapte ensuite son contenu et son interaction avec l'utilisateur [76]. La figure 2.1 illustre la communication entre différents clients et un serveur via internet.

La communication client-serveur est différente selon les besoins et les services proposés. Par exemple, une application de streaming et une application web de vente en ligne n'utiliseront pas les mêmes types de communication client-serveur. Le premier nécessite

un transfert de données client-serveur en continu tandis que le deuxième déclenche une communication grâce aux actions de l'utilisateur. Les technologies utilisées seront alors différentes. Nous en parlerons plus en détail dans le chapitre 4.

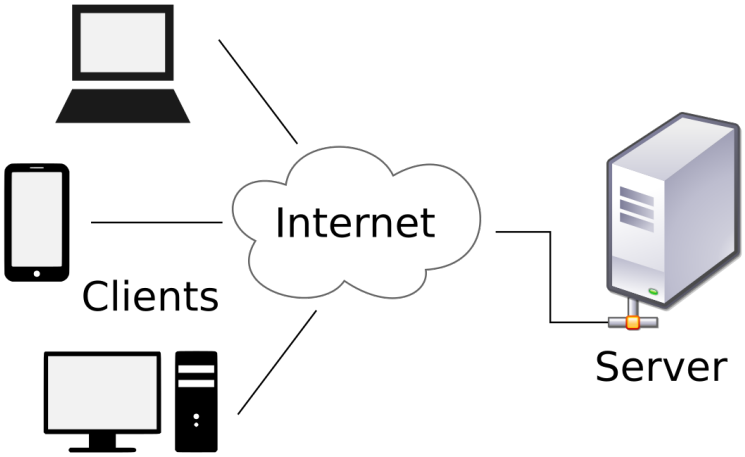

FIGURE 2.1 – Communication entre des clients et un serveur à travers internet - [83]

2.3 Histoire

Pour comprendre l'évolution des applications web, il est nécessaire de commencer depuis l'apparition du web. Le web, appelé également "World Wide Web" a été développé par Tim Berners-Lee en 1991. Son principal but était de permettre la publication et le partage de documents scientifiques [15]. Le 30 avril 1993, le CERN rend le world wide web disponible au public [49]. Il n'existait, à cette époque, pas encore d'outils ou de langages de programmation offrant une vraie interaction utilisateur. Les premiers sites web et applications web étaient des pages statiques. L'utilisateur accédait aux sites au travers d'un navigateur. Le serveur récupérait la demande d'accès et retournait un résultat, sous forme de page, qui s'affichait sur l'écran de l'utilisateur. L'interaction utilisateur était limitée. Chaque interaction ou changement demandait au serveur d'envoyer une nouvelle page web complète [63].

À partir de 1995, de nouvelles possibilités d'interaction apparaissent grâce au lancement de JavaScript développé par la société Netscape communication Corp. L'entreprise est connu pour avoir développé l'un des premiers navigateurs web. Le langage permettait au développeur d'ajouter des éléments dynamiques côté client. L'expérience était plus rapide et plus agréable pour l'utilisateur même si le chargement complet devait toujours être effectué par le serveur [63].

Au fil des années, le client devenait de plus en plus indépendant. Le serveur recevait de moins en moins de requêtes et l'interaction utilisateur est devenu plus rapide et plus agréable. À partir de 2005, les applications web sont divisées en plusieurs couches. L'interface est séparée de la couche des données et les pages n'ont plus besoin d'être entièrement chargées depuis le serveur. De nouvelles technologies sont apparues depuis la création du web, par exemple HTML5, les API, les requêtes asynchrones et bien plus encore. Elles

facilitent grandement l'interaction et le développement [63]. Dans la prochaine section, quelques-uns de ses avantages sont présentés.

2.4 Avantages des applications web

Les applications web sont de plus en plus populaires. Elles offrent plusieurs avantages non négligeables [66][70]:

- **Accès à l'application par plusieurs personnes en même temps**. Des milliers d'utilisateurs peuvent se connecter en même temps à un site web. Leurs actions et interactions sont isolées et n'impactent pas celles des autres.
- **Aucune installation nécessaire**. L'accès à l'application se fait à travers internet. Cela ne nécessite donc aucune installation. La maintenance de l'application est gérée par l'entreprise qui propose le service. Cela permet également à l'application d'être compatible avec plusieurs appareils différents.
- **Actualisées en continu.** Les mises à jour et maintenances se font de manière centralisées et l'utilisateur n'a pas besoin de s'en préoccuper.
- **Flexibles et évolutives.** Les applications web sont simples à développer et à intégrer dans des systèmes. Contrairement à une application traditionnelle, il est simple de faire évoluer les applications web en fonction des besoins de l'entreprise.

2.5 Les défis

Les applications web offrent de nombreux avantages mais rencontrent des défis lors de leur conception [15].

- **Disponibilité en tout temps**. Une application web doit être accessible en tout temps et depuis n'importe quel appareil. Il faut que l'application adapte son contenu en fonction du support.
- **L'apparence.** Le développement d'une application web demande un travail important sur les aspects tels que la mise en page, la cohérence, l'accessibilité, la navigation, le contenu ou les outils de recherche. Si le visuel n'est pas attractif, que l'utilisation est très complexe, que les informations ne sont pas claires ou que l'ergonomie est mal développée, l'utilisateur risque de ne plus revenir. Un visuel attrayant permet de gagner la confiance de l'utilisateur et de le fidéliser.
- **Ouverture au monde.** Les applications web ne sont plus liées à une région ou à un pays. Elles doivent être disponibles pour n'importe quel utilisateur dans le monde. Le contenu doit pouvoir s'adapter au pays, mais également à la culture, aux règles en vigueur dans le pays et aux conditions technologiques.
- **Sécurité et évolutivité.** Les applications web doivent garantir une grande sécurité. Par exemple, les applications de vente en ligne collectent des informations sensibles telles que les numéros des cartes bancaires. Ces informations doivent être protégées en tout temps et ne jamais être accessibles aux personnes non autorisées. La sécurité est également un défi lorsque l'application évolue. Le nombre de personnes qui accéderont aux sites ne peut pas être défini à l'avance. Par exemple, les

sites de ventes en ligne connaissent des gros flux d'utilisateurs en période de soldes. L'application web doit être capable de gérer le trafic et en même temps d'assurer la sécurité des données.

2.6 Eléments à considérer dans la création d'une application web

La section précédente a défini les défis liés aux applications web. Néanmoins, ces défis ne se limitent pas qu'aux applications mais s'étendent également aux entreprises. Plusieurs questions sont à se poser: quels sont mes besoins ? Quels sont les services que l'entreprise va proposer ? Est-ce qu'un service est proposé temporairement ou de manière récurrente ? Est-ce que ce service évoluera dans le temps ? Trois aspects peuvent être pris en compte pour répondre à ces questions [44][37][93]:

- **La rapidité.** Quelle est la durée disponible pour le développement ? Quels sont les délais et le niveau d'urgence ? Le temps est un facteur important. Ainsi, l'entreprise doit réfléchir à ses besoins à court terme et à long terme. Pour une entreprise de vente en ligne, précipiter le développement peut être néfaste et causer la faillite de l'entreprise. En effet, une application instable ne permettra pas de gagner la confiance de l'utilisateur et pourrait mettre un terme à l'activité de l'entreprise.
- **La fiabilité.** Une application web doit être disponible en tout temps. Il est donc important d'avoir les bons outils et de continuellement vérifier les failles et les erreurs. Par exemple, le système de paiement doit toujours être opérationnel et sécurisé.
- **L'évolutivité.** Une application web évolue au cours du temps. De nouvelles technologies apparaissent tous les jours. L'entreprise connaît une croissance à la fois en termes d'effectif et d'expansion de son marché. Une structure solide doit permettre une évolution qui s'adapte aux composants actuels sans les impacter négativement. L'ajout d'un nouveau service ne doit pas avoir des répercussions sur le bon fonctionnement des éléments déjà mis en place.

Ces trois aspects sont importants dans le choix de l'infrastructure, des langages de programmation et des librairies utilisées pour la structure de l'application web. Les prochains chapitres traiteront des différents composants d'une architecture serveur et seront analysés en prenant en compte ces trois aspects.

3
Services et microservices

3.1 Service web **8**
3.2 Principaux types d'architecture web **9**
3.2.1 Architecture monolithique 9
3.2.2 Architecture SOA 9
3.2.3 Architectures en microservices 10
3.3 Comparaison **11**
3.4 Quelle est l'architecture la plus adaptée pour mon entreprise ? **16**
3.5 Communication entre les services **16**
3.6 Workflow **16**
3.6.1 Workflow engine 17
3.6.2 Workflow en orchestration 17
3.6.3 Worflow chorégraphique 18
3.6.4 Complexités liées aux workflows 19
3.6.5 Comparaison des workflows dans le cadre d'une entreprise . . 21
3.7 Synthèse **22**

Ce chapitre présente les différents types d'architecture existants pour structurer un backend.

3.1 Service web

Un service web est une application fonctionnant sur un réseau et capable d'interagir avec d'autres applications à travers un protocole internet. Ces services possèdent une interface qui sert de point de communication. Un service propose des fonctionnalités qui sont déclenchées par un événement ou invoquées directement [6].

En général, une application web est constituée de plusieurs services. Dans le cas d'une application de vente en ligne, par exemple, le serveur peut être constitué d'un service pour les paiements, un pour gérer les produits et un pour l'authentification. Il est important de structurer ces services en suivant une architecture. Trois types d'architecture sont couramment utilisés: l'architecture monolithique, les architectures en microservices et les architectures orientées service (SOA) [22]. Ils ont chacun leurs avantages et ont tous été conçus pour répondre à différents besoins organisationnels [73].

3.2 Principaux types d'architecture web

3.2.1 Architecture monolithique

Une architecture monolithique est une architecture regroupant toutes les fonctionnalités d'une application dans un seul service. Cela signifie que le service est constitué d'un seul bloc de code et que les fonctionnalités sont souvent étroitement liées les unes aux autres, comme représenté par la figure 3.1. Ce style architectural est simple à développer et à déployer [23]. Il peut être adapté pour des applications simples, pour des petites entreprises avec peu de développeurs ou pour des entreprises ayant une logique métier très faible (par exemple, peu de services fournis ou peu d'acteurs sur la chaîne) [53].

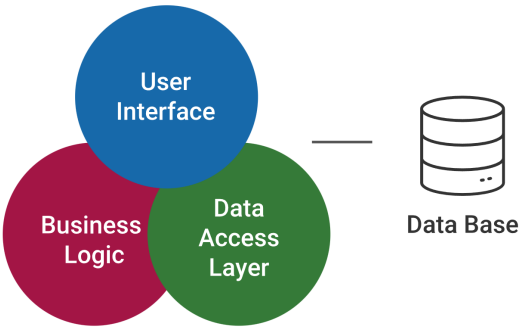

FIGURE 3.1 – Architecture d'une application monolithique - inspiré de [51]

L'architecture monolithique était auparavant un standard lors du développement d'une application web. Néanmoins, cette architecture est devenue obsolète avec l'évolution des modèles économiques des entreprises. En effet, certaines sociétés proposent des services à leurs clients sous forme de pay-as-you-go. Les clients sont facturés en fonction de leur utilisation. L'architecture monolithique ne permet pas de distinguer précisément les services utilisés et la mise en place du pay-as-you-go complexifie le développement de l'application. D'autres versions de l'architecture monolithique ont également vu le jour (monolithique modulaire, monolithique distribué) mais ces architectures ne permettent pas de clairement séparer les services. Le couplage fort entre les services est toujours présent. Les architectures SOA ou en microservices permettent de résoudre ce problème [17].

3.2.2 Architecture SOA

Une architecture SOA, appelée architecture orientée service, est constituée de plusieurs services fonctionnant ensemble afin de fournir un processus métier complet. Chaque service est indépendant l'un de l'autre. Les services communiquent via le réseau et s'échangent des informations. Ils partagent, en général, la même base de données. Le but d'une SOA est de créer des services indépendants afin qu'ils puissent être réutilisés dans plusieurs processus métiers différents au sein de l'entreprise. Ainsi, une architecture SOA améliore l'efficience et le fonctionnement de l'organisation [18]. En effet, les services sont partagés au sein de l'entreprise grâce à un système d'inventaire [79]. Cette approche contribue à réduire les coûts associés au développement des projets informatiques et à accélérer leur réalisation [14].

La figure 3.2 représente une entreprise qui n'utilise pas une architecture SOA. Chaque processus A, B et C, représenté sur la figure 3.2, a créé ses propres et nouveaux services. Cependant, cette approche n'est pas optimale car elle peut induire une duplication du travail. En effet, certains services similaires peuvent être utilisés indépendamment dans les différents processus. Par exemple, un système de paiement possède généralement le même fonctionnement. Dans certains cas, une entreprise peut développer plusieurs fois ce service au lieu de le réutiliser. Grâce à l'architecture SOA, illustrée par la figure 3.3, l'entreprise garde un inventaire des services. Elle partage ces différents services au sein de l'entreprise et permet d'éviter la redondance [8][18].

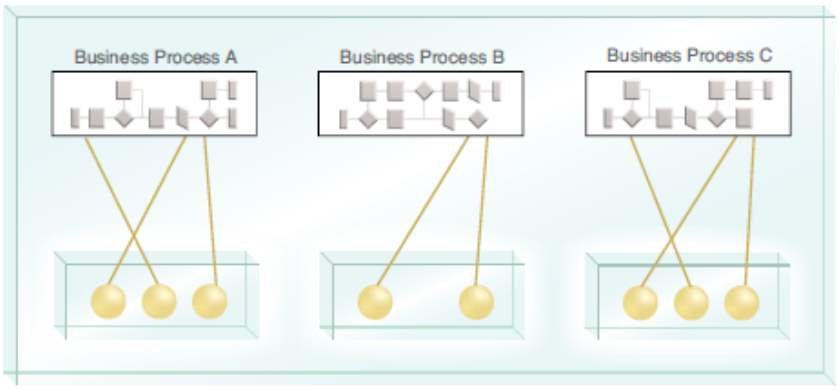

FIGURE 3.2 – Entreprise A - Découpage du processus en service: manque de transparence au sein de l'entreprise [8]

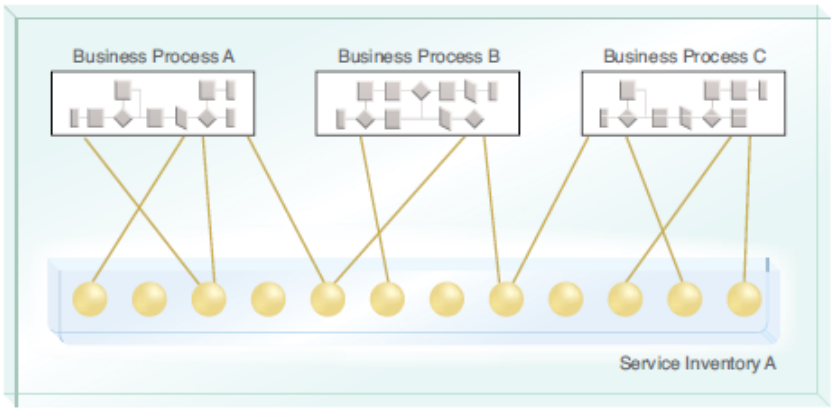

FIGURE 3.3 – Entreprise B - Découpage du processus en service: transparence et partage au sein de l'entreprise (SOA) - [8]

3.2.3 Architectures en microservices

Une architecture en microservices, représentée par la figure 3.4, est composée de plusieurs services indépendants les uns des autres. Chaque service est défini dans le but de fournir une seule et unique fonctionnalité. Par exemple, un microservice peut être utilisé pour gérer les paiements, tandis qu'un autre est utilisé pour authentifier l'utilisateur. Plusieurs microservices sont nécessaires pour composer un processus complet. L'architecture en microservices possède des similitudes avec l'architecture SOA. Cependant, elle se distingue

par son domaine d'application. Contrairement à l'architecture SOA qui organise les services au sein d'une entreprise, l'architecture en microservices est conçue pour structurer les services à l'intérieur d'une application. Cette architecture est généralement utilisée pour des applications complexes et dans des grandes entreprises car elle rend le développement des services plus agiles. En effet, l'indépendance entre les microservices permet de mieux répartir le développement de l'application entre les différentes équipes [9].

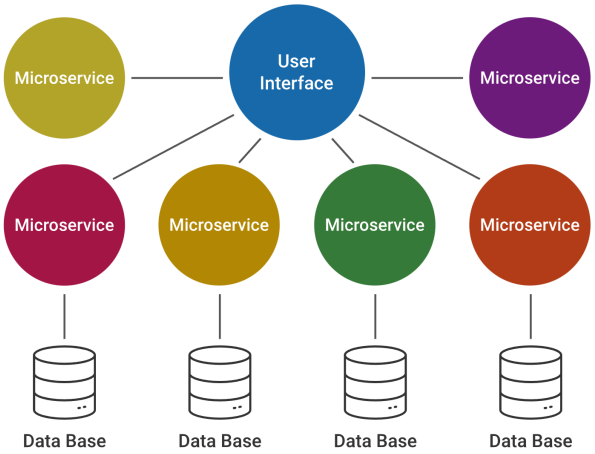

FIGURE 3.4 – Architecture d'une application composée de microservices - inspiré de [51]

3.3 Comparaison

Dans cette section, les trois architectures présentées ci-dessus vont être comparées selon les critères suivants:

- **Réutilisabilité.** La réutilisabilité définit la possibilité de pouvoir utiliser du code, des fonctionnalités ou un service déjà existant, pour les intégrer dans un autre service. Cet aspect est à considérer dans le choix de l'architecture car il permet de réduire le temps de développement des nouvelles applications et leurs coûts [45].
- **Maintenance.** Ce critère cherche à estimer l'effort nécessaire pour modifier un service ou pour étendre ses fonctionnalités. La maintenance peut être évaluée en tenant compte de l'estimation du temps pour faire des changements et l'impact de ces modifications sur les autres fonctionnalités et services [50].
- **Evolutivité.** L'évolutivité est la capacité d'un service ou d'une application à augmenter ses ressources pour répondre à la demande. Cette demande peut se traduire, par exemple, par l'augmentation du nombre d'utilisateurs utilisant le service. Une architecture favorable à l'évolutivité permet de mieux répartir les ressources entre les différents services et ainsi de réduire les coûts [91].
- **Latence du réseau.** La latence du réseau évalue l'impact d'un problème, lié au réseau, sur le fonctionnement des services et sur la communication entre eux.
- **Sécurité.** La sécurité comprend deux aspects principaux: la protection des données et la diminution de la surface d'attaque. La protection des données peut se faire grâce à la mise en place d'un système d'authentification et d'autorisation. La surface d'attaque, qui représente le nombre de systèmes potentiellement vulnérables, peut être réduite grâce à des systèmes de contrôle et une protection active (utilisation

d'un antivirus ou modification régulière des mots de passe par exemple). Ces deux aspects sont essentiels pour garantir une sécurité optimale.

- **Performance.** La performance évalue le temps de réponse et la quantité de requêtes pouvant être traitée par une application utilisant une architecture définie.

Le tableau 3.1 présente les résultats.

TABLE 3.1 – Comparaison des architectures monolithique, SOA et en microservices

Critère	**Architecture**	**Explications**
Réutilisabilité	Monolithique	**Non** En raison de l'interdépendance entre les fonctionnalités et les services, il est très difficile d'isoler du code pour le réutiliser dans une autre application [23].
	SOA	**Oui** Le principe fondamental de l'architecture SOA est la réutilisation des **services** indépendants les uns des autres [78][58]. Cependant, cette architecture n'est pas favorable à la réutilisation du **code** car au sein d'un service, les fonctionnalités peuvent être étroitement liées [7].
	microservices	**Oui** L'architecture en microservices privilégie la réutilisation de bout de **code** plutôt que des **services** complets afin d'éviter les dépendances [79].
Maintenance	Monolithique	**Difficile** L'architecture monolithique est simple à maintenir lorsque l'application possède peu de fonctionnalités. Cependant, la maintenance devient plus complexe à mesure que l'application grandit. Cela est principalement dû au couplage fort entre les fonctionnalités. Ainsi, toute modification ou changement risque d'affecter les autres fonctionnalités qui en dépendent [23].
	SOA	**Facile** L'architecture SOA facilite la maintenance des services car ceux-ci sont indépendants. Chaque service peut être modifié et ses fonctionnalités étendues indépendamment des autres [18].
	microservices	**Facile** L'architecture en microservices est composée de plusieurs microservices indépendants. Cette architecture facilite la maintenance car chaque service peut être modifié ou étendu sans impacter les autres [23].

Evolutivité	Monolithique	**Difficile** L'interdépendance des services complexifie l'évolutivité de l'application. Il n'est notamment pas possible de répartir les ressources (CPU par exemple) en fonction des besoins de chaque service. Par conséquent, cela cause une mauvaise répartition des ressources et engendre des coûts supplémentaires [23].
	SOA	**Facile** Il est simple de faire évoluer l'application car les services sont indépendants. De plus, les ressources nécessaires pour chaque service peuvent être allouées selon les besoins [18].
	microservices	**Facile** Les ressources peuvent être allouées selon les besoins [23].
Impact du réseau	Monolithique	**Fonctionnement des services: Oui** Un problème lié au réseau rendra tous les services indisponibles. **Communication entre les services: Non** Les différents services communiquent entre eux directement dans l'application. Par conséquent, l'architecture monolithique est peu impactée par les problèmes liés au réseau [75].
	SOA	**Oui** La communication entre les différents services se fait via le réseau. Les coupures de réseau et les latences peuvent avoir des effets négatifs sur le fonctionnement des services (perte des données par exemple). Ainsi, cette architecture nécessite la mise en place de fonctionnalité ou l'utilisation d'outils externes pour détecter ce type de problème [97][75].
	microservices	**Oui** Les services communiquent à travers le réseau. Les latences et problèmes liés au réseau peuvent impacter négativement le fonctionnement de l'application. Par conséquent, il est nécessaire de surveiller la communication entre les services afin de détecter d'éventuels problèmes [97][75].

Sécurité	Monolithique	**Modérée** Dans une architecture monolithique, la mise en place d'un système d'authentification et d'autorisation est facile à implémenter. Tous les services étant regroupés dans une même application, un seul système d'authentification et d'autorisation est suffisant. Une application développée sur cette architecture n'a qu'une seule surface d'attaque. Toutefois, cela ne signifie pas que sa protection est assurée car l'interdépendance des fonctionnalités complexifie la sécurité [88].
	SOA	**Difficile** La mise en place d'un système d'authentification et d'autorisation est complexe dans une architecture SOA. Plusieurs interrogations doivent être prises en compte: doit-on utiliser le même jeton d'authentification sur tous les services ? Doit-on mettre en place plusieurs niveaux de sécurité ? Comment passer d'un service à un autre sans perdre l'authentification ? [31][23]. La mise en place d'un système de sécurité est complexe dans une architecture SOA. Cela est le cas pour deux principales raisons: le nombre de services à sécuriser et les communications s'effectuant via le réseau. De ce fait, la surface d'attaque est plus importante [88].
	microservices	**Difficile** L'architecture en microservices possède les mêmes risques de sécurité et doit prendre en compte les mêmes éléments que l'architecture SOA.

Performance

Un comparatif entre l'architecture monolithique et l'architecture en microservices a été effectué par la Budapest University of Technology and Economics [1]. Cette étude a examiné deux paramètres: le temps de réponse et le nombre de requêtes que l'application peut gérer par seconde en fonction du nombre d'utilisateurs. Pour mesurer ses deux éléments, deux types de test sont mis en place. Le premier test consiste à augmenter graduellement le nombre d'utilisateurs sur l'application. L'objectif est d'évaluer comment chaque architecture supporte une augmentation de la charge et de la consommation des ressources. Ce test est connu sous le nom de load testing. Le deuxième test consiste à évaluer la capacité de chaque architecture à gérer une charge élevée lorsque plusieurs services sont sollicités simultanément. Il s'agit d'un test concurrent ou concurrency testing.

Les résultats du load testing sont présentés par les figures 3.5 et 3.6. En ce qui concerne le nombre de requêtes traitées par seconde, illustrées par la figure 3.5, l'architecture monolithique semble avoir un avantage sur l'architecture en microservices lorsque le nombre

d'utilisateurs est inférieur à 1000. Cependant, au-delà de 1000 utilisateurs, la différence en termes de nombre de requêtes traitées par seconde entre les deux architectures n'est plus significative. En terme de temps de réponse, présentés par la figure 3.6, les deux architectures semblent présenter des performances similaires.

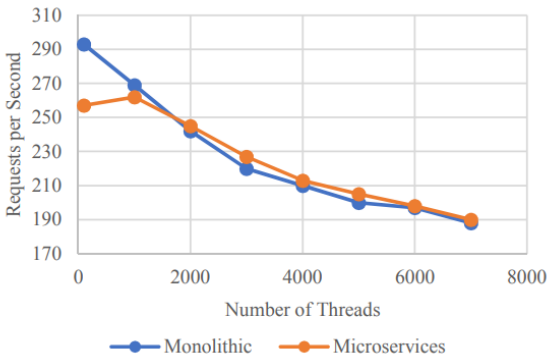

FIGURE 3.5 – Nombre de requêtes par seconde en fonction du nombre d'utilisateurs - Load testing - [1]

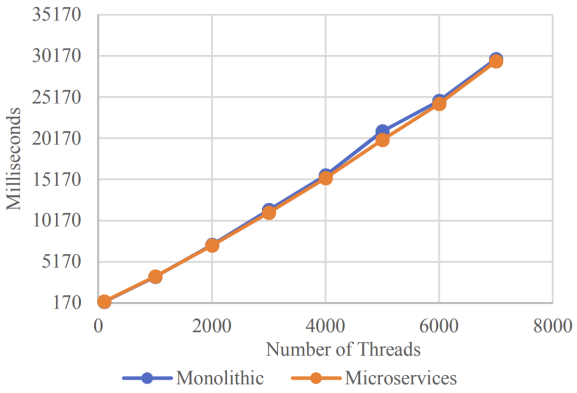

FIGURE 3.6 – Temps de réponse en fonction du nombre d'utilisateurs - Load testing - [1]

Les figures 3.7 et 3.8 présentent les résultats du test concurrent. Concernant le nombre de requêtes traitées par seconde, illustrées par la figure 3.7, l'architecture monolithique a un réel avantage sur l'architecture en microservices. Les auteurs estiment la différence à 6% en faveur de l'architecture monolithique. Quant aux temps de réponse, présentés par la figure 3.8, les deux architectures présentent toujours des performances similaires et aucune différence significative n'a été observée.

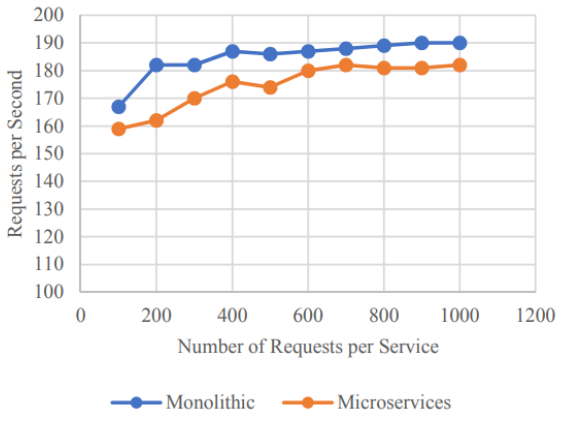

FIGURE 3.7 – Nombre de requêtes par seconde en fonction du nombre d'utilisateurs - Concurrency testing - [1]

FIGURE 3.8 – Temps de réponse en fonction du nombre d'utilisateurs - Concurrency testing - [1]

Aucune étude comparative entre l'architecture SOA et les architectures monolithique ou en microservices n'a été découverte.

3.4 Quelle est l'architecture la plus adaptée pour mon entreprise ?

Le choix de l'architecture pour une entreprise doit se faire en prenant en compte plusieurs facteurs. Les points suivants complètent la section précédente et peuvent aider à la prise de décision [54][100][58] :

- **Taille de l'entreprise**. Le choix de l'architecture approprié dépend de la taille de l'entreprise ainsi que du nombre de produits ou de services proposés. Dans le cas d'une entreprise proposant plusieurs produits, il peut être judicieux de séparer distinctement les différents services en utilisant une architecture SOA ou une architecture en microservices. En revanche, pour les entreprises avec un nombre limité de produits ou avec peu d'employés, une architecture monolithique est plus facile à mettre en place et sera également moins coûteuse.
- **Nature des produits et services vendus.** Si l'entreprise vend une multitude de produits sur divers canaux de distribution (en ligne, magasin, etc), l'adoption d'une architecture en microservices est recommandée. Cette solution offre une meilleure traçabilité des ventes effectuées sur les différents canaux. En revanche, si les ventes se font uniquement sur un canal, par exemple, un canal digital, alors une architecture monolithique est plus adaptée.
- **L'objet commercial global**. Il est primordial de choisir une architecture qui puisse soutenir l'évolution de l'entreprise. Dans le cas d'une entreprise prévoyant une forte croissance, une architecture en microservices est plus appropriée, car elle offre une meilleure évolutivité. Bien que l'architecture SOA soit également envisageable, elle est plus complexe à mettre en place et est de moins en moins utilisée.

3.5 Communication entre les services

Les architectures SOA ou en microservices nécessitent que des informations soient échangées entre les différents services. Cependant, contrairement à l'architecture monolithique, la communication se fait à travers le réseau, ce qui apporte une contrainte supplémentaire. Il existe un risque de créer des dépendances entre les services, ce qui peut entraîner des problèmes de fonctionnement. Par exemple, lorsque un service A appelle un service B, le service A effectue une requête sur l'API du service B. Il attend une réponse puis continue son exécution. Cependant, si la requête échoue pour une quelconque raison liée au service B, le service A est bloqué. Ainsi, plusieurs services peuvent être affectés par cette dépendance [52]. Pour faciliter la communication entre les services tout en maintenant un couplage faible, des outils appelés workflow peut être utilisés.

3.6 Workflow

Dans une entreprise, un workflow désigne l'ensemble des processus métiers qui sont modélisés et gérés. Il est composé d'un ensemble d'étapes pour la création et la livraison d'un produit ou service, et ces étapes sont réparties entre différents acteurs et définies dans un ordre précis [96]. Le mode de fonctionnement est identique pour les services web.

Un outil externe coordonne les services et les fonctions nécessaires pour effectuer une suite de tâches et répondre à un besoin [52]. L'utilisation d'un workflow permet de mieux coordonner les services et de visualiser l'avancée des tâches.

Deux concepts de workflow sont généralement utilisés [52]:

- **Les workflows chorégraphiques.**
- **Les workflows en orchestration.**

Ces deux workflows sont présentés dans la section 3.6.2 et 3.6.3. Le workflow en orchestration fonctionne grâce à un workflow *engine* tandis que le workflow chorégraphique ne l'utilise pas.

3.6.1 Workflow engine

Un workflow *engine* est un logiciel qui s'occupe de gérer un workflow. Il permet de définir l'ordre des tâches, de les automatiser, de diriger les données vers les services appropriés et de visualiser en temps réel l'avancée du processus. Pour qu'un workflow *engine* fonctionne, un processus métier doit être défini au préalable (avec le langage BPMN par exemple) et doit être déployé sur l'outil [24].

3.6.2 Workflow en orchestration

Un workflow en orchestration est géré par un workflow *engine* qui sert de contrôleur. Il s'agit d'un système centralisé. La mise en place se fait en plusieurs étapes. Il faut, premièrement, représenter les différents processus et leur ordre en utilisant un langage spécifique, généralement BPMN. Cette étape s'appelle la définition des processus. Ensuite, cette définition est publiée sur le workflow *engine*. L'outil permet ensuite de visualiser le processus et de le démarrer, soit depuis l'application, soit en utilisant l'API. Une fois lancé, l'application se charge d'exécuter les tâches définies [24].

Un workflow en orchestration fixe l'ordre des activités, se charge de récupérer les données auprès des services, transmet les données aux destinataires et garde une trace des différents états. Les microservices n'ont pas d'interaction directe les uns avec les autres [24].

La figure 3.9 présente le processus de traitement de commande qui a été défini en utilisant le langage BPMN et déployé sur le workflow en orchestration. Le workflow fonctionne de la manière suivante:

1. Lors de la réception de la commande, la disponibilité des produits est vérifiée.
    - Si les produits sont indisponibles, la commande est annulée et le processus se termine.
    - Si les produits sont disponibles, le processus passe à l'étape suivante.
2. L'entreprise prépare la commande.
    - Si la commande est annulée (par le client ou l'entreprise), le processus retourne à l'étape précédente.
3. L'entreprise livre la commande.
    - Si la livraison s'est déroulée avec succès, le processus se termine.

- Si la livraison n'a pas pu être effectuée, une nouvelle livraison est programmée pour le lendemain.
- Si une erreur est détectée, le processus retourne à l'étape précédente.

Les différentes étapes sont représentées par les rectangles. La progression du workflow, allant de la réception de la commande jusqu'à sa clôture, est représentée en vert.

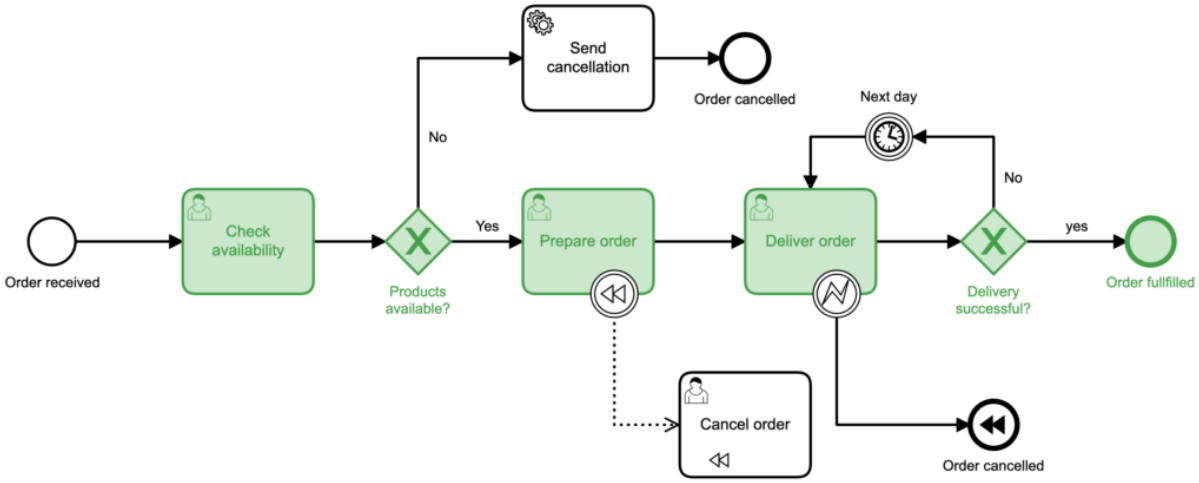

FIGURE 3.9 – processus de traitement des commandes - [94]

3.6.3 Worflow chorégraphique

Un workflow chorégraphique utilise un principe d'événement. Les services transmettent leurs données à une application externe en y associant un événement spécifique. D'autres services peuvent souscrire à ces événements, ce qui leur permet d'être informés lorsque des données sont disponibles. Ils peuvent ensuite récupérer ces données, les traiter, les modifier et les renvoyer à l'application en y associant un événement. Le workflow fonctionne comme un courtier. Les données peuvent y être publiées et récupérées. Un workflow chorégraphique est avantageux car il permet d'éviter que les données se perdent lors de la communication entre les services. Cependant, il ne se charge pas de la redirection des informations vers les services appropriés, ni de la coordination des différents services. Cette logique doit être définie dans les services même. Par conséquent, il s'agit d'un système décentralisé [24].

Le figure 3.10 présente le fonctionnement du workflow chorégraphique RabbitMQ. Un service (PRODUCER) publie des données sur l'application RabbitMQ en associant un événement et un type de distribution (associé à un événement ou distribution à tous les services, par exemple). RabbitMQ va ensuite stocker ces données dans une ou plusieurs files d'attente. Les services souscrits à l'événement sont informés et peuvent récupérer les données [71].

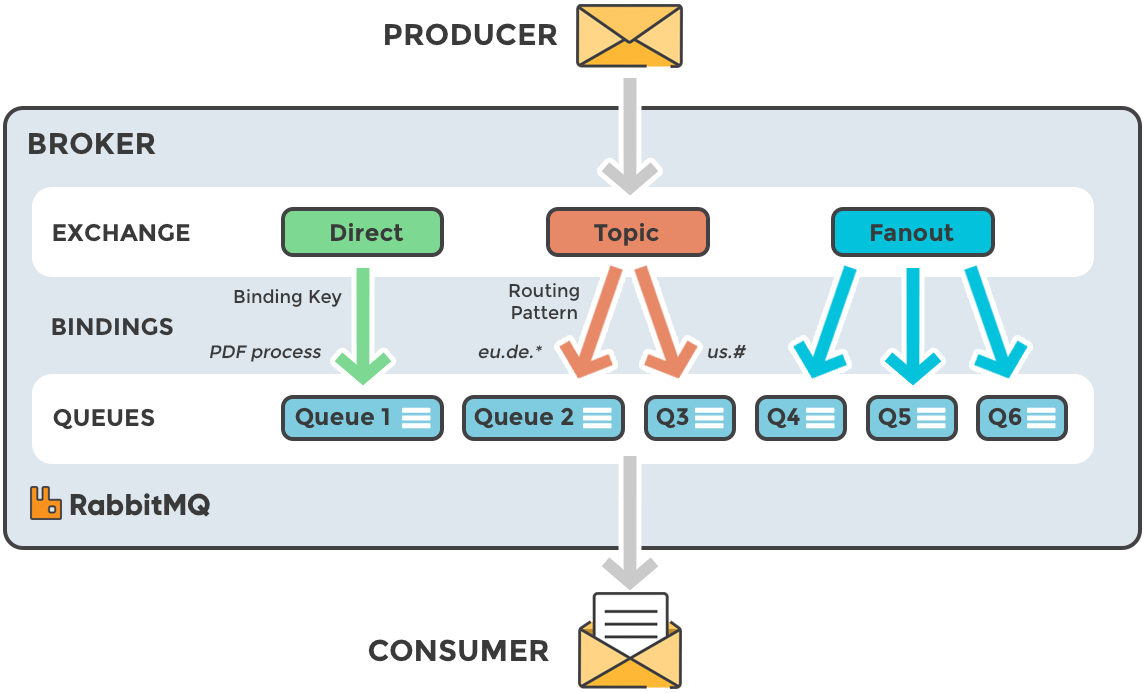

FIGURE 3.10 – Fonctionnement d'un workflow chorégraphique avec RabbitMQ - [71]

3.6.4 Complexités liées aux workflows

La mise en place d'un workflow peut être complexe, notamment lorsque plusieurs services et processus sont implémentés. Les workflows chorégraphiques peuvent être particulièrement difficiles à intégrer en raison de leur approche décentralisée, qui peut entraîner un manque de visibilité et de transparence sur les informations échangées. De plus, lorsque plusieurs services utilisant des protocoles différents sont impliqués, la communication entre ces services est fastidieuse. Les développeurs doivent implémenter des systèmes efficaces pour gérer la communication, ce qui complique la documentation de la partie serveur et la collaboration entre les différentes équipes [24]. Une manière de résoudre ce problème est d'utiliser *AsyncAPI*.

AsyncAPI

L'outil *AsyncAPI* peut être utilisé pour simplifier l'implémentation des workflows chorégraphiques. Il s'agit d'une spécification aidant à la documentation des architectures événementielles en fournissant un standard pour décrire les événements, la structure des messages, leur contenu et le protocole utilisé. Cette spécification permet de mieux comprendre le fonctionnement des services et facilite la communication entre les différentes équipes [82]. De plus, grâce à sa capacité à générer automatiquement du code à partir de la documentation, *AsyncAPI* facilite l'intégration d'un workflow et la communication entre les services [99].

La figure 3.11 illustre la documentation développée grâce à *AsyncAPI*. Cette documentation décrit les canaux de communication, les types de message et les paramètres acceptés

par les services. La partie droite montre le visuel de cette documentation. Elle peut ensuite être utilisée pour générer du code et faciliter l'intégration des différents services.

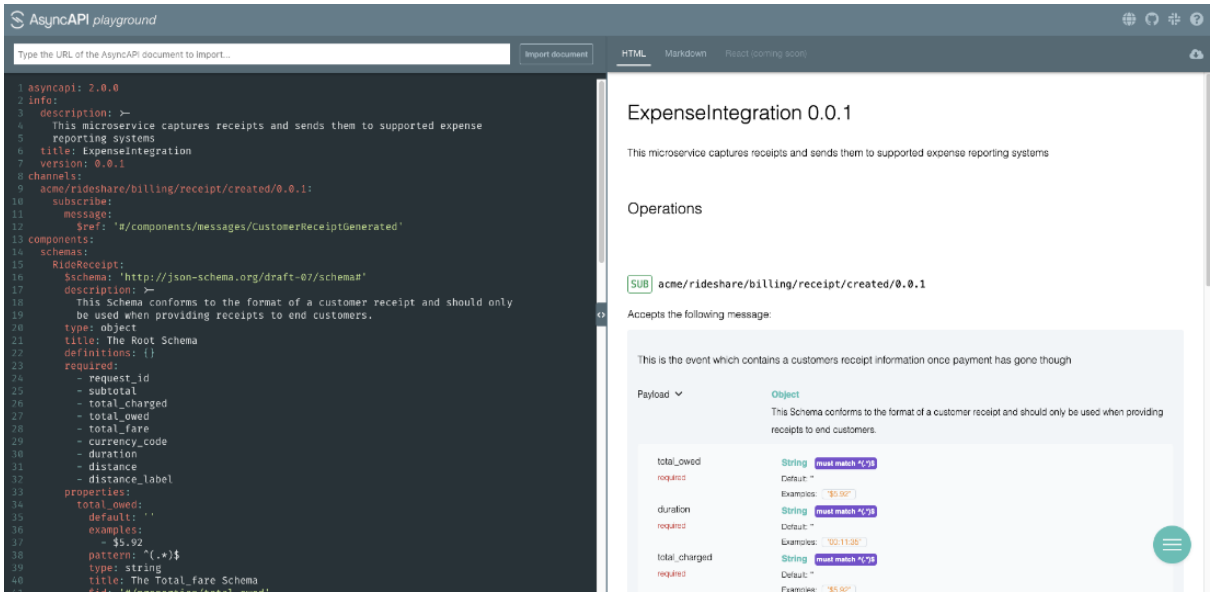

FIGURE 3.11 – Documentation AsyncApi - [90]

La commande illustrée dans la Figure 3.12 permet de créer un microservice Spring Cloud Stream en utilisant la documentation AsyncAPI de la figure 3.11. Les différents paramètres de la commande ne sont pas tous obligatoires et leur utilisation dépendra de nos besoins spécifiques. Généralement, l'utilisateur fournit des informations sur le langage de programmation, les paramètres de connexion (host, nom d'utilisateur, mot de passe, etc), l'emplacement de la documentation AsyncAPI et, si besoin, le template à utiliser.

```
ag -o ExpenseIntegration -p binder=solace -p view=provider -p actuator=
    true -p artifactId=ExpenseIntegration -p groupId=acme.rideshare -p
    javaPackage=acme.rideshare.expense -p host=localhost:55555 -p username
    =default -p password=default -p msgVpn=default ~/Downloads/
    ExpenseIntegration.yaml https://github.com/asyncapi/java-spring-cloud-
    stream-template.git
```

FIGURE 3.12 – Commande pour générer un microservice Spring cloud stream à partir de la documentation AsyncAPI - [90]

La figure 3.13 présente le projet qui a été créé à partir de la commande de la figure 3.12. Sa structure ressemble à celle d'un projet Spring Boot standard, avec différentes fonctions créées automatiquement. L'utilisateur peut ensuite ajouter sa propre logique métier. Le contenu du fichier Application.java est exposé dans la figure 3.14 et il est conforme aux informations (schémas, entrées, sorties, etc) spécifiées dans la documentation AsyncAPI (figure 3.11).

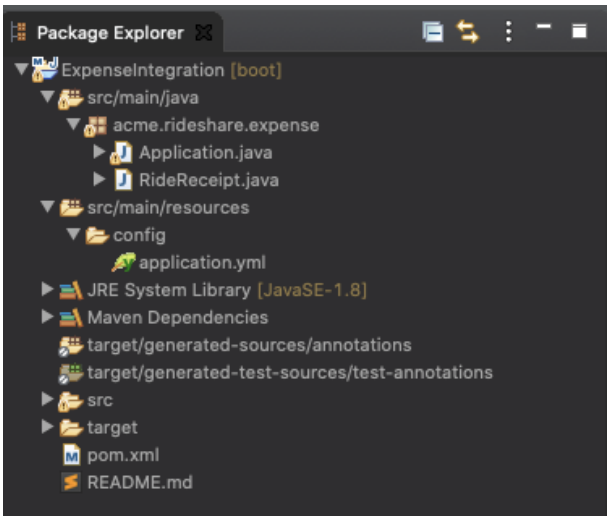

FIGURE 3.13 – Structure du projet créé par le générateur de code AsyncAPI - [90]

```java
@SpringBootApplication
public class Application {
    private static final Logger logger = LoggerFactory.getLogger(
        Application.class);
    public static void main(String[] args) {
        SpringApplication.run(Application.class);
    }

    @Bean
    public Consumer<RideReceipt>
        acmeRideshareBillingReceiptCreated001Consumer() {
        // Add business logic here.
        return null;
    }
}
```

FIGURE 3.14 – Contenu du fichier Application.java - [90]

3.6.5 Comparaison des workflows dans le cadre d'une entreprise

Le workflow n'est pas un outil indispensable pour coordonner les différents services. En effet, son utilisation peut être justifiée dans certaines situations. Par exemple, si une application comporte un ou plusieurs processus complexes, l'outil peut être bénéfique pour visualiser l'évolution de chaque processus. De plus, son utilisation est intéressante pour détecter les problèmes de communication entre les services. Toutefois, si ces conditions ne sont pas remplies, l'introduction d'un workflow risque de complexifier inutilement l'application [24].

Les workflows en orchestration et chorégraphiques ont chacun leurs avantages et leurs inconvénients. Lorsque l'objectif principal est de développer une application évolutive, le choix d'un workflow chorégraphique est plus judicieux. En effet, décentraliser les différents

services permet de mieux maîtriser la complexité de chacun. En revanche, l'utilisation d'un workflow en orchestration est plus avantageux lorsque nous cherchons à centraliser les différents services, à avoir un meilleur contrôle sur chacun et pouvoir visualiser la progression de manière plus précise [55][56].

3.7 Synthèse

Durant plusieurs années, les entreprises ont adopté une architecture monolithique pour leurs applications. Les services sont simples, avec peu de fonctionnalités, ce qui permet un développement et une mise en production rapides. Cependant, avec la croissance des entreprises et l'augmentation des services proposés, cette architecture a commencé à montrer ses limites. Les applications web deviennent de plus en plus complexes et l'interdépendance des services empêchent toute évolution, rendant la maintenance fastidieuse. De plus, les entreprises ont également remarqué que certaines fonctionnalités ont souvent été redéveloppées au sein de l'entreprise par manque de transparence. Afin de remédier à ces problèmes, les entreprises ont revu leur mode de développement des applications. Elles ont découpé leurs services en plusieurs services indépendants, plus petits et plus facilement maintenables, permettant ainsi de réduire la complexité des applications. Ainsi, les architectures SOA et les microservices ont vu le jour amenant de nouveaux défis. Notamment, comment peut-on faire communiquer les services pour fournir une logique métier complète sans créer de dépendance ? L'utilisation d'un workflow est une solution possible car il permet de coordonner les services et facilite l'échange d'informations tout en préservant l'indépendance des services. Finalement, l'utilisation de AsyncAPI simplifie la documentation des architectures et services événementiels et facilite l'intégration des workflows.

4

REST, GraphQL et gRPC

4.1 Définition d'une API 23
4.2 REST 24
4.3 Les fondements de REST 24
4.3.1 Endpoints 25
4.3.2 Interface http 25
4.3.3 Contraintes de développement 26
4.3.4 Hypermedia et HATEOAS 26
4.3.5 Les codes des états pour REST 27
4.4 GraphQL 28
4.4.1 Problématique de REST et fondement de GraphQL 28
4.4.2 Notion de Graphe 28
4.4.3 Principe de conception de GraphQL 29
4.4.4 Caratéristiques de GraphQL 29
4.5 gRPC 34
4.5.1 Protocole RPC 34
4.5.2 Principe de conception de gRPC 36
4.5.3 Type de communication client-serveur 36
4.5.4 gRPC et HTTP/2 37
4.6 Synthèse 38

Ce chapitre présente une API et trois technologies utilisées pour le développement de celle-ci: REST, GraphQL et gRPC. Ces trois technologies varient sur de nombreux critères mais sont utilisées dans le même but: exposer des services à travers une API.

4.1 Définition d'une API

Une API, *Application Programming Interface*, est l'interface d'un service qui peut être appelée ou exécutée par un autre service. L'API a pour but de cacher la complexité d'un système et de mettre à disposition, de l'utilisateur ou d'un service, une interface simple à utiliser. Cette interface permet pour découvrir le service, d'extraire les données mises à disposition et de les utiliser [20].

Pour permettre une meilleure utilisabilité, le développement d'une API doit satisfaire plusieurs critères [16][20]:

- **Simplicité d'utilisation**. Une API doit être facile à utiliser. Certaines API sont peu intuitives et requièrent un grand effort de compréhension, ce qui diminue la productivité des développeurs. Une API doit être ergonomique afin de faciliter son utilisation et doit être correctement documentée pour éviter toute mauvaise utilisation.
- **Interface uniforme**. L'interface doit être correctement structurée afin d'assurer son uniformité et faciliter son utilisation. L'emploi d'un outil de documentation peut aider à la mise en place de cette structure uniforme.
- **Evolutive**. Une API doit pouvoir évoluer avec le temps et s'adapter aux futurs changements. De nouvelles fonctionnalités peuvent être développées au cours du temps. Par conséquent, l'évolutivité s'accompagne avec une flexibilité de la structure de l'API. En effet, celle-ci doit avoir une structure robuste pour permettre de faciliter l'évolutivité.
- **Performance**: Pour assurer le bon fonctionnement du système, l'API doit être performante. Celle-ci peut se mesurer sur son temps de réponse et sur les ressources consommées.
- **Gestion des versions**. Les API évoluent constamment pour fournir de nouvelles fonctionnalités et répondre aux nouveaux besoins. Une gestion des versions efficace aide les utilisateurs à migrer d'une version à une autre sans causer de dysfonctionnements.
- **Sécurité**. La sécurité est un élément primordial des API. De grandes quantités de données circulent à travers les requêtes. Ces données doivent être sécurisées. Une mauvaise utilisation ne doit en aucun cas causer des problèmes de sécurité et exposer des données sensibles.

4.2 REST

REST, *Representational state transfer*, est un style d'architecture logicielle qui définit un ensemble de règles à respecter lors du développement d'une API. Il est devenu un standard incontournable pour le développement d'application web [29].

4.3 Les fondements de REST

REST est apparu avec le développement et l'évolution du web. À cette époque, les chercheurs s'interrogent sur la croissance exponentielle d'internet et cherchent à en définir les premiers fondements. Durant cette période, Roy Fielding, dans le cadre de son doctorat, généralise les principes architecturaux du web. Il présente dans son travail comment les systèmes d'informations distribués sont construits et exploités, et comment les données circulent. Il souhaite définir un style architectural, avec un nombre limité d'opérations, capable de supporter tout type d'application. Il énonce cela sous un framework de contraintes, appelé *Representational state transfer*. REST décrit le web comme une application constituée d'hypermédia qui s'échange des ressources [29].

REST se base sur deux éléments importants du web : les ressources et les URI. Les ressources représentent toutes les informations qui sont exposées sur le web. Il peut s'agir d'un document, d'une image ou d'une vidéo. Ces éléments sont identifiables grâce à un *Uniform Resource Identifier* ou URI. URI peut être considéré comme un identifiant unique de la ressource. Une ressource peut avoir plusieurs URI mais un URI n'est lié qu'à une seule ressource [29].

4.3.1 Endpoints

Un endpoint, ou URL, définit le schéma d'accès à une ressource. Il est utilisé avec le protocole HTTP afin d'accéder à une ressource. Les endpoints REST doivent suivre le principe d'uniformité. La structure suivante doit être respectée :

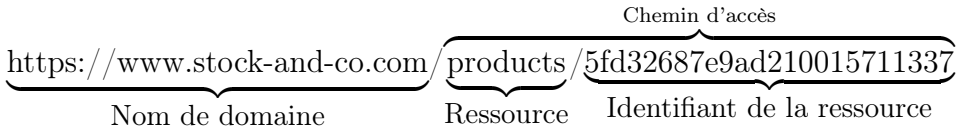

FIGURE 4.1 – Structure d'un endpoint

Cette structure est composée de plusieurs éléments:

- **Un protocole (http ou https).** REST se base sur le protocole http ou https. La communication s'effectue via le port 80 (http) ou 443 (https).
- **www.** Le www indique qu'il s'agit d'un site web. Ce champ peut, en général, être omis car il est implicitement défini.
- **stock-and-co.com** Il s'agit du nom de domaine. Il permet au serveur DNS de retourner l'adresse IP correspondant. Il est souvent plus facile pour l'utilisateur de le retenir.
- **products/5fd32687e9ad210015711337.** Il s'agit du chemin d'accès vers la ressource. Le "products" indique que la ressource est de type "produit" et l'identifiant après le slash permet d'identifier précisément la ressource souhaitée.

Il est également possible d'ajouter des paramètres supplémentaires. Ils peuvent être ajoutés à la fin de l'url sous la forme " ?paramètre=valeur". Les paramètres ne sont traités par le serveur que si l'implémentation le permet.

4.3.2 Interface http

Le protocole HTTP met à disposition plusieurs verbes pour décrire les actions qui peuvent être effectuées. REST repose sur ce protocole et utilise cinq de ses verbes [29]:

- **GET**. Permet de récupérer une ressource.
- **POST**. Permet de créer une ressource.
- **PUT**. Permet de mettre à jour une ressource. Un PUT réécrit entièrement une ressource.
- **PATCH**. Permet de mettre à jour partiellement une ressource. Seuls les champs transmis dans le PATCH seront mis à jour.
- **DELETE**. Permet de supprimer une ressource.

Pour que ces verbes puissent être utilisés, l'URL doit supporter l'action.

4.3.3 Contraintes de développement

Une API REST doit respecter six contraintes lors de son développement [92] :

- **Client-serveur**. Le client et le serveur doivent être indépendants pour permettre une meilleure synergie et évolution.
- **Sans état**. Les communications doivent se faire sans connaissance de l'état. Le serveur ne doit posséder et recevoir que les données nécessaires pour répondre correctement à une requête.
- **En couche**. L'architecture REST doit diviser plusieurs couches pour faciliter son évolution. Le client ne doit pas savoir à quelle couche il s'adresse.
- **Interface uniforme**. La méthode de communication doit être la même pour toutes les ressources. Les ressources doivent être identifiables, via des URI, représentables et auto-descriptives. L'accès doit se faire via des requêtes (GET, POST, PUT, PATCH ou DELETE) et le serveur retourne une réponse contenant un corps et une entête.
- **Mise en cache**. Les réponses de l'API REST peuvent être stockées temporairement en mémoire. Lorsqu'une requête similaire est de nouveau exécutée, la nouvelle réponse mettra moins de temps à s'effectuer.
- **Code à la demande**. Cette dernière contrainte est facultative. Le serveur doit être capable de transmettre du code au client pour que celui-ci l'exécute. Cette contrainte permet d'alléger la charge du serveur en transférant une partie au client.

4.3.4 Hypermedia et HATEOAS

Dans les sections précédentes, nous avons vu que REST interagit avec une ressource à travers une URI. Néanmoins, cette interaction peut être considérée comme statique. En effet, l'API effectue l'action demandée mais ne nous permet pas de la découvrir. Par exemple, il n'est pas possible, en général, de savoir si une ressource supporte l'action GET ou POST sans l'avoir essayée au préalable. Cette situation peut être problématique lorsque nous utilisons une API avec de nombreux endpoints. Imaginons, par exemple, que nous utilisons une API mais n'avons pas accès à la documentation. Comment faire pour découvrir les actions et les endpoints disponibles ? La découverte de l'API peut être simplifiée grâce au hypermédia et au concept HATEOAS.

HATEOAS, qui signifie *Hypermedia As The Engine of Application State*, est un concept important de REST. A l'aide d'une simple requête vers un endpoint, HATEOAS permet de retourner, en plus des ressources demandées, des informations complémentaires qui facilitent la découverte de l'API. Comme pour les sites web et leur système de navigation, l'utilisateur peut naviguer d'une URL à l'autre [2]. Prenant l'exemple, d'un endpoint retournant des informations sur un utilisateur.

Pour illuster ce concept, la figure 4.2 montre une réponse classique:

```
{
  "customerID": 5,
  "name": "Thomas"
}
```

FIGURE 4.2 – Exemple de réponse sans HATEOAS

Cette réponse peut être suffisante. Cependant, si l'utilisateur n'a pas accès à la documentation, il manque d'informations sur les actions possibles et les endpoints disponibles. Une API REST utilisant le principe HATEOAS retournera une réponse comme celle de la figure 4.3. En plus de la ressource, l'utilisateur reçoit, dans l'entête "Links", des instructions additionnelles qui lui permettent de découvrir d'autres endpoints. De ce fait, l'utilisateur prend dynamiquement connaissance de la capacité de l'API.

```
{
  "customerID": 5,
  "name": "Thomas",
  "Links": [
    {
      "href": "http://localhost:8080/customer/5",
      "action": "PUT"
    },
    {
      "href": "http://localhost:8080/customer/5",
      "action": "DELETE"
    },
    {
      "href": "http://localhost:8080/customer/5/products",
      "action": "GET"
    }
  ]
}
```

FIGURE 4.3 – Exemple de réponse HATEOAS

4.3.5 Les codes des états pour REST

Pour chaque requête, le serveur retourne une réponse accompagnée d'un code indiquant son état. Cinq catégories de code existent :

- **1xx** indique que le serveur transmet une information. (Par exemple: le code 101 signifie que le changement de protocole a été accepté.)
- **2xx** indique que la requête a pu être traitée avec succès.
- **3xx** indique une redirection. Le client doit effectuer des actions supplémentaires pour traiter la requête (par exemple : le document souhaité a été déplacé.).
- **4xx** indique que l'erreur provient du client (par exemple: les droits d'accès sont refusés).

- **5xx** indique que l'erreur provient du serveur (par exemple: connexion impossible avec le serveur).

Nous avons dorénavant une bonne compréhension de REST. La section suivante présente GraphQL.

4.4 GraphQL

GraphQL est un langage de requête développé par Facebook en 2012. Son principal but est de résoudre les problèmes de performance liés aux API REST [19]. GraphQL communique au travers du protocole HTTP.

4.4.1 Problématique de REST et fondement de GraphQL

GraphQL a été développé pour trouver une solution aux problèmes de performance liés aux API REST. En 2012, lorsque Facebook demande à ses ingénieurs de développer une nouvelle version de leur application mobile, ils se rendent compte que leur API REST n'arrive pas à répondre aux requêtes de manière optimale. Deux raisons expliquent ce problème. La première est la quantité de données retournées. REST retourne tous les attributs, même ceux non utilisés. Les requêtes sont allourdies et prennent du temps à être traitées. La deuxième raison est liée à la fusion de plusieurs sources de données. Lorsque les données proviennent de plusieurs sources (API, base de données, etc), REST nécessite plusieurs requêtes, au minimum une par API. Suivant l'architecture, le nombre de requêtes peut être très élevé amenant des problèmes de performance. GraphQL résout ces deux problèmes en offrant la possibilité de ne retourner que les attributs souhaités et en simplifiant la récupération des données de plusieurs sources [19].

4.4.2 Notion de Graphe

GraphQL adopte une structure inspirée des graphes pour organiser ses données. Un graphe est une structure de données composées d'un ensemble de points, appelés noeuds, qui sont reliés par des lignes, appelées arêtes. Les graphes permettent de créer des relations entre différents éléments (ressources, objets, etc). Par exemple, le graphe de l'entreprise Stock&Co, illustré par la figure 4.4, met en relation les produits vendus par l'entreprise avec les commandes et les utilisateurs. Le système de graphe permet de facilement visualiser les relations entre les données et de passer d'un noeud à l'autre. GraphQL adopte ce même principe et permet, grâce à des requêtes, de récupérer des informations sur un noeud et sur les noeuds adjacents. Cet aspect permet de réduire le nombre de requêtes [19].

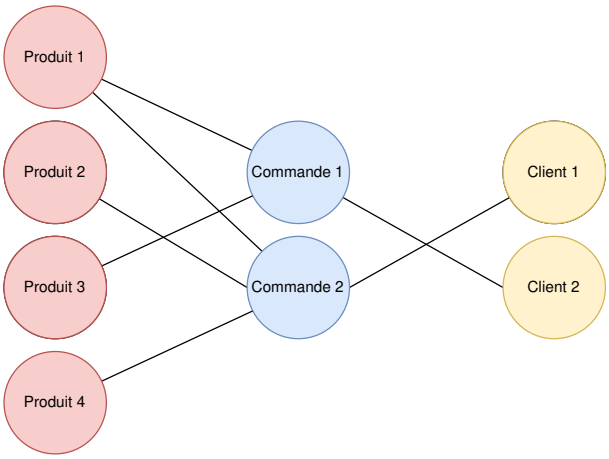

FIGURE 4.4 – Graphe de Stock&Co

4.4.3 Principe de conception de GraphQL

GraphQL propose quelques lignes directrices à prendre en compte lors du développement de l'API [19] :

- **Structure hiérarchique.** Les requêtes GraphQL ont une structure hiérarchique. Les champs sont imbriqués les uns dans les autres et la réponse a la même forme que la requête.
- **Centré produit.** GraphQL est dirigé par le langage de programmation et l'environnement d'exécution du client. Le client est décisionnaire. En effet, GraphQL lui met à disposition les données nécessaires sous la forme souhaitée.
- **Fort typage.** Chaque attribut de GraphQL a un type défini. À chaque requête, les attributs sont contrôlés et leur type est validé. Par exemple, si un nombre est défini comme valeur dans un attribut de type texte, la requête échoue avant même qu'elle ne soit envoyée au serveur.
- **Réponse spécifique au client**. Le client a le contrôle sur toutes les données qu'il récupère du serveur, que ce soit les attributs ou les relations. Le serveur fournit les droits nécessaires au client. Ce principe évite la récupération de données inutiles.
- **Introspective.** GraphQL peut être interrogé pour obtenir plus de détails sur les fonctions mises à disposition, les attributs et de leurs spécifications.

Dans la prochaine section, nous allons voir les concepts de GraphQL.

4.4.4 Caratéristiques de GraphQL

GraphQL est composé de plusieurs concepts fondamentaux [40] :

- les schémas
- les Queries
- les Mutations
- les Subscriptions
- l'introspection. Ce point est décrit dans le prochain chapitre.

Ces concepts permettent de mieux comprendre le développement d'un API en GraphQL.

Les schémas

GraphQL adopte une structure hiérarchique appelée schéma. Un schéma, comme sur la figure 4.5, définit la structure des ressources, la liste des attributs, leurs types et les relations entre les ressources. Dans les sections suivantes, nous verrons également que le schéma contient les fonctions permettant d'interagir avec les ressources.

```
type Order {
  _id: String
  createdAt: timestamptz,
  products: [String]
  user: User
  status: String
}

type User {
  _id: String
  name: String
  email: String
  points: Int
  orders: [Order]
}
```

FIGURE 4.5 – Schéma Order et User

Deux schémas sont définis dans la figure 4.5. Le premier représente la structure d'une commande et la deuxième celui d'un utilisateur. Le schéma Order contient un attribut user de type User (même type que l'utilisateur). Cela signifie que les deux schémas ont une relation et qu'il est possible de passer d'un schéma à l'autre. La définition du schéma est primordiale dès le début du développement car les requêtes et leurs réponses auront la même structure.

Les Queries

Le terme Query, spécifique à GraphQL, est l'équivalent du verbe HTTP GET. En GraphQL, la récupération des données s'effectue en utilisant une fonction de type Query définie dans le schéma au préalable. Reprenons l'exemple de la figure 4.5 et ajoutons lui un schéma de type Query (figure 4.6).

```
type Query {
  users: [User]
  orders: [Order]
}
```

FIGURE 4.6 – Schéma d'une query GraphQL

Le schéma Query contient deux fonctions: **users** pour retourner la liste des utilisateurs et **orders** pour retourner la liste des commandes. Par convention, les Queries portent le

même nom que la ressource retournée. Par conséquent, la fonction qui retourne la liste des utilisateurs s'appelle **users** et non **getUsers**. Une requête GraphQL de type Query ressemble à la figure 4.7.

```
query {
  users {
    name
    email
  }
}
```

FIGURE 4.7 – Query GraphQL

La réponse de cette Query est représenté par la figure 4.8 et suit la même structure.

```
{
  "data": {
    "users": [
      {
        "name": "Dimitri",
        "email": "dimitri@boscova.xyz"
      },
      {
        "name": "Gustave",
        "email": "gustave@flobinou.xyz"
      }
    ]
  }
}
```

FIGURE 4.8 – Réponse de la figure 4.7

Cette Query peut également être plus complexe, comme illustrée par la figure 4.9. En partant de la Query **users**, nous pouvons obtenir des informations sur les commandes.

```
query {
  users {
    name
    email
    orders {
      _id
      createdAt
    }
  }
}
```

FIGURE 4.9 – Obtention des informations sur les commandes de l'utilisateur

La figure 4.10 montre que l'utilisateur **dimitri** possède une commande avec l'**id** appelé **1** et l'attribut **createdAt** contenant la valeur **2023-01-04**. En revanche, pour l'utilisateur **Gustave**, aucune commande n'a été retournée et cela est visible par les crochets vides de l'attribut **orders**.

```
{
  "data": {
    "users": [
      {
        "name": "Dimitri",
        "email": "dimitri@boscova.xyz",
        "orders": [
          {
            "_id": "1",
            "createdAt": "2023-01-04"
          }
        ]
      },
      {
        "name": "Gustave",
        "email": "gustave@flobinou.xyz",
        "orders": []
      }
    ]
  }
}
```

FIGURE 4.10 – Réponse de la figure 4.9

Les Mutations

Le terme Mutation est utilisé pour tout type de modifications sur les ressources [19]. Il est l'équivalent des verbes HTTP POST, PUT et DELETE. Reprenons le schéma de la figure 4.5 et définissons une Mutation. Cela est représenté par la figure 4.11.

```
register(name: String!, email: String!, password: String!): User
```

FIGURE 4.11 – Schéma d'une Mutation GraphQL

Notre Mutation prend trois arguments obligatoires, marqués par un point d'exclamation, et retourne l'utilisateur créé. Contrairement aux Queries, il est généralement obligatoire de donner un nom aux Mutations, comme illustré par la figure 4.12. Dans notre cas, la Mutation **register** s'appelle **registerUser**. De plus, les arguments sont séparés de la Mutation et envoyer à part. Cela améliore grandement la lisibilité.

Tous les arguments de la Mutation ne sont pas obligatoirement définis dans le schéma. Par exemple, l'argument **password** n'est pas défini dans la structure du **User** à la figure 4.5. En effet, cela permet de protéger les données et évite que certaines données sensibles

soient accessibles à travers des Queries. Sur la figure 4.12, nous remarquons que notre Mutation a été réalisée. Cependant, l'argument password n'apparaît pas car celui-ci n'a pas été spécifié dans le schéma. Si l'utilisateur tente d'accéder à un attribut non défini, la Mutation échoue, comme représentée par la figure 4.13.

```
mutation registerUser($name: String!, $email: String!, $password: String
    !) {
  register(name: $name, email: $email, password: $password) {
    name
    email
  }
}

{
  "name": "Jean-Paul Caillier",
  "email": "jeanpaul@caillier.xyz"
}
```

FIGURE 4.12 – Exemple Mutation

```
mutation registerUser($name: String!, $email: String!, $password: String
    !) {
  register(name: $name, email: $email, password: $password) {
    name
    email
    password #cet attribut est inexistant
  }
}

{
  "erreur": "attribut inexistant"
}
```

FIGURE 4.13 – Exemple Mutation non autorisée

Subscription

Les Subscriptions permettent d'être informées, en temps réel, des changements effectués sur un élément. Par exemple, il est possible d'afficher en direct la liste des commandes et le changement des statuts. Les Subcriptions fonctionnent grâce à un webSocket. Contrairement aux Queries et Mutations, les Subscriptions retournent des données qu'en cas de changements. La définition du schéma et l'utilisation des Subscriptions sont similaires aux Queries et Mutations [19]. La figure 4.14 illustre la définition d'un Subscription permettant de suivre les changements de statut d'une commande. La fonction **orderStatusChanged** prend comme argument le numéro d'une commande et informe l'utilisateur si le statut de la commande change. La figure 4.15 est la requête permettant de démarrer le Subscription. Elle est faite sur la commande ayant comme id **12345**.

```
type Subscription {
  orderStatusChanged(orderId: String!): Order
}
```

FIGURE 4.14 – Schéma d'une Subscription GraphQL

```
subscription {
  orderStatusChanged(orderId: "12345") {
    _id
    status
  }
}
```

FIGURE 4.15 – Exemple subscription

La définition des schémas, type, Queries, Mutations et Subscriptions varient en fonction du langage de programmation utilisé. Les exemples présentés utilisent le langage GraphQL pur. La syntaxe ne sera pas la même si le serveur est codé avec GraphQL pour NodeJS ou Python.

Cependant, la structure des requêtes (Queries, Mutations et Subscriptions) reste la même peu importe le langage utilisé.

Dans la prochaine section, nous allons discuter de gRPC.

4.5 gRPC

gRPC est un framework open source basé sur le protocole RPC. Il a été développé par Google et rendu public en 2016. La fonction principale de gRPC est de permettre à des composants indépendants, tels que les microservices, de communiquer entre eux plus facilement [25]. Le protocole de communication est HTTP/2.

4.5.1 Protocole RPC

RPC, ou *remote procedure call*, est un protocole réseau qui permet à un ordinateur de communiquer avec un autre sans connaître les détails du réseau. Il s'agit d'une des premières formes d'API [5]. Ce protocole est simple à utiliser mais possède un grand défaut. Contrairement aux API modernes, telles que REST, RPC manque de clarté et l'ensemble du travail repose sur la partie client. Le client doit connaître les fonctions existantes, savoir lesquelles appeler et où aller chercher l'information. Ce point complexifie le développement d'une application avec le protocole RPC, notamment lorsque plusieurs services distincts sont utilisés [62].

Le modèle conceptuel de RPC est différent de celui des API HTTP tel que REST. REST est défini comme une architecture centrée sur les ressources. Les ressources sont l'élément principal de l'API. Les endpoints donnent le chemin d'accès à cette ressource. Plusieurs endpoints similaires peuvent exister et c'est le verbe HTTP qui définit l'action qui se

déroule. RPC, quant à lui, est utilisé pour effectuer des actions. L'endpoint ne permet pas d'accéder à une ressource mais à une méthode. Les verbes HTTP ont, par conséquence, peu d'importance [5]. Les données retournées dépendent également de ces méthodes [43].

La figure 4.16 illustre trois endpoints permettant d'interagir avec une ressource de type commande.

```
GET /orders/order
POST /orders/order
DELETE /orders/order
```

FIGURE 4.16 – HTTP REST - Action caché derrière la ressource

Le chemin d'accès à la ressource est identique pour les trois verbes HTTP. La figure 4.17 présente ces mêmes endpoints, mais pour une API RPC.

```
GET /orders/getOrder
POST /orders/createOrder
DELETE /orders/deleteOrder
```

FIGURE 4.17 – RPC - Données cachées derrière les actions

Les trois URL sont différentes. Le chemin d'accès ne pointe pas sur une ressource mais sur une méthode. Le verbe HTTP permet de mieux visualiser l'action qui sera effectuée mais n'a pas d'impact direct sur les changements.

La section suivante présente la langage protoBuf qui est la base de gRPC.

Le langage protocol Buffers

Protocol Buffers, ou protoBuf, est un langage descriptif utilisé par gRPC. Il permet de définir la structure des composants en spécifiant les différentes méthodes et les arguments. Ce fichier est ensuite utilisé par gRPC pour générer des clients et des serveurs. La figure 4.18 présente la structure d'un fichier protoBuf.

```
syntax = "proto3";

service userService {
  rpc getUserById (UserRequest) returns (UserReply) {}
}

message UserRequest {
  int32 Id = 1;
}
message User {
  string email = 1;
  string name = 2;
}
```

FIGURE 4.18 – Exemple de fichier .proto

Le fichier protoBuf de la figure 4.18 est composée de trois parties [5]:

- La première partie (en rouge) contient les informations concernant le fichier. On y retrouve le langage de programmation choisi, les librairies utilisées et l'importation d'autres fichiers protoBuf.
- La deuxième partie (en vert) contient la liste des services proposés. On y définit les inputs et les outputs de chaque procédure. Un service peut contenir plusieurs procédures.
- La dernière partie (en jaune) contient les messages. Les différentes structures des objets y sont définies.

Le fichier peut ensuite être compilé. Les procédures sont réécrites automatiquement dans le langage de programmation souhaité. De ce fait, il est très facile de développer plusieurs microservices dans différents langages de programmation avec un seul fichier protoBuf [5]. La figure 4.19 illustre la conversion d'un fichier protoBuf vers plusieurs langages de programmation.

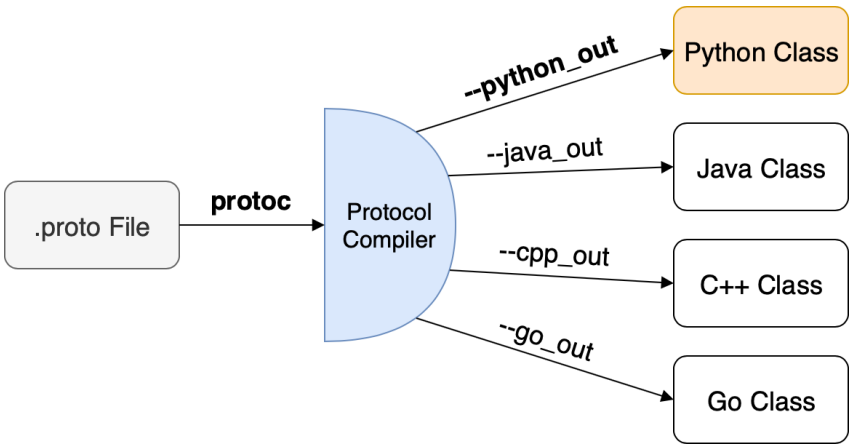

FIGURE 4.19 – Fichier .proto compilé vers d'autres langages - [74]

4.5.2 Principe de conception de gRPC

gRPC a pour but de faciliter la communication entre divers composants tels que les microservices. Ces composants ont, en général, été développés dans plusieurs langages de programmation différents. gRPC part de ce principe et permet, grâce au langage protoBuf, de définir une structure commune et ensuite de générer des composants dans différents langages de programmation. gPRC permet également de décider si le composant à générer est un client ou un serveur. De ce fait, la communication entre les différents composants est facilitée car ils possèdent tous la même base et chaque composant peut plus facilement découvrir les méthodes disponibles [5].

4.5.3 Type de communication client-serveur

gRPC propose quatre types de communication client-serveur, illustré par la figure 4.20 [5]:

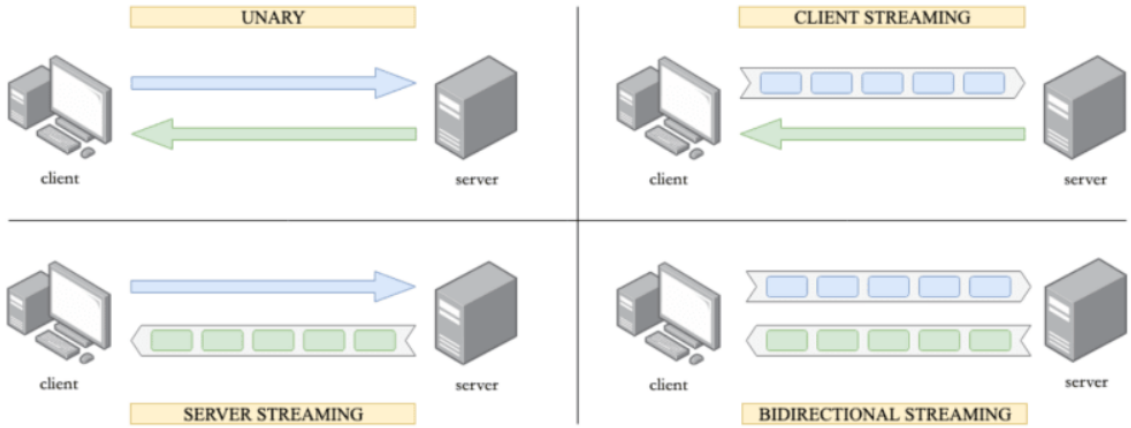

FIGURE 4.20 – Types de communication client-serveur avec gRPC - [5]

- **Unidirectionnel**. Le client initialise un appel de procédure, le serveur l'exécute et retourne une réponse. Il s'agit du type de communication le plus simple. Cette fonctionnalité est la seule disponible pour REST et GraphQL.
- **Streaming client**. Le client initialise un appel de procédure et envoie plusieurs messages. Le serveur les traite et ne retourne qu'une seule réponse.
- **Streaming serveur**. Le client initialise un appel de procédure. Le server répond en envoyant une séquence de message.
- **Bidirectionnel**. Le client et le serveur communique par requêtes et réponses en continu. Le client n'a pas besoin d'attendre la réponse avant d'envoyer une nouvelle requête. Il n'y a pas d'ordre défini pour les message envoyés et reçus.

Afin de minimiser les erreurs, il est courant de spécifier une limite de temps d'exécution pour les requêtes. Si ce délai est dépassé, l'expéditeur annule sa requête.

4.5.4 gRPC et HTTP/2

Contrairement à REST et GraphQL, gRPC ne se base pas sur le protocole HTTP/1.1 mais HTTP/2. Avec l'évolution du web, le protocole HTTP/1.1 ne répond plus aux besoins actuels. Il n'est, par exemple, pas possible d'effectuer du streaming avec REST. Cette limitation peut être contournée grâce à l'utilisation de webSocket mais cela se fait au détriment de la performance. HTTP/2 a été développé dans le but d'optimiser la performance du protocole HTTP en réduisant le temps de réponse. Pour cela, de nouvelles stratégies ont été implémenter pour faciliter la communication client-serveur [5]:

- **Multiplexage**. Il n'est plus nécessaire d'ouvrir une nouvelle connexion pour chaque requête comme cela est le cas pour HTTP/1.1. Une seule connexion est suffisante pour envoyer plusieurs requêtes et recevoir plusieurs réponses. Le streaming gPRC est basé sur cette stratégie.
- **Compression des données**. HTTP/2 permet le transport de données binaires. Les données sont moins volumineuses et circulent plus rapidement.
- **Contrôle du flux**. Il est possible d'optimiser le flux des données. Par exemple, le client peut demander au serveur de stopper l'envoi des données pour une certaine durée. Cela permet de ne pas surcharger inutilement le réseau.

Ces nouvelles stratégies profitent à gRPC et le rendent plus performant que REST et GraphQL. Néanmoins, l'utilisation de HTTP/2 apporte quelques inconvénients:

- **Problème de compatibilité**. Les navigateurs web actuels ne supportent pas l'interprétation des données binaires, ce qui rend gRPC difficilement utilisable pour développer une application client-serveur. De plus, HTTP/2 est obligatoire pour le fonctionnement de gRPC mais il n'existe aucun moyen de forcer un navigateur web à utiliser ce protocole.
- **Données illisibles pour l'homme**. Contrairement à JSON et XML, les données binaires sont difficilement lisibles par l'homme. Par conséquent, il est très difficile de déchiffrer les informations qui circulent. Cela est un avantage en termes de sécurité mais un problème majeur lorsqu'il s'agit de débugger.

4.6 Synthèse

Dans ce chapitre, trois technologies ont été présentées: REST, GraphQL et gPRC. REST est un style architectural qui définit des contraintes à respecter pour développer une API. Il est simple à mettre en place et facilite l'accès aux ressources grâce à des endpoints uniformes et structurés. Cependant, avec l'évolution du web, REST montre ses limites. La quantité de données circulant à travers les requêtes est de plus en plus importante. Ces limites créant des problèmes de performance, Facebook a développé GraphQL, un langage de requête. GraphQL permet de filtrer les attributs, de joindre plusieurs sources de données, et finalement, de parcourir les ressources en utilisant le principe de graphe. GraphQL inverse les rôles et définit sa conception en partant du client.

Avec l'expansion du web et le développement de nombreux composants indépendants, la communication entre les différents éléments est devenue plus complexe. Google développe le framework gPRC pour simplifier la communication entre les différents composants. Le framework utilise le langage ProtoBuf pour définir sa structure et le protocole HTTP/2 pour la communication. gPRC est plus performant que REST et GraphQL mais son utilisation n'est actuellement pas entièrement supportée par les navigateurs web. De ce fait, il est, par exemple, difficile de développer des applications client-serveur avec cette technologie.

Dans le prochain chapitre, nous allons comparer ces trois technologies en nous basant sur la littérature et les recherches scientifiques.

5
Analyse comparative des API

5.1 Cas d'utilisation **40**
5.2 Exposition des API **41**
5.3 Comparaison des API selon les critères 3 à 10 . . . . . **46**
5.4 Analyse de la performance: temps de réponse des API . . . **49**

Ce chapitre compare les technologies REST, GraphQL et gRPC en se basant sur la littérature et les recherches scientifiques. La comparaison se fait sur plusieurs points [35][30][5]:

1. **Cas d'utilisation**. Il s'agit de définir dans quelles situations l'utilisation des différentes technologies est la plus appropriée.
2. **Exposition des API.** Exposer une API signifie rendre disponibles les services d'une API aux utilisateurs ou aux services externes. Les personnes ou les appareils peuvent atteindre les services mis à disposition grâce à des endpoints. Ce point compare les diverses méthodes utilisées par les API pour exposer leurs données.
3. **Complexité de développement.** Ce point mesure la complexité liée à la définition de la structure de l'API et à son développement pour un développeur backend.
4. **Complexité de développement d'un client pour l'API.** Il s'agit d'évaluer avec quelle facilité l'API peut être prise en main par un développeur frontend. Ce critère couvre la compréhension de l'API et sa documentation et le développement d'un client utilisant cette API.
5. **Mise en place d'outils de redirection.** La plupart des entreprises utilisent des outils, tels que des middlewares ou des gateways, pour rediriger le trafic entrant sur l'API vers les services appropriés. L'ajout de cet outil permet de mieux sécuriser l'API et les différents services. Cet élément évalue la complexité de mettre en place un tel outil en fonction de la technologie choisie.
6. **Composition de l'API.** La composition d'une API implique le développement d'une nouvelle API en intégrant des fonctionnalités provenant de plusieurs API existantes. Plusieurs raisons peuvent favoriser cette composition: coordonner les API pour fournir des fonctionnalités plus complexes ou combiner des ressources pour transmettre plus d'informations à travers un endpoint. Les trois technologies sont comparées sur leur favorabilité à composer une API.
7. **Authentification et autorisation.** Cet aspect évalue si la technologie choisie facilite la mise en place d'un système de sécurité et dans quelle mesure.

8. **Caching.** Le caching permet d'accélérer les requêtes en stockant temporairement les données en mémoire. Ce point compare la mise en place du caching en fonction de la technologie.
9. **Gestion des versions.** Ce point compare les méthodes disponibles pour gérer les versions des différentes API.
10. **Mode de communication.** Cet aspect présente les divers modes de communication proposés par chaque technologie.
11. **Performance**. Ce critère compare le temps de réponse des API REST, GraphQL et gRPC.

La section 5.1 présente les cas d'utilisation des trois API. La section 5.2 compare l'exposition des API. La section 5.3 couvre les critères 3 à 10. Finalement la section 5.4 évalue la performance des API.

5.1 Cas d'utilisation

REST

REST est un choix optimal lorsqu'une API nécessite une certaine flexibilité tout en gardant un cadre lors de sa conception. En effet, les contraintes définies par REST guident le développeur dans la conception de l'API tout en lui laissant le choix sur les aspects comme le langage de programmation ou la structure des données. De plus, REST permet de développer des API performantes et robustes car il applique des principes d'optimisation pour réduire le temps de réponse des requêtes. REST permet également de diminuer la charge du serveur. Par exemple, le principe de mise en cache permet de répondre plus rapidement à une requête si une requête similaire a déjà été effectuée auparavant. De même, le système de code à la demande réduit la charge du serveur en transférant une partie du code au client. L'API est solide et résistante à de nombreuses situations. Le troisième critère en faveur de REST est qu'il peut être considéré comme une architecture facilitant le développement d'API polyvalentes. Généralement, il n'est pas possible de connaître à l'avance tous les clients qui consommeront l'API REST. De ce fait, il est primordial de fournir toutes les informations disponibles sur une ressource. Le client peut ensuite filtrer les attributs nécessaires à son fonctionnement. Aujourd'hui, REST est le standard en entreprise grâce à sa maturité. C'est le style d'architecture le plus simple et facile à mettre en place. Quand le besoin n'est pas spécifique, REST est la meilleure option [35].

GraphQL

GraphQL a été développé pour résoudre deux problèmes majeurs liés à REST. De ce fait, il est optimal dans deux situations:

- Lorsque la réponse à une requête est constituée de données provenant de plusieurs sources.
- Lorsque plusieurs clients doivent accéder à différents attributs des ressources de façon optimal.

La première situation est un cas où l'accès aux données se fait via divers API. Avec REST, la récupération des ressources demande au minimum une requête par API. Par exemple, pour obtenir la liste des messages envoyés par chaque utilisateur, deux requêtes sont nécessaires: une sur l'API utilisateur et une autre sur l'API message. Le fait d'effectuer plusieurs requêtes alourdit l'API et réduit ses performances. Avec GraphQL, plusieurs API peuvent être combinées à travers un seul schéma. En effet, une seule requête suffit pour récupérer toutes les informations nécessaires. Les performances de l'API sont, par conséquent, meilleures [35].

La deuxième situation est lorsque l'API doit être optimale pour des clients de supports différents. Par exemple, une entreprise peut offrir ses services à travers une application web et une application mobile. Les deux interfaces sont différentes et les éléments affichés dépendent du support. Dans ce cas de figure, GraphQL est un choix optimal car le client peut, lors de sa requête, sélectionner uniquement les attributs nécessaires. Par conséquent, les performances de l'API et du client sont maintenues [35].

gRPC

gRPC est une technologie performante pour la communication des composants indépendants, comme les microservices. La performance de gRPC est principalement liée à l'utilisation du protocole HTTP/2 qui permet de réduire le volume des données, la consommation des ressources et d'accélérer la vitesse de transmission. De ce fait, gRPC est adapté aux services nécessitant une communication rapide, ainsi qu'aux appareils à faible consommation énergétique, tels que les IoT ou les appareils mobiles. Finalement, gRPC propose la communication bidirectionnelle, ce qui en fait une technologie idéale pour le streaming [5].

5.2 Exposition des API

REST

L'exposition des API REST implique plusieurs étapes. Premièrement, il est nécessaire de définir les ressources qui seront exposées, ainsi que les endpoints et les verbes HTTP associés à chaque ressource. Ensuite, le format des données doit être choisi. La représentation des ressources peut se faire soit en JSON soit en XML. Selon la sensibilité des données, l'implémentation d'un système de sécurité peut s'avérer utile. Finalement, une documentation détaillée de l'API est essentielle pour faciliter sa compréhension et son utilisation. L'outil Swagger, illustré par la figure 5.1, permet de documenter l'API en utilisant la spécification OpenAPI.

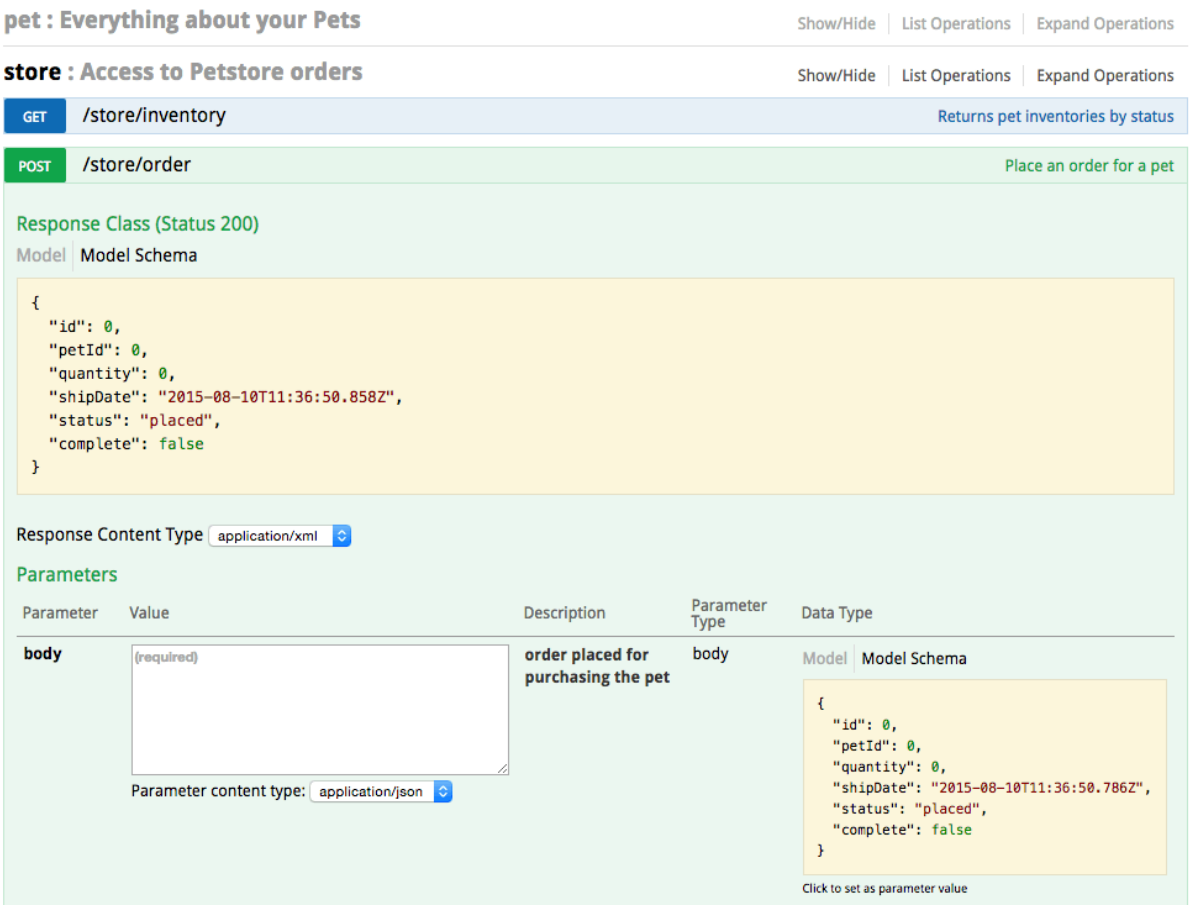

FIGURE 5.1 – Documentation swagger - [95]

La figure 5.1 illustre deux endpoints et les verbes HTTP associés. Swagger offre la possibilité de documenter les endpoints en détail. La structure de la ressource et le type de réponse sont visibles sur la figure 5.1. Finalement, Swagger permet de tester l'API depuis la documentation.

GraphQL

L'exposition d'une API GraphQL se fait en mettant à disposition de l'utilisateur, à travers une documentation, les différents schémas, Queries et Mutations. La documentation est par défaut prise en charge par GraphQL via l'outil GraphiQL. Cet outil, illustré par la figure 5.2, génère automatiquement la documentation en utilisant le principe d'introspection de GraphQL. L'introspection permet d'obtenir des détails sur tous les éléments disponibles dans l'API si elles sont utilisables par un utilisateur ou un service externe à l'API. Par exemple, avec une API REST, il est possible de choisir les endpoints qui nécessitent une documentation. Néanmoins, il est tout à fait possible d'utiliser un endpoint qui n'est pas documenté. De ce fait, les endpoints "sensibles" ne sont pas affichés publiquement. Avec une API GraphQL, cela n'est pas possible. Tous les schémas, Queries et Mutations pouvant être utilisés via l'API sont automatiquement documentés. Par conséquent, l'utilisateur peut connaître précisément le type de chaque attribut, les relations entre les schémas ou les différentes réponses disponibles [60][46].

L'outil GraphiQL, illustré par la figure 5.2, est constitué de trois parties. La partie de gauche contient la documentation générée grâce à l'introspection. La partie du milieu est un éditeur permettant de tester les différentes Queries et Mutations. Finalement, la partie de droite affiche la réponse des diverses requêtes.

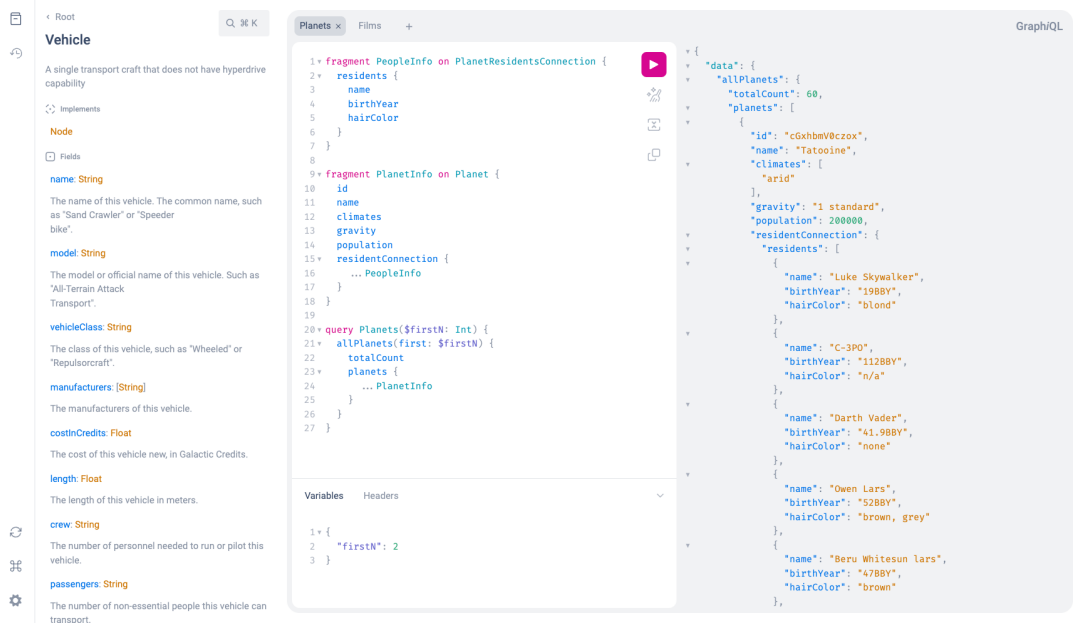

FIGURE 5.2 – Documentation GraphQL - [61]

La figure 5.3 montre une Query permettant l'introspection. La figure 5.4 présente la réponse de cette Query.

```
query IntrospectionQuery {
  __schema {
    types {
      name
      description
      kind
      fields {
        name
        description
        type {
          name
          description
          kind
        }
      }
    }
  }
}
```

FIGURE 5.3 – Query permettant l'introspection

```
{
  "data": {
    "__schema": {
      "types": [
        {
          "name": "Query",
          "description": "All queries",
          "kind": "OBJECT",
          "fields": [
            {
              "name": "user",
              "description": "A single user",
              "type": {
                "name": "User",
                "description": "This represents a user",
                "kind": "OBJECT"
              }
            }
          ]
        }
      ]
    }
  }
}
```

FIGURE 5.4 – Réponse de la Query 5.3

L'introspection facilite la compréhension et l'utilisation de l'API. Cependant, cette fonctionnalité amène quelques problèmes, notamment de sécurité. Par exemple, les descriptions des Queries et Mutations sont visibles via l'introspection. Si ces données contiennent des informations sensibles, comme des clés de chiffrement, alors l'API est entièrement exposée à des problèmes de sécurité. De même, certaines Queries ou Mutations retournant des données sensibles ne doivent pas être présentes dans la documentation. Il est généralement conseillé de désactiver l'introspection lors de l'exposition d'une API GraphQL [68].

Une façon plus sécurisée pour exposer une API GraphQL est de la rendre disponible sous forme d'API REST. Cette approche permet d'associer les Queries et Mutations à des endpoints REST, ce qui évite la documentation des éléments sensibles. Ainsi, le serveur peut traiter les différentes requêtes de manière appropriée.

gRPC

L'exposition des API gRPC est généralement très complexe pour plusieurs raisons. Premièrement, les clients et les serveurs gRPC sont générés à partir du fichier protoBuffer. Les différents composants ont une structure commune et connaissent les données et procédures déclarées. Comme les données sont transmises via le fichier protoBuffer, il n'est pas nécessaire d'exposer l'API aux utilisateurs ou services externes. La figure 5.5 illustre

la communication entre un client en Java et un serveur en Go. Les requêtes et les réponses transitent via le fichier protoBuffer.

La deuxième raison est liée au protocole HTTP/2. Les données échangées entre le client et le serveur sont en format binaire. Cette structure est illisible par l'homme et incompatible avec des clients ou serveurs non gRPC. Par exemple, l'outil Postman, utilisé pour tester les API, possède un module indépendant spécialement dédié aux API gRPC. Par conséquent, exposer une API gRPC ne permettra pas de l'utiliser avec des services externes et il est souvent nécessaire de développer à nouveau le service avec gRPC.

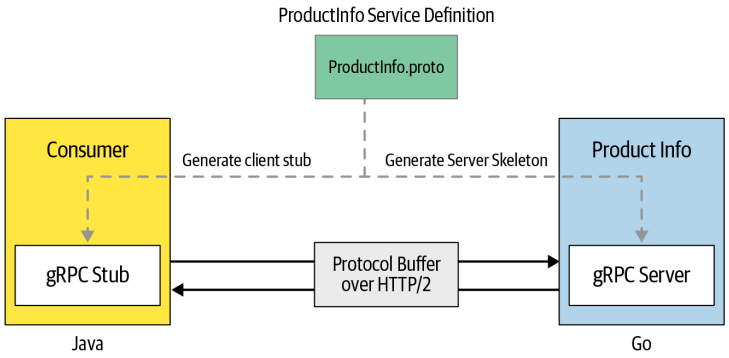

FIGURE 5.5 – Exposition des fonctions entre un client et un serveur GPRC - [33]

Dans certaines situations, la conversion d'un service en gRPC n'est pas une solution envisageable. Par exemple, lorsque les développeurs souhaitent élargir les fonctionnalités d'un client REST API grâce à un service gRPC, des problèmes de compatibilité surviennent. La résolution de ce problème nécessite la migration de tous les services vers gRPC, ce qui n'est pas réalisable. Pour résoudre ce type de situation, un reverse proxy peut être utilisé. La figure 5.6 présente son fonctionnement.

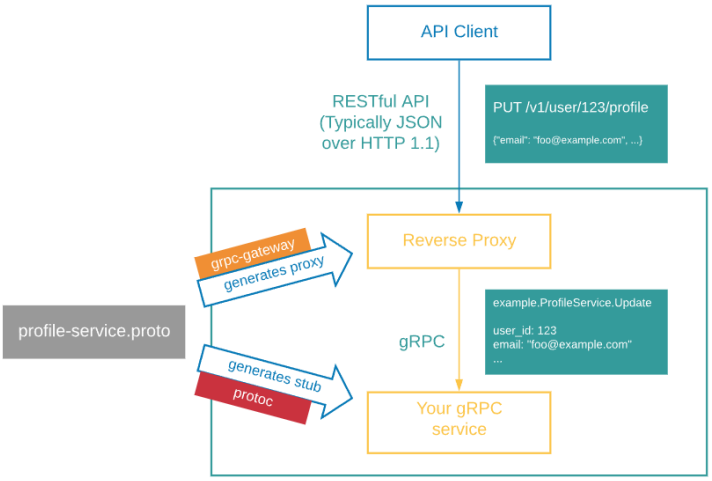

FIGURE 5.6 – Reverse Proxy gRPC - [48]

Le reverse proxy joue un rôle d'intermédiaire entre le client qui utilise les API REST et le service gRPC. Il se charge de la conversion des données du format binaire au format JSON et vice-versa. Cette fonctionnalité permet de faciliter la communication entre les services gRPC et ceux qui n'utilisent pas ce protocole. En outre, cet outil permet de simplifier l'exposition des API gRPC en les présentant comme des API REST.

5.3 Comparaison des API selon les critères 3 à 10

Le tableau 5.1 compare les technologies REST, GraphQL et gRPC selon les critères 3 à 10 [30].

Critère	API	Explications
Complexité de développement	REST	**Facile** Le développement d'une API REST est simple et intuitif. De nombreux outils sont disponibles pour faciliter son développement, que ce soit pour structurer l'API, la tester ou générer automatiquement du code [36]. Dans certaines situations, le développement d'une API REST peut être complexe, notamment si elle utilise le principe HATEOAS [89].
	GraphQL	**Difficile** GraphQL est une technologie récente. Il existe peu d'outils pour structurer et tester l'API. De manière générale, les développeurs doivent écrire du code pour définir les ressources, les schémas et les relations. Cet aspect est peu pratique, particulièrement dans les premières phases, car la structure de l'API n'est pas encore définie [30].
	gRPC	**Moyen** gRPC est une technologie récente et peu d'outils sont disponibles pour simplifier son développement. De plus, le protocole RPC peut être complexe à mettre en place et à utiliser. Cependant, le fichier protoBuffer et le système de compilation vers divers langages de programmation offre une structure à l'API, ce qui facilite en partie son développement [30].
Complexité de développement d'un client pour l'API	REST	**Facile-Moyen** La difficulté liée au développement d'un client utilisant l'API REST est fortement influencé par la documentation de celle-ci. Si la documentation est peu intuitive, la prise en main de l'API peut être complexe [69]. Toutefois, si HATEOAS est mis en place, il peut aider à surmonter cette difficulté en facilitant la découverte et la compréhension de l'API.
	GraphQL	**Facile** L'introspection de GraphQL permet de parcourir les différents schémas, Queries et Mutations, ce qui simplifie la compréhension et l'utilisation de l'API [19].

	gRPC	**Difficile** Le développement du client ne peut se faire qu'une fois le fichier protoBuffer défini et le client compilé dans le langage de programmation souhaité. gRPC ne permet pas d'exposer les diverses fonctions à travers une API. Ce point complique le développement du client et la compréhension de l'API par les développeurs frontend [30].
Mise en place d'outils de redirection	REST	**Facile** Les outils de redirection tels que les gateways ou middlewares supportent nativement l'architecture REST. De ce fait, il est aisé de les intégrer [30].
	GraphQL	**Moyen** La majorité des outils disponibles ne supportent pas GraphQL par défaut. Cependant, certaines fonctionnalités sont compatibles avec GraphQL car la communication se fait via le protocole HTTP. La mise en place d'un tel outil est complexe dûe à la présence d'un seul endpoint [30].
	gRPC	**Difficile** La plupart des outils sont incompatibles avec le protocole HTTP/2 et la structure protoBuffer. De ce fait, il est généralement nécessaire de développer une solution spécifique à l'API [30].
Composition de l'API	REST	**Oui** REST favorise la composition des API pour fournir des services plus complexes. Il est également courant de combiner des ressources et de les rendre disponibles à travers un endpoint. L'aspect négatif de la combinaison des API REST est le nombre de requêtes effectuées [81]. Si les fonctionnalités ou les ressources sont séparées dans plusieurs services, au minimum une requête par service est nécessaire.
	GraphQL	**Oui** GraphQL utilise le principe de Graphe. Il est commun et simple de combiner différentes structures. De plus, GraphQL possède des librairies simplifiant la combinaison des services en un seul. Par conséquent, cela réduit le nombre de requêtes et augmente les performances de l'API [84].
	gRPC	**Oui** La composition des services gRPC est possible grâce au fichier protoBuffer. Plusieurs services gRPC distincts peuvent ainsi communiquer entre eux, peu importe le langage de programmation [38].

Authentification et autorisation	REST	**Facile** Les standards d'authentification comme OAuth et OpenID sont compatibles avec REST. Leur mise en place est par conséquent simple. De plus, les règles concernant les accès peuvent être définies indépendamment pour chaque endpoint [30].
	GraphQL	**Moyen** Les standards d'authentification comme OAuth et OpenID sont compatibles avec GraphQL grâce à l'utilisation du protocole HTTP. Cependant, la présence d'un seul endpoint complexifie la mise en place des règles d'accès [30].
	gRPC	**Difficile** OpenID et OAuth sont compatibles avec gRPC. Il est en général nécessaire de développer une couche de protection supplémentaire dûe à l'architecture de gRPC [30].
Caching	REST	**Facile** Le caching fait partie des contraintes de développement de REST. Par conséquent, REST profite de cet avantage. La complexité liée à la mise en place du caching dépend du service et des exigences de l'API. Par exemple, si les réponses de l'API ne changent pas fréquemment, la mise en place d'un caching est simple. Cependant, lorsque une API possède un système d'authentification, sa mise en place peut être complexe car le caching doit être géré indépendamment pour chaque utilisateur [39].
	GraphQL	**Moyen** Le caching n'est pas supporté par défaut avec GraphQL. La présence d'un seul endpoint complexifie la mise en place de ce système. Deux façons permettent d'implémenter le caching avec GraphQL: l'utilisation d'outils externes ou l'exposition de l'API via REST [41].
	gRPC	**Difficile** Le caching n'est pas supporté par défaut par gRPC et les outils externes permettant sa mise en place sont peu existants. La documentation de gRPC ne fournit pas d'informations à ce sujet.

Gestion des versions	REST	**Moyen** De nombreux standards existent pour gérer les versions de REST. Par exemple, il est possible de maintenir différentes versions via les URI ou via les paramètres. La présence de ces nombreux standards complexifie le choix de la bonne méthode [85].
	GraphQL	**Facile** Les bonnes pratiques concernant la gestion des versions sont disponibles dans la documentation officielle. Les différentes versions sont également réversibles, ce qui facilite l'évolution et la maintenance de l'API [42].
	gRPC	**Difficile** Les clients et les serveurs gRPC sont générés à partir du fichier ProtoBuffer et sont réversibles. Plusieurs versions du fichier ProtoBuf peuvent exister en simultané, ce qui complique la gestion des versions [30].
Mode de communication	REST	**1 mode de communication** REST ne supporte que la communication unidirectionnelle par défaut. La communication bidirectionnelle peut être implémentée via un webSocket [29].
	GraphQL	**2 modes de communication** GraphQL supporte nativement la communication unidirectionnelle et le streaming client (via les Subscriptions) [19].
	gRPC	**4 modes de communication.** gRPC supporte nativement les communications unidirectionnelles, streaming client, streaming serveur et bidirectionnelles [5].

Table 5.1 – Comparaison des API REST, GraphQL, et gRPC

5.4 Analyse de la performance: temps de réponse des API

La performance est l'un des principaux points de décision lors du choix de la technologie pour l'API. Il existe actuellement plusieurs recherches littéraires comparant REST et GraphQL ou REST et gRPC. Cependant, les articles comparant les trois technologies sont peu fréquents. Cette section présente les recherches effectuées par différentes universités. Ces recherches se portent principalement sur l'étude du temps de réponse des API. Dans le chapitre 7, nous effectuerons nos propres tests de performance pour comparer REST et GraphQL.

Plusieurs recherches universitaires comparent REST et GraphQL sur leur temps de réponse. De manière générale, les résultats ne montrent pas de différence significative entre les deux API. En effet, selon la méthodologie et les outils utilisés, le temps de réponse varie en faveur de l'une ou de l'autre technologie. Par exemple, l'étude effectuée par l'Université d'Hasanundin, illustrée par la figure 5.7, montre un temps de réponse plus faible et une meilleure stabilité pour l'API REST comparée à l'API GraphQL [12]. L'étude a été menée sur un des systèmes d'information de l'université. D'autres recherches comme celles de l'université de Carleton [28], présentées par la figure 5.8, ne permettent pas de départager les deux API. Les chercheurs expliquent que le temps de réponse entre les deux technologies varie en fonction de la requête effectuée. Les mesures ont été réalisées cinquante fois sur quatre requêtes différentes, allant de la requête la plus simple à la requête la plus complexe.

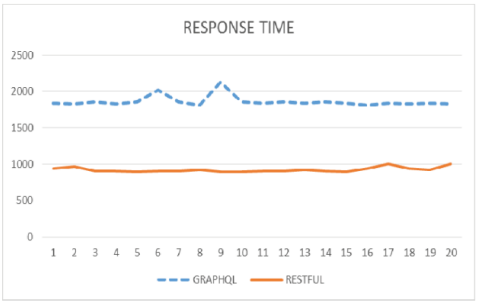

FIGURE 5.7 – Temps de réponse de REST et GraphQL [12]

Query	REST	GraphQL
Query 1	171.16	176.96
Query 2	627.00	606.34
Query 3	225.44	144.88
Query 4	338.16	388.46

FIGURE 5.8 – Temps de réponse moyen entre REST et GraphQL - [28]

Le même cas de figure se présente lors des comparaisons de REST et gRPC. L'University of Technology de Rzeszów [3] a mesuré le temps de réponse entre REST et gRPC sur des requêtes chiffrées (HTTPS) et non chiffrées (HTTP). Les résultats montrent un avantage pour REST lorsque aucun chiffrement n'est utilisé mais une similitude sur le temps de réponse lorsque les requêtes sont chiffrées. La figure 5.9 illustre les résultats de cette étude.

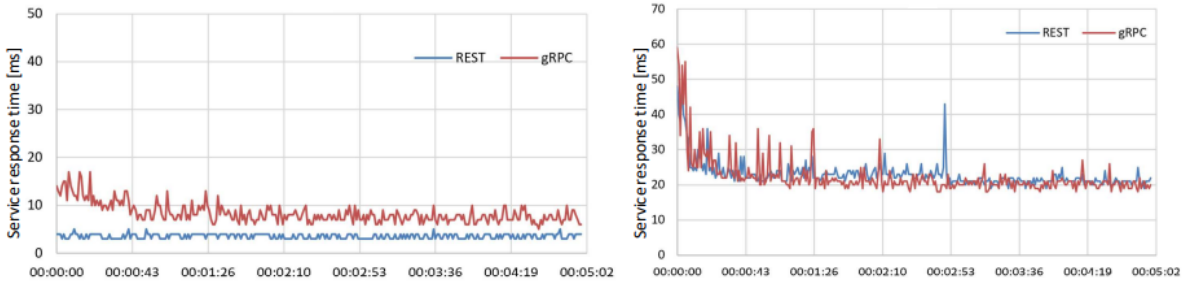

FIGURE 5.9 – Temps de réponse moyen entre REST et gRPC sans chiffrement (HTTP) et avec chiffrement (HTTPS) - [3]

Deux recherches scientifiques comparent les trois technologies en même temps. Les recherches de Lublin University of Technology [32] présentent REST comme le meilleur choix possible dans la plupart des situations. Les auteurs précisent que gRPC est une bonne alternative lorsque les données ne sont pas volumineuses. En effet, le papier *Comparative review of selected Internet communication protocols* [13] va également dans ce sens et place gRPC comme technologie performante pour le transfert des données. La

figure 5.10 présente les résultats obtenus lors de la récupération de 100 éléments depuis la base de données. gRPC montre des temps de réponse fortement inférieurs aux autres technologies. Concernant GraphQL, celui-ci se démarque lorsque les requêtes récupèrent des données imbriquées. Finalement, REST est le plus performant dans la plupart des situations car il n'a pas été développé pour répondre à un besoin spécifique.

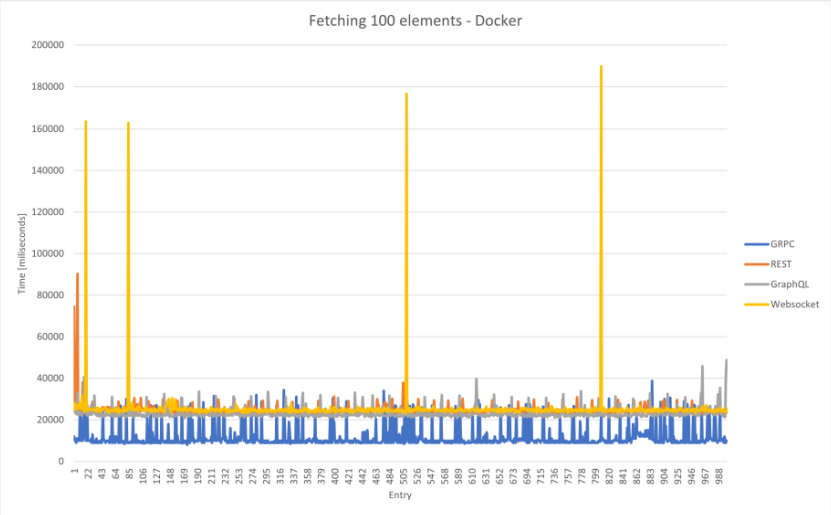

FIGURE 5.10 – Temps de réponse pour cent éléments [13]

Ces différentes recherches permettent de conclure qu'il n'y a pas de technologie parfaite capable de performer dans toutes les situations. gRPC et GraphQL ont été développés pour des besoins spécifiques. gRPC facilite la communication entre différents services et GraphQL a permis de résoudre les problèmes de performance de REST, notamment pour la récupération de données volumineuses. Par conséquent, ces deux technologies excellent dans leurs domaines respectifs. REST, quant à lui, est satisfaisant pour l'utilisation générale d'une API.

A travers ce chapitre, les technologies REST, GraphQL et gRPC ont été comparées en se basant sur la littérature. Dans le chapitre suivant, le prototype de l'application web Stock&Co est présenté.

6
Présentation du prototype

6.1 Architecture du serveur 52
6.1.1 Gateway 53
6.1.2 Workflow 55
6.1.3 Stock&Co et le processus de paiement 57
6.2 Le client 59

Ce chapitre présente l'application qui a été développée pour l'entreprise Stock&Co. Le prototype a mis en application les différentes notions apprises dans les chapitres précédents. L'architecture, les technologies utilisées et leurs fonctionnements y sont discutées. L'application est composée de deux parties. La partie serveur a été développée en Javascript avec le framework Express. La partie client utilise la librairie ReactJS.

6.1 Architecture du serveur

Pour répondre à la demande de Stock&Co, l'architecture en microservices a été choisie pour le développement de leur application. Stock&Co est une petite-moyenne entreprise qui prévoit une forte croissance dans les années à venir. Les services doivent continuellement être mis à jour pour répondre à la demande et ces changements ne doivent pas impacter les autres services. Le nombre de produits vendus par l'entreprise va également augmenter au fil des années. L'architecture en microservice répond à tous ces besoins.

L'application est composée de trois microservices: un pour gérer les utilisateurs, un pour les commandes et un pour les produits. Chaque microservice est indépendant, a sa propre base de données et son propre API. La technologie utilisée pour l'API est GraphQL. Stock&Co s'intéresse aux relations entre les ressources (commandes précédentes, produits les plus vendus, etc) afin de mieux conseiller ses clients. De plus, comme l'entreprise grandit, il est judicieux de choisir une technologie qui peut supporter la transmission de grandes quantités de données. Finalement, l'entreprise prévoit également le développement d'une application mobile. Toutes ces raisons font de GraphQL la technologie la plus adaptée pour leur API.

L'architecture de Stock&Co inclut deux autres services comme présentés sur la figure 6.1: un workflow en orchestration et un gateway pour l'API. Leur utilité et fonctionnement sont définis dans les sections suivantes.

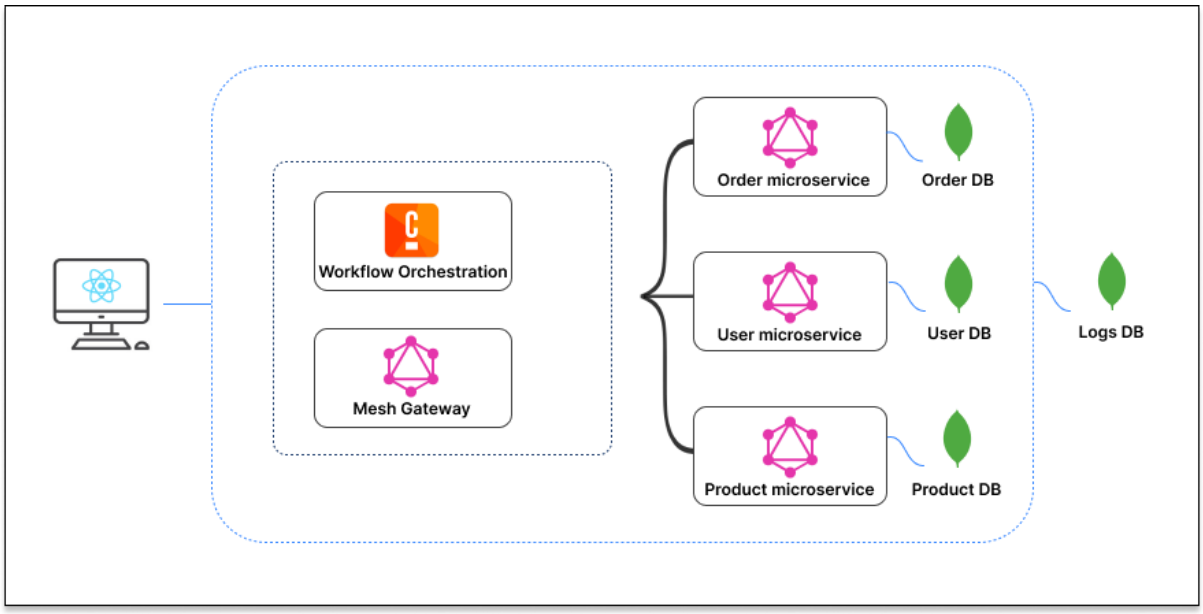

FIGURE 6.1 – Architecture de Stock&Co

6.1.1 Gateway

L'application web de Stock&Co est constituée de trois microservices indépendants avec chacun leur propre API. Cette séparation complexifie l'utilisation par le client car différents endpoints existent. Bien que GraphQL ne possède qu'un seul endpoint par API, le client doit utiliser trois endpoints différents et connaître, pour chaque requête, où se trouvent les ressources nécessaires. De plus, la plupart des ressources partagent des caractéristiques communes. Par exemple, une commande est toujours liée à un utilisateur. Par conséquent, il est essentiel d'avoir un outil permettant à la fois de récupérer des ressources depuis plusieurs microservices et de combiner ces ressources. En d'autres termes, l'outil doit permettre de composer une API à partir des trois microservices. Le Gateway Mesh, illustré sur la figure 6.1, joue ce rôle, en redirigeant les requêtes vers la bonne API et en permettant également la combinaison des ressources.

La figure 6.2 illustre la définition de Mesh. L'API **utilisateur** et l'API **commande** sont déclarées avec leur endpoint respectif. Cette déclaration permet à Mesh de récupérer toutes les informations (schéma, Queries, Mutations, etc) disponibles.

```
sources:
  - name: Users
    handler:
      graphql:
        endpoint: http://user-microservice:8082/graphql
  - name: Orders
    handler:
      graphql:
        endpoint: http://order-microservice:8083/graphql
```

FIGURE 6.2 – Définitions des endpoints dans Mesh

À partir de là, il est possible de créer de nouvelles fonctionnalités et d'étendre celles déjà existantes. La figure 6.3 illustre l'extension du schéma **User** en lui rajoutant un attribut **orders**.

```
additionalTypeDefs: |
  extend type User {
    orders: [Order!]!
  }
```

FIGURE 6.3 – Extension du schéma utilisateur

L'extension du schéma n'est pas suffisante car Mesh ne connaît pas les relations qui lient deux ressources. La prochaine étape est la définition de ces relations, illustrée par la figure 6.4. Pour obtenir la liste des commandes associées à un utilisateur dans Mesh, il est nécessaire d'appeler la Query **orderByUser** disponible dans le microservice **Orders** en fournissant l'**id** de l'utilisateur comme argument.

```
  - targetTypeName: User
    targetFieldName: orders
    sourceName: "Orders"
    sourceTypeName: "Query"
    sourceFieldName: "orderByUser"
    requiredSelectionSet: "{_id}"
    sourceArgs:
      userid: "{root._id}"
```

FIGURE 6.4 – Liaison entre le microservice utilisateur et le microservice commande

Finalement, la Query de la figure 6.5 retourne la liste des commandes associées à chaque utilisateur.

```
query {
  users {
    name
    email
    orders {
      _id
      createdAt
    }
  }
}
```

FIGURE 6.5 – Query combiné avec Mesh

La figure 6.6 présente la réponse de cette Query.

```
{
  "data": {
    "users": [
      {
        "name": "Dimitri",
        "email": "dimitri@boscova.xyz",
        "orders": [
          {
            "_id": "1",
            "create_at": "2023-01-04"
          }
        ]
      },
      {
        "name": "Gustave",
        "email": "gustave@flobinou.xyz",
        "orders": []
      }
    ]
  }
}
```

FIGURE 6.6 – Réponse de la figure 6.5

La Query est similaire à celle présentée par la figure 4.9. Lorsque les schémas sont définis dans le même service, ce type de query est nativement supporté. Cependant, lorsque les schémas sont séparés dans plusieurs services, l'utilisation d'une librairie comme Mesh est indispensable pour lier les ressources et créer des schémas, Queries et Mutations plus complexes.

6.1.2 Workflow

L'entreprise Stock&Co possède trois microservices qui doivent être coordonnés pour fournir des services. Les chapitres 3.5 et 3.6 présentaient les différentes manières existantes pour organiser la communication entre les services. Dans le cas de Stock&Co, l'entreprise souhaite visualiser l'avancée de ses processus. Par conséquent, le choix s'est porté sur un workflow en orchestration, appelé Camunda.

Liaison du workflow avec les microservices

Pour utiliser le workflow Camunda, il est nécessaire de modéliser les différentes étapes du processus. La modélisation se fait grâce à l'outil **Camunda modeler**. Le langage utilisé est BPMN. La figure 6.7 montre l'interface de Camunda Modeler.

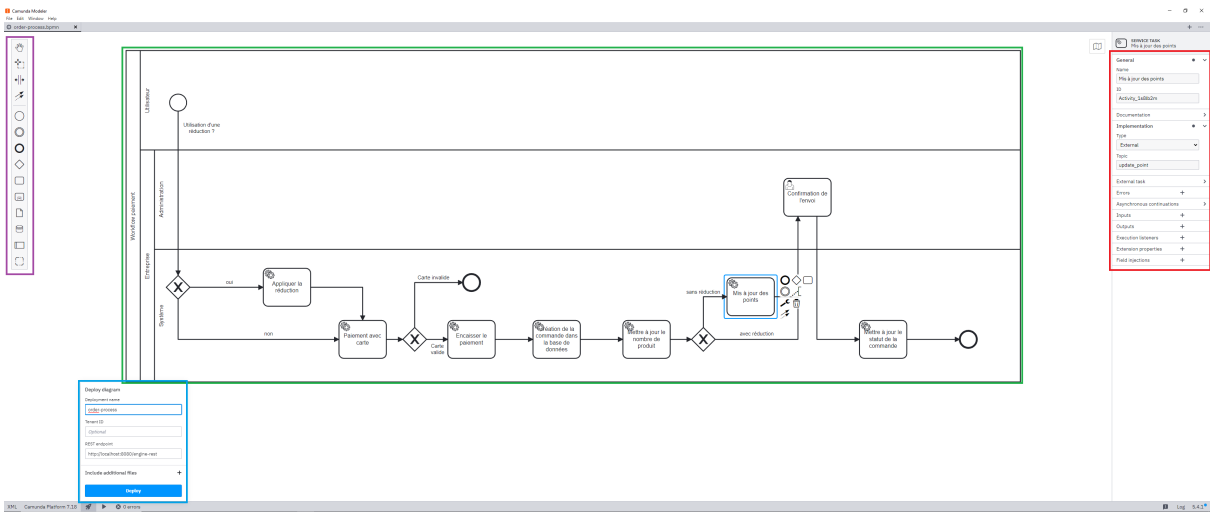

FIGURE 6.7 – L'interface de camunda modeler

Cette interface est composée de plusieurs éléments. La partie au centre (en vert) est l'éditeur permettant de modéliser le processus. La partie à gauche (en violet) contient les différents composants du langage BPMN. Ce langage est complet et possède de nombreux composants. Seuls les composants suivants ont été utilisés:

- **Cercle au bord noir simple**. Il indique le début d'un processus.
- **Cercle au bord noir épais**. Il indique la fin d'un processus.
- **Losange avec une croix**. Il s'agit d'un point de décision.
- **Rectangle au bord arrondi avec l'icône boulon**. Il s'agit d'une tâche liée à un service. Camunda déclenche le service responsable de cette tâche.
- **Rectangle avec l'icône bonhomme**. Il s'agit d'une tâche manuelle. Camunda attend une action de l'utilisateur.

La partie à droite (en rouge, présentée également sur la figure 6.8) devient visible lorsqu'un élément du modèle est sélectionné. Dans cette partie, les composants peuvent être liés à des Topics, permettant à Camunda de les identifier. Ces Topics sont ensuite utilisés pour faire le lien entre l'élément du modèle et les différents microservices. Par exemple, la figure 6.8 montre que la tâche de service **Mis à jour des points** est liée au Topic **update_point**. Le code de la figure 6.9 implémenté dans le microservice **utilisateur** sera déclenché par Camunda lorsque le processus arrivera à cette étape.

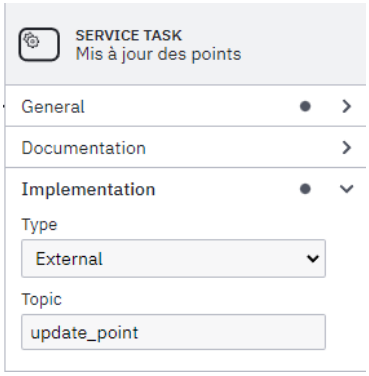

FIGURE 6.8 – Paramètres pour la tâche "Mise à jour des points"

```
client.subscribe("update_point", async function ({ task, taskService }) {
  try {
    const order = task.variables.get("order");
    const oneUser = await User.findOne({ _id: order.userid });
    if (oneUser) {
      const newPoints = Math.floor(order.total / 100);
      oneUser.points += newPoints;
      await oneUser.save();
    }
    await taskService.complete(task);
  } catch (err) {
    console.log("error", `Update points | ${err}`);
  }
});
```

FIGURE 6.9 – Topic update_point

Finalement, la partie en bleu de la figure 6.7 permet de déployer le modèle sur l'outil Camunda. Le déploiement se fait via une API REST. Le modèle est ensuite visible sur l'outil et peut être démarré grâce à un endpoint. La figure 6.10 montre l'interface de Camunda et le processus en cours. La progression peut être visualisée via les pastilles bleues. Par exemple, 32 instances attendent la confirmation de l'envoi.

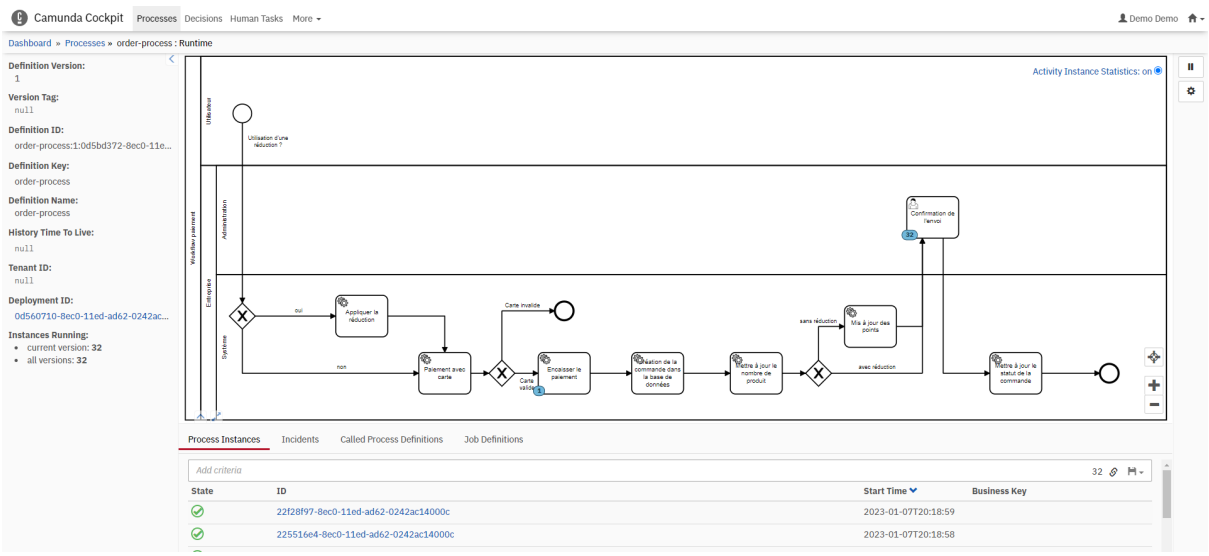

FIGURE 6.10 – Interface de Camunda

6.1.3 Stock&Co et le processus de paiement

Pour la définition des processus de l'application Stock&Co, seules les étapes depuis le paiement jusqu'à l'envoi de la commande ont été modélisées, comme illustré par la figure 6.11. Concernant les tâches effectuées par l'utilisateur, ainsi que les étapes de livraison (poste, emplacement du colis,etc), celles-ci ne sont pas représentées car elles ne sont pas le sujet de cette thèse. De plus, Camunda ne gère pas les tâches qui ne peuvent pas

être reliées à un Topic et qui ne peuvent pas être gérées depuis la partie serveur de l'application. Par exemple, les actions de l'utilisateur sur le client ne sont pas envoyées au serveur et ne sont, par conséquent, pas utilisables par Camunda.

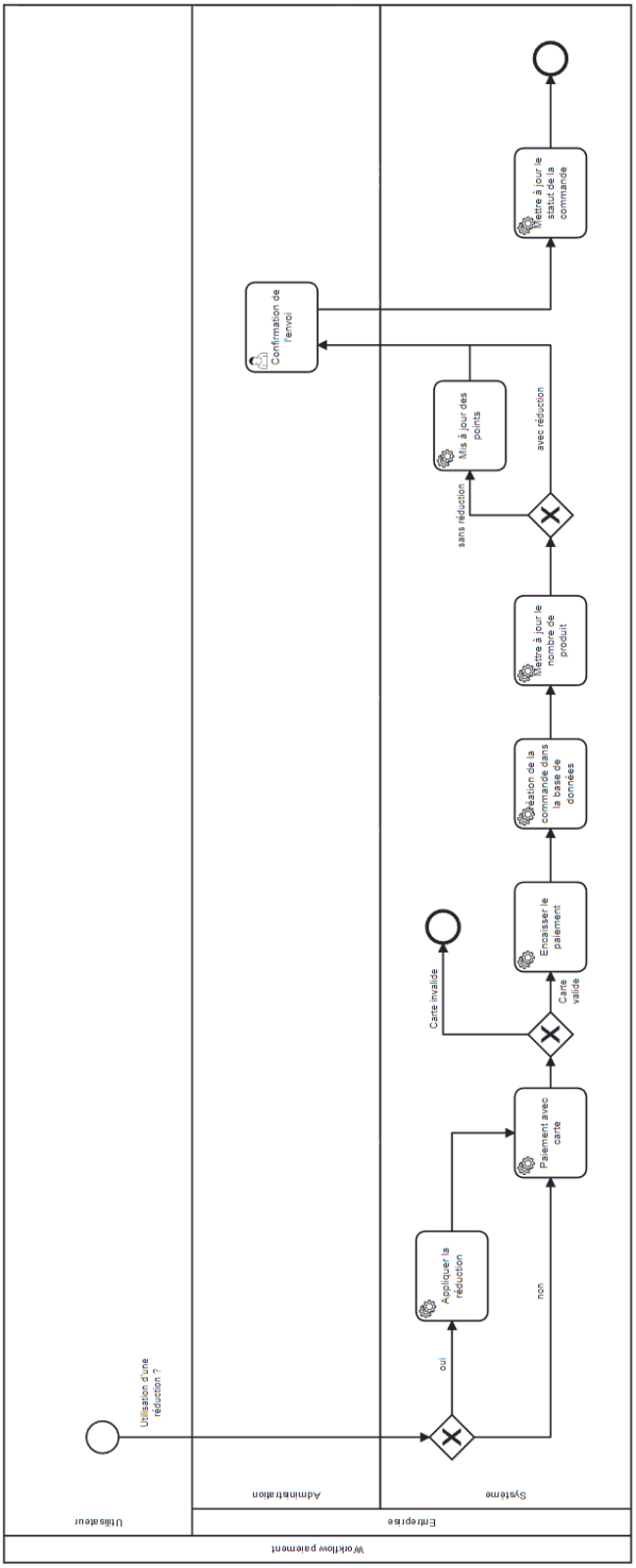

FIGURE 6.11 – Workflow paiement

Stock&Co a mis en place un système de récompense pour fidéliser ses clients. Pour chaque tranche de 100 CHF dépensée, l'utilisateur reçoit un point. Ces points peuvent ensuite être utilisés comme bon de réduction.

Le processus de la figure 6.11 fonctionne de la manière suivante:

1. L'utilisateur confirme la commande.
2. Camunda démarre le processus.
3. Le processus vérifie si des points ont été utilisés.
   a) Si oui, une réduction est appliquée sur le montant total. Le processus passe ensuite à l'étape suivante.
   b) Si non, aucune réduction n'est appliquée et le processus passe directement à l'étape suivante.
4. Le processus vérifie si la carte de paiement est valide.
   a) Si oui, le processus passe à l'étape suivante.
   b) Si non, le processus se termine et informe l'utilisateur.
5. Le paiement est encaissé. Si une réduction a été utilisée, les points consommés sont retirés.
6. Camunda informe **le microservice commande** qu'une commande a été passée. La commande est créée dans la base de données.
7. Camunda informe **le microservice produit** qu'une commande a été effectuée. Le microservice produit reçoit la liste des produits et la quantité achetée. Il utilise ces informations pour mettre à jour sa base de données.
8. Si aucune réduction n'a été utilisée, les points de l'utilisateur sont mis à jour. Si ce n'est pas le cas, Camunda passe à l'étape suivante.
9. L'entreprise reçoit une notification l'informant qu'une commande a été effectuée et que le paiement a bien été encaissé. Le processus est en attente. L'entreprise prépare la commande. Elle valide ensuite son envoi. Le processus reprend et passe à l'étape suivante.
10. Camunda informe **le microservice commande** que la commande a été envoyée. Le microservice met à jour le statut de la commande.
11. Le processus se termine.

6.2 Le client

L'API possède de nombreuses Queries et Mutations et il est compliqué d'y simuler un scénario. Pour faciliter la tâche, un client a été développé. Une partie de l'interface est utilisable sans que l'utilisateur soit authentifié. Cependant, le processus de paiement, lié à Camunda, n'est disponible que pour les utilisateurs authentifiés. L'authentification est faite via un **JSON Web Token**.

L'entreprise Stock&Co vend des produits informatiques. La page principale du client, illustrée par la figure 6.12, affiche la liste des produits disponibles. Plusieurs informations sont présentes: l'image du produit, la description, le prix, la note des clients et la disponibilité. L'utilisateur peut cliquer sur un produit et sera redirigé sur une page spécifique au produit.

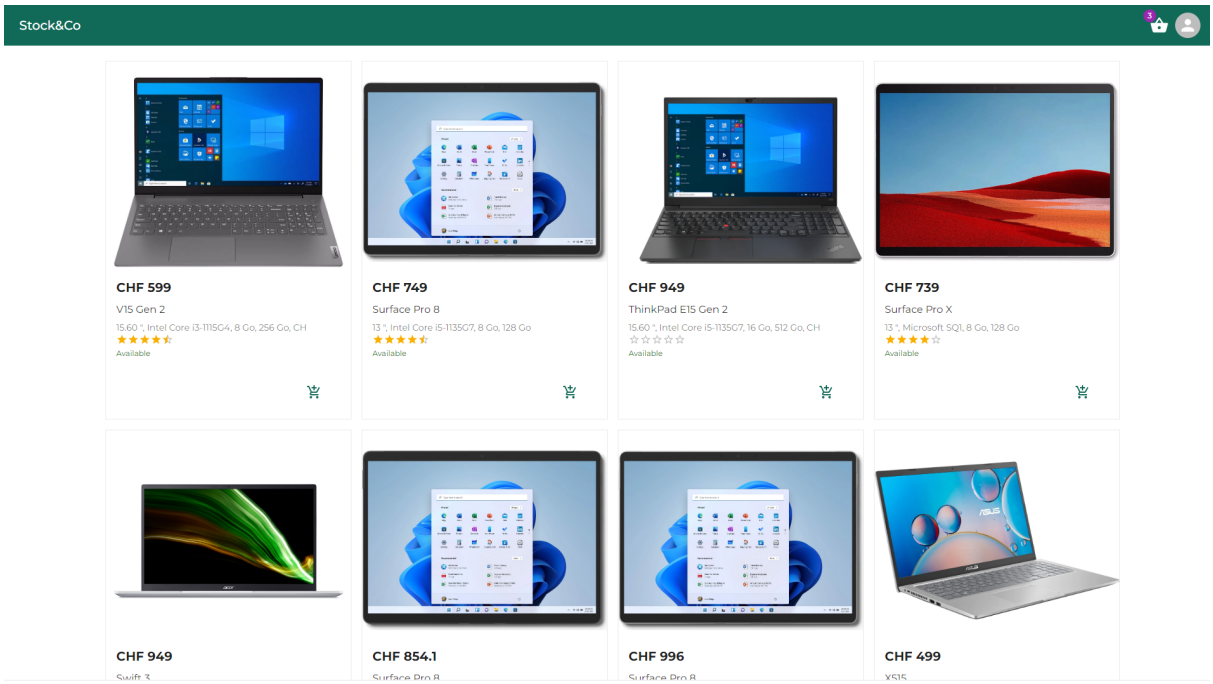

FIGURE 6.12 – Interface du client

La figure 6.13, présente la page du produit. Elle contient, en plus des informations précédentes, les avis des clients. Comme expliqué précédemment, les informations sur les commandes, les produits et les utilisateurs sont stockées dans des microservices différents. Ici, toutes ces ressources proviennent de la même Query grâce à l'utilisation de Mesh.

L'utilisateur peut choisir la quantité souhaitée et peut ajouter le produit au panier. Le panier, en haut à droite, indique le nombre de produits distincts qu'il contient.

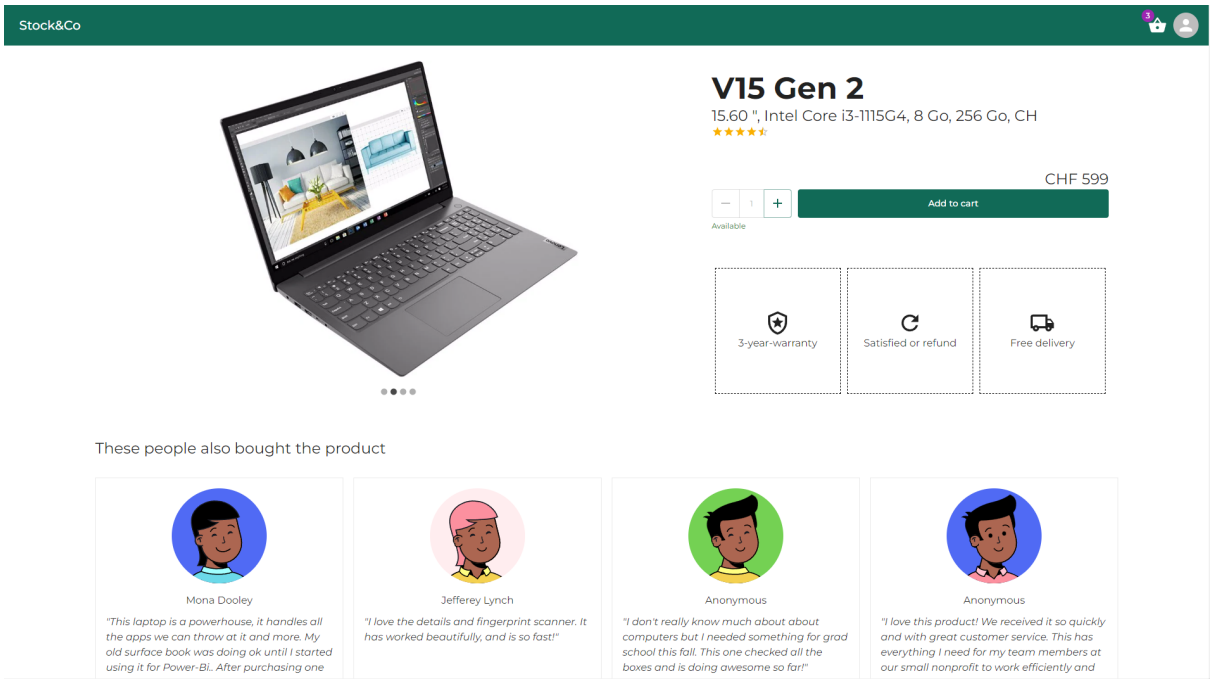

FIGURE 6.13 – Page d'un produit

En cliquant sur l'icône panier, l'utilisateur est dirigé vers une page qui résume ses achats. Sur la figure 6.14, une vue détaillée des produits ajoutés dans le panier est visible. Le prix total du panier est également indiqué de manière détaillée. Si l'utilisateur possède des points et souhaite les utiliser, il indique la valeur dans le champ de texte et clique sur le bouton **Apply**. La réduction est ainsi appliquée et est visible dans la vue détaillée des prix.

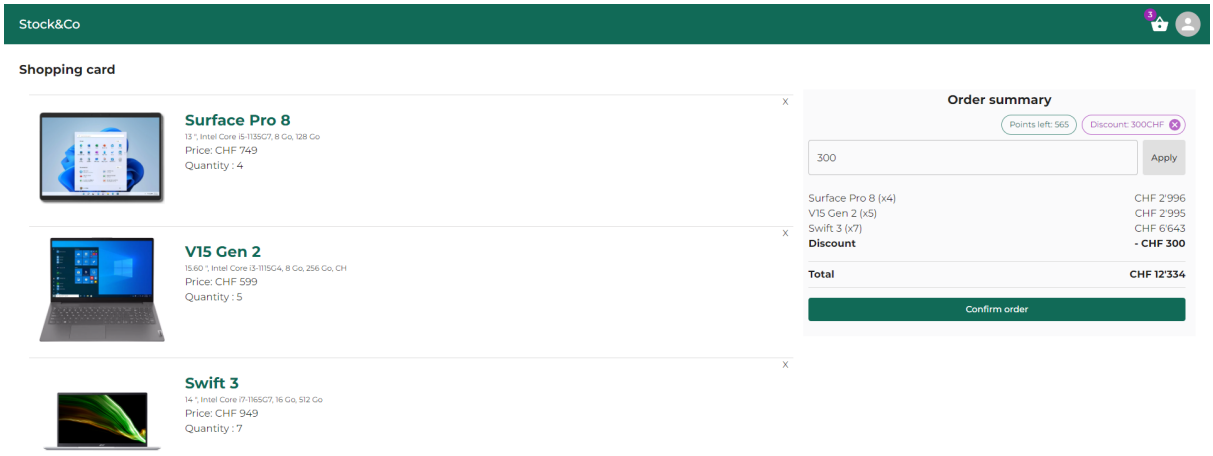

FIGURE 6.14 – Détails du panier

Tant que le paiement n'est pas confirmé, l'utilisateur peut modifier son panier et la réduction appliquée. Une fois la commande validée, il est redirigé vers une page pour entrer ses coordonnées bancaires. Cette page est présentée par la figure 6.15.

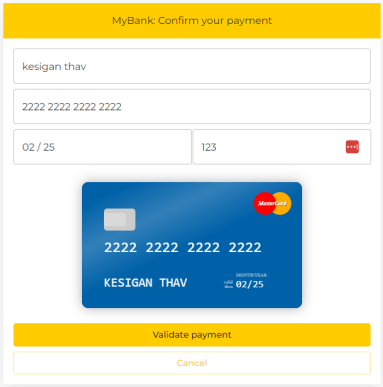

FIGURE 6.15 – Page de paiement

Le processus Camunda démarre lorsque l'utilisateur clique sur le bouton **Validate payment**. La redirection est effectuée vers le profil du client, illustré par la figure 6.16. Cette page contient des informations personnelles comme le nom, l'email et le nombre de points. La liste des commandes est également affichée et l'utilisateur peut voir la progression et le nombre de points gagnés par commande. Le statut de la commande, visible en bleu, est **Paid**.

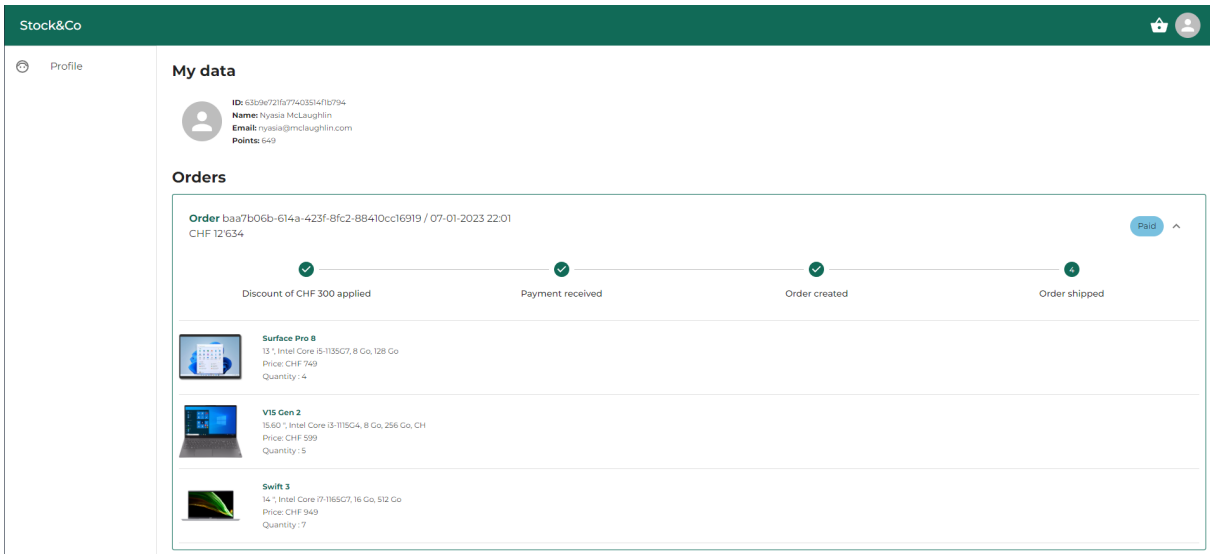

FIGURE 6.16 – Page de profil

Stock&Co est informé qu'une commande a été effectuée et qu'elle attend une confirmation de l'envoi. L'entreprise peut voir les détails de la commande depuis son profil. Une fois le colis prêt et envoyé, l'entreprise clique sur le bouton **Confirm the payment**. La figure 6.17 montre le profil de Stock&Co.

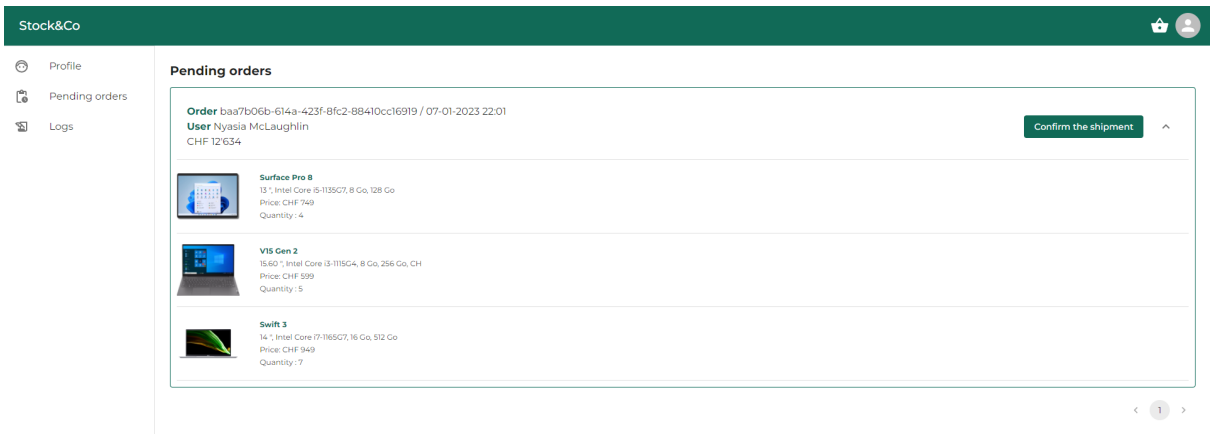

FIGURE 6.17 – Commande en attente

Le statut de la commande change chez l'utilisateur. La progression a également changé. La commande a été envoyée. Elle est dorénavant clôturée. La figure 6.18 affiche l'état de la commande après la confirmation de l'entreprise.

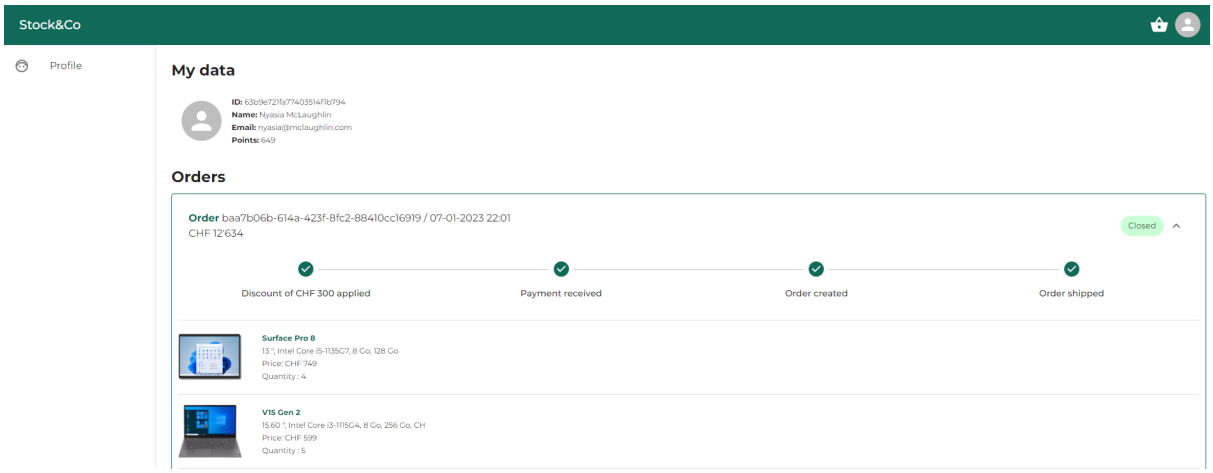

FIGURE 6.18 – Mis à jour du statut de la commande

La figure 6.19 montre l'historique généré par les processus. L'entreprise peut ainsi détecter les éventuelles erreurs.

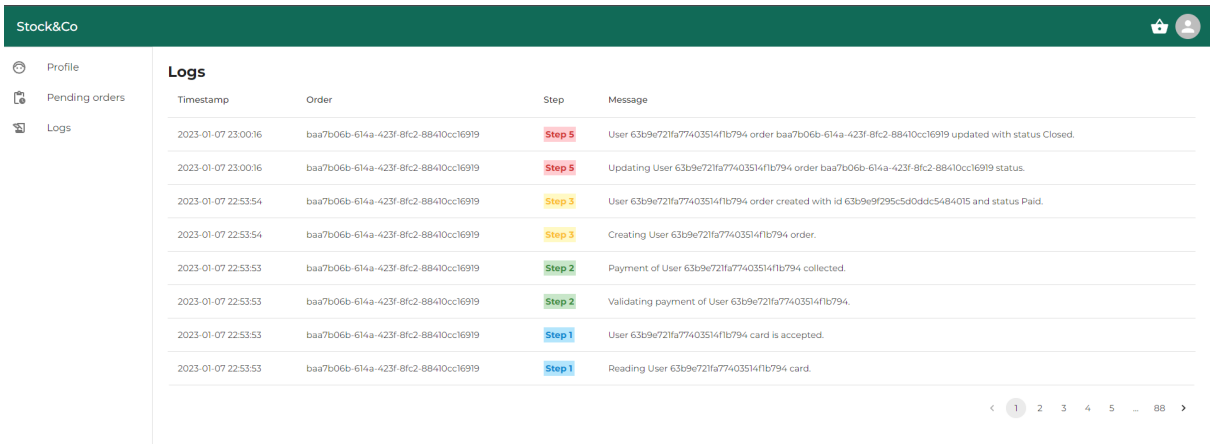

FIGURE 6.19 – Historique du workflow

Pour générer cette historique, la librairie Winston a été utilisée.

Dans ce chapitre, nous avons pu mettre en pratique les différentes notions présentées dans les chapitres précédents. Ce prototype permet d'avoir une application web efficiente permettant de suivre les processus d'achat de l'utilisateur, de mettre en avant les produits et leurs avis et d'assurer une expérience d'achat ergonomique à l'utilisateur. En effet, celui-ci peut à tout moment suivre ses commandes et leurs progressions. De plus, un système de bonus a été implémenté dans le cadre de la fidélisation de l'utilisateur. Tout cela a été possible grâce à l'utilisation de Camunda, GraphQL, du Gateway Mesh et l'architecture en microservices. Dans le chapitre suivant, des tests de performance sont effectués sur la partie serveur de l'application.

7

Evaluation pratique des performances

7.1 Introduction **64**
7.2 Méthodologie **65**
7.2.1 Types de test 65
7.2.2 Echantillons 66
7.2.3 Récapitulatif 66
7.2.4 Matériel 66
7.3 Test séquentiel **67**
7.4 Test de charge **69**
7.4.1 Autocannon 69
7.4.2 JMeter 71
7.5 Test concurrent **74**
7.6 Synthèse **78**

Ce chapitre est consacré à l'évaluation de la performance de REST et GraphQL. Les tests sont effectués sur le serveur de l'application Stock&Co.

7.1 Introduction

Les API de Stock&Co ont été développées avec GraphQL. Il est intéressant de tester les performances de cette API afin de les comparer à la théorie du chapitre 4 et aux résultats présentés dans le chapitre 5. Il est également intéressant de comparer certaines fonctions de notre API avec son équivalent en REST. Pour ce faire, trois fonctions de GraphQL ont été développées à nouveau et ont été exposées via une API REST.

La tableau 7.1 présente les trois fonctions exposées avec GraphQL et REST et les end-points pour y accéder.

Fonctions	Endpoint GraphQL	Endpoint REST
getUsers (GET)	http://localhost:8082/graphl query:users	http://localhost:8082/users
addProduct (POST)	http://localhost:8084/graphql mutation:addProduct	http://localhost:8084/product
products-users (GET)	http://localhost:10000/graphql query:getProductByUser	http://localhost:10000/products/users

TABLE 7.1 – Fonctions utilisées pour le test de performance

La fonction **product-users** retourne, pour chaque produit, ses détails et la liste détaillée des utilisateurs l'ayant acheté. Cette fonction est disponible dans GraphQL grâce à la librairie Mesh. Cependant, il n'est pas possible d'exposer la fonction de Mesh à travers REST. Pour assurer une équité impartiale entre les deux API, une nouvelle version de cette fonction a été développée pour GraphQL et pour REST. Aucun système de caching n'a été implémenté.

Les performances de Camunda ne sont pas testées. Le workflow utilise une API REST et ne propose pas de support pour GraphQL.

7.2 Méthodologie

Les tests de performance suivent une méthodologie définie. Cette méthodologie est décomposée en deux points : types de test et échantillons.

7.2.1 Types de test

Trois types de test sont effectués :

- **Test séquentiel**. Les requêtes sont envoyées les unes après les autres. Ce test calcule le temps nécessaire pour traiter n requêtes. Le temps de réponse est le temps écoulé entre l'envoi de la première requête et la réception de la dernière réponse. Pour mesurer cette valeur, nous avons développé un test manuel qui envoie des requêtes à la suite.
- **Test de charge**. Ce test consiste à augmenter progressivement le nombre d'utilisateurs effectuant des requêtes sur un endpoint spécifique. Le temps de réponse est mesuré grâce à JMeter et Autocannon. Le comportement de l'API est également évalué.
- **La méthode concurrente**. Ce test augmente progressivement le nombre d'utilisateurs présent utilisant l'API. Le trois endpoints sont sollicités en même temps à différentes fréquences. Le temps de réponse est mesuré grâce à JMeter. Le comportement de l'API est évalué.

7.2.2 Echantillons

Chaque test est effectué sur deux échantillons différents. Le premier échantillon, appelé **few**, remplit les bases de données avec:

- 10 utilisateurs
- 84 produits
- 79 commandes

le deuxième échantillon, appelé **lot**, remplit avec :

- 100'000 utilisateurs
- 84 produits
- 300'204 commandes

7.2.3 Récapitulatif

Le tableau 7.2 résume la liste des tests qui sont effectués :

Méthode	**API**	**Echantillon**	**Outil**
Test séquentielle	GraphQL	few	Test manuel
		lot	Test manuel
	REST	few	Test manuel
		lot	Test manuel
Test de charge	GraphQL	few	Autocannon
		lot	Autocannon
	REST	few	Autocannon
		lot	Autocannon
	GraphQL	few	JMeter
		lot	JMeter
	REST	few	JMeter
		lot	JMeter
Test concurrent	GraphQL	few	JMeter
		lot	JMeter
	REST	few	JMeter
		lot	JMeter

TABLE 7.2 – Liste des tests

Les informations concernant les algorithmes et l'utilisation des différents outils sont disponibles en annexe. Le langage utilisé pour les analyses est R.

7.2.4 Matériel

L'application tourne sur Docker. La version de Docker Desktop est 4.15.0. L'ordinateur utilisé pour les tests a les caractéristiques suivantes: Windows 10 Professionel 64Bits, Processeur i5-1145g7, RAM 32Gbits DDR4-3200.

7.3 Test séquentiel

Pour chaque fonction, cinq types de test ont été utilisés: l'envoi de 1, 10, 100, 1'000, et 10'000 requêtes. Chaque type a été réalisé un certain nombre de fois, comme présenté dans le tableau 7.4.

Il avait été prévu de réaliser 50 fois chaque type de test, pour chaque API, chaque fonction et chaque échantillon. Cependant, lors des tests, certaines fonctions ont présenté des temps de réponse dépassant une heure. Pour remédier à cette situation, le nombre de tests a été ajusté pour ces fonctions spécifiques.

Ech.	**Fonctions**	**API**	**1 R**	**10 R**	**100 R**	**1'000 R**	**10'000 R**
few	getUsers	GraphQL	50	50	50	50	50
		REST	50	50	50	50	50
	addProduct	GraphQL	50	50	50	50	50
		REST	50	50	50	50	50
	products-users	GraphQL	50	50	50	50	-
		REST	50	50	50	50	-
lot	getUsers	GraphQL	5	5	5	5	-
		REST	5	5	5	5	-
	addProduct	GraphQL	50	50	50	50	50
		REST	50	50	50	50	50
	products-users	GraphQL	5	5	-	-	-
		REST	5	5	-	-	-

[1] R = Requête

TABLE 7.4 – Nombre de requêtes

Les mesures ont permis d'estimer le temps de réponse moyen pour chaque type de test par fonction, par API et par échantillon. Le package zoo du langage R a permis d'interpoler les valeurs manquantes (le temps de réponse pour 3, 300 ou 3000 requêtes par exemple).

Les graphiques des figures 7.1, 7.2 et 7.3 illustrent le temps de réponse moyen par API, par fonction et par échantillon.

Le temps de réponse de la fonction **addProduct** (figure 7.1) ne montre pas de différence entre REST et GraphQL lorsque le nombre de requêtes est inférieur à 1000. Au-delà de 1000, un léger avantage en faveur de REST est visible. Le résultat est similaire entre les échantillons **few** et **lot**. La taille de l'échantillon ne semble pas avoir d'impact sur cette fonction.

Concernant la fonction **getUsers** (figure 7.2) et l'échantillon **few**, aucune différence entre REST et GraphQL n'est visible lorsque le nombre de requêtes est inférieur à 2500. À partir de 2500, un léger avantage en faveur de REST est visible. Cependant, pour la même fonction **getUsers** mais avec l'échantillon lot, GraphQL présente un temps de réponse fortement plus faible que REST. Pour 10'000 requêtes, la différence est de 31% en faveur de GraphQL. GraphQL semble être plus avantageux que REST lorsque la quantité de données retournées est élevée.

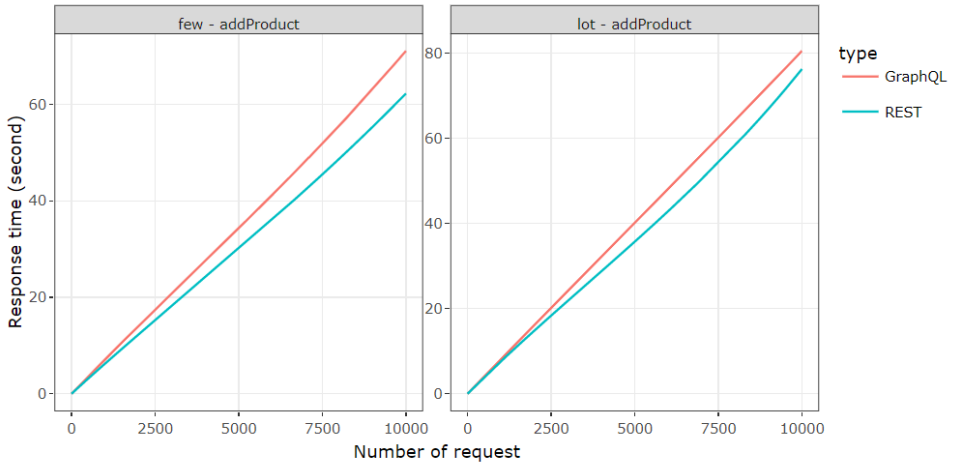

FIGURE 7.1 – Temps de réponse en fonction du nombre de requêtes - addProduct (POST)

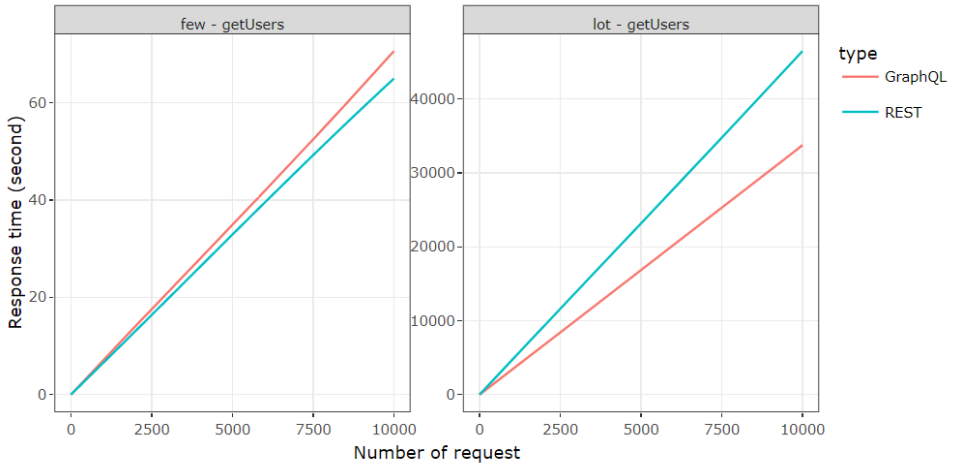

FIGURE 7.2 – Temps de réponse en fonction du nombre de requêtes - getUsers (GET)

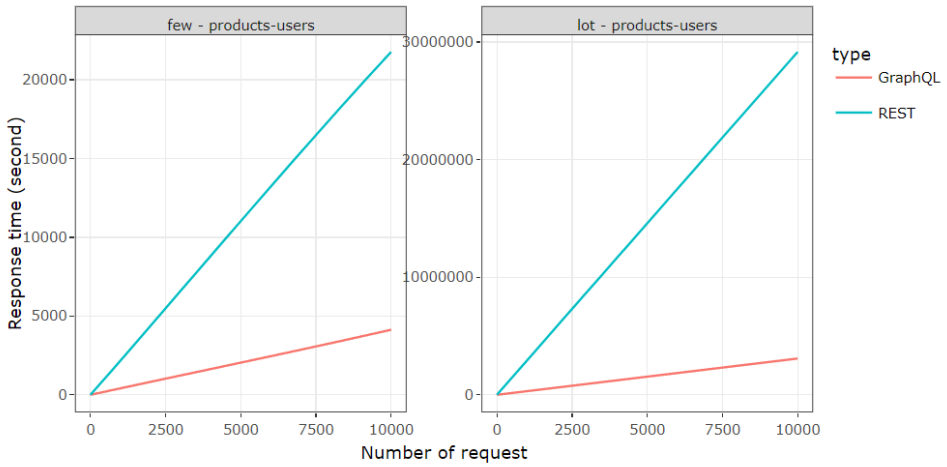

FIGURE 7.3 – Temps de réponse en fonction du nombre de requêtes - products-users (GET)

Pour la fonction **products-users** (figure 7.3), GraphQL est plus performant que REST. La taille de l'échantillon (few ou lot) n'influence pas les résultats. Pour 10 requêtes, la différence est de 161% en faveur de GraphQL. GraphQL semble avoir un avantage lorsque les données sont imbriquées.

Le test séquentiel permet de conclure que la différence entre REST et GraphQL se fait sur trois critères:

- Le nombre de requêtes à effectuer.
- La quantité de données retournées.
- Le type de données retournées (simples ou imbriquées).

Pour de petites quantités de données, REST et GraphQL s'équivalent. Cependant, lorsqu'un grand nombre de données doit être retourné ou que ces données proviennent de plusieurs sources, l'API GraphQL est plus performante.

À retenir

- REST: Avantageux pour les données peu volumineuses ou données avec structures simples.
- GraphQL: Avantageux pour les données complexes et volumineuses.

7.4 Test de charge

7.4.1 Autocannon

Le premier test de charge est effectué grâce à Autocannon. Cette librairie NodeJS permet de tester les performances d'une application en simulant l'envoi de nombreuses requêtes simultanées sur un endpoint spécifique. L'outil retourne un résumé des données mesurées.

La présence de 1, 10, 100, 1000 et 10000 utilisateurs simultanés a été simulée. Chaque simulation a duré 100 secondes. Chaque test a été effectué 10 fois et sur les deux échantillons. Les graphiques de la figure 7.4 présentent la répartition des requêtes réussies et échouées en fonction du nombre d'utilisateurs pour l'échantillon few.

GraphQL et REST ne montre pas de différence pour les fonctions **addProduct** et **getUsers**. Ces fonctions ont tous les deux 100% de requêtes réussies lorsque le nombre d'utilisateurs concurrents est inférieur ou égal à 1000. Cependant, cela n'est plus le cas si le nombre d'utilisateurs est supérieur à ce seuil.

Pour la fonction **products-users**, GraphQL ne supporte pas la présence de 100 utilisateurs concurrents. Le taux d'échec est de quasiment 100% à partir de ce seuil. Pour REST, l'échec des requêtes commence à partir de 1000 utilisateurs. Par conséquent, REST semble adapté pour les services devant supporter un nombre élevé d'utilisateurs.

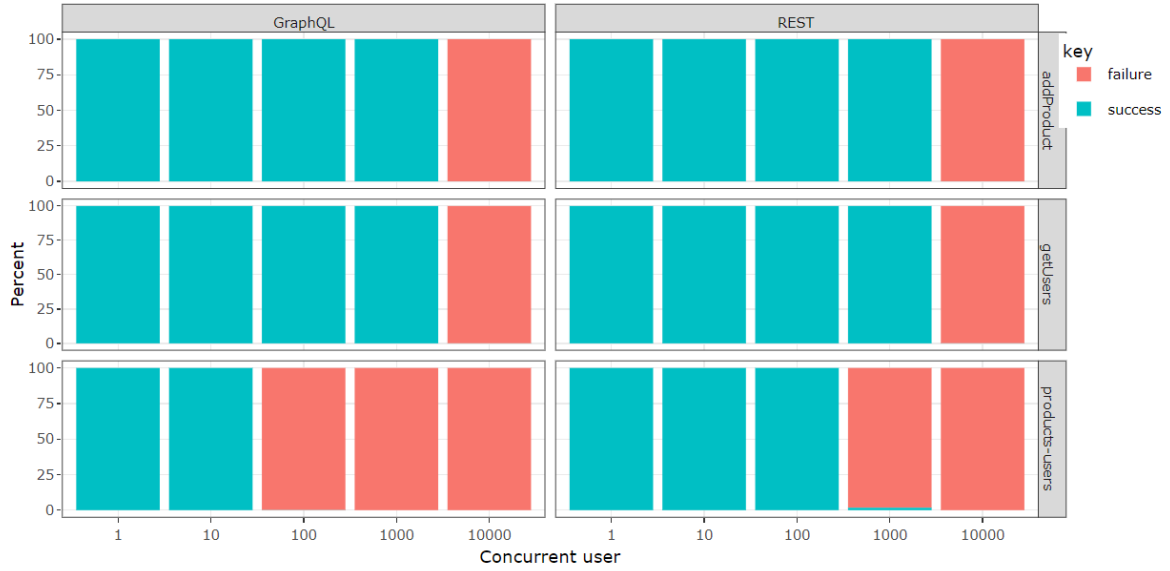

FIGURE 7.4 – Répartition des requêtes réussies et échouées en fonction du nombre d'utilisateurs concurrents sur l'échantillon few

Les graphiques de la figure 7.5 présentent le temps de réponse moyen et l'écart-type en fonction du nombre d'utilisateurs concurrents pour l'échantillon few.

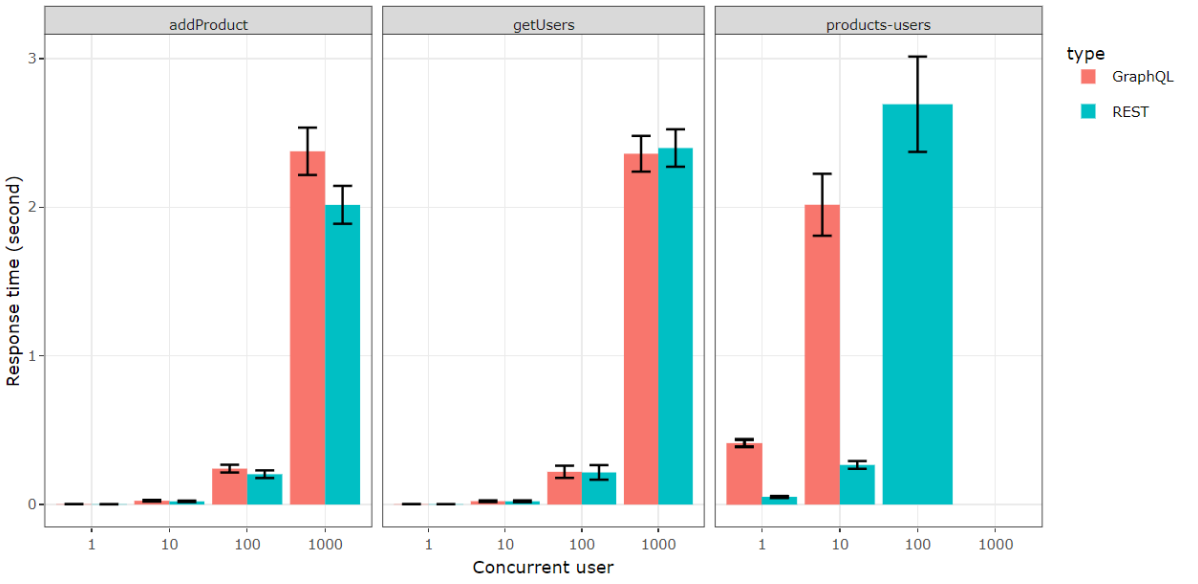

FIGURE 7.5 – Temps de réponse moyen et écart type

Ce graphique contient seulement les requêtes réussies. De manière générale, REST présente un temps de réponse moyen nettement inférieur à celui de GraphQL. GraphQL semble ne pas pouvoir traiter de grands nombres de requêtes concurrentes.

Le test a également été effectué pour l'échantillon lot. Malheureusement, les temps de réponse très élevés des fonctions **getUsers** / **products-users** et le pourcentage de requêtes échouées rendent les données inexploitables. Par conséquent, l'étude de cet échantillon ne sera pas réalisée.

7.4.2 JMeter

Le deuxième test de charge a été effectué avec JMeter qui est un outil de benchmark très complet. Il propose plusieurs types de requêtes (http, graphQL, FTP, TPC, etc...) et supporte également l'utilisation de plugins pour étendre ses fonctionnalités. Les résultats peuvent être suivis en temps réel grâce à des graphiques et des tableaux. Les données peuvent être exportées en csv.

Contrairement à Autocannon, JMeter fournit des résultats détaillés qui peuvent être utilisés pour des analyses plus poussées.

Pour le test de charge, le plugin Concurrency Thread Group, développé par BlazeMeter Inc. a été utilisé. Ce plugin permet de simuler une augmentation progressive des utilisateurs concurrents. La figure 7.6 présente le paramétrage de JMeter.

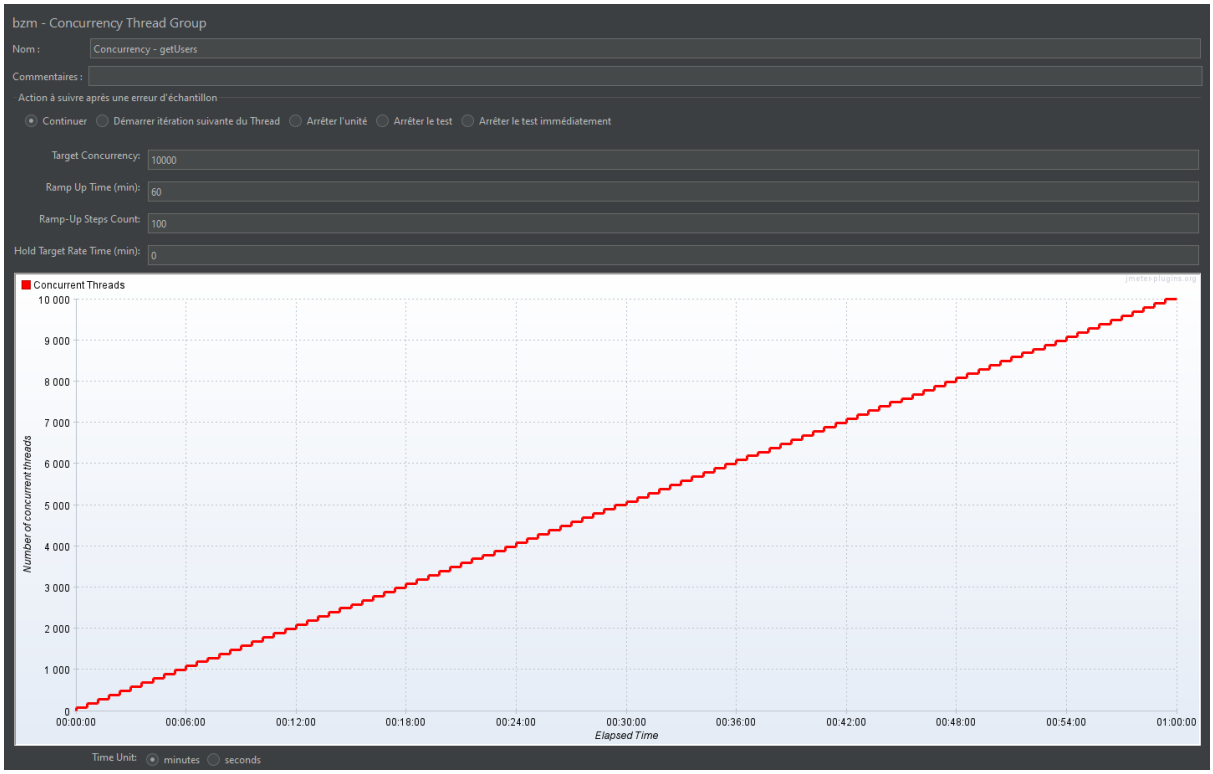

FIGURE 7.6 – Paramétrage de JMeter pour le test de charge

L'objectif est d'atteindre les 10'000 utilisateurs concurrents. Pour cela, le plugin va simuler une augmentation des utilisateurs par palier. Le benchmark tourne durant 60 minutes et effectue 100 paliers. Cela fait une augmentation de 100 utilisateurs toutes les 36 secondes. Les trois fonctions sont testées séparément.

Résultat

Le tableau 7.5 présente, pour chaque fonction, la répartition de l'état des requêtes pour l'échantillon **few**.

Fonctions	API	Succès	Echec	Pourcentage succès (%)
getUsers	graphql	403'457	5'712'903	6.60
getUsers	rest	370'761	5'148'928	6.72
addProduct	graphql	372'543	3'158'084	10.55
addProduct	rest	380'091	4'163'873	8.36
products-users	graphql	140'286	12'510'546	1.11
products-users	rest	178'045	7'617'850	2.28

TABLE 7.5 – Etats des requêts pour l'échantillon few

Le taux de succès des requêtes est faible. En effet, le serveur n'a pas été optimisé pour supporter un nombre d'utilisateurs concurrents aussi élevé. En général, dans une entreprise, la charge de 10'000 utilisateurs est partagée entre plusieurs serveurs distincts. De plus, les différents services tournent sur Docker et doivent se partager les ressources, ce qui conduit à une instabilité et explique en partie ce taux d'échec élevé.

En termes de pourcentage de succès, GraphQL et REST semblent similaires. La différence entre les deux est de plus ou moins 1.5%.

Le test a été effectué une deuxième fois afin d'améliorer la fiabilité des résultats. Les résultats sont similaires à ceux présents dans le tableau.

Comme pour Autocannon, les requêtes effectuées avec l'échantillon **lot** présentent un très faible pourcentage de succès. En effet, le temps de réponse très élevé des fonctions getUsers et product-users ne permettent pas d'obtenir des résultats exploitables. Par conséquent, cet échantillon ne sera pas analysé.

Les graphiques suivants se baseront sur l'échantillon **few** et les requêtes réussies.

Temps de réponse

Les graphiques de la figure 7.7 montre l'évolution du temps de réponse durant les 60 minutes. Les temps de réponse des trois fonctions augmentent avec la durée du test. La fonction **products-users** a un temps de réponse qui s'accroît exponentiellement, tandis que les fonctions **addProduct** et **getUsers** ont une augmentation plutôt linéaire. La différence n'est pas significative entre REST et GraphQL pour les fonctions **addProduct** et **getUsers**. De temps en temps, les deux API subissent des pics d'augmentation de leurs temps de réponse. Cependant, une différence est notable pour la fonction **products-users**. À partir de 30 minutes, le temps de réponse fluctue fortement, variant entre 100 secondes et 200 secondes. Le nombre d'utilisateurs et la forte consommation des ressources peuvent expliquer cette variation. GraphQL montre des temps de réponse plus bas que REST. Par conséquent, il semblerait que GraphQL soit en mesure de supporter plus de requêtes concurrentes que REST. Ces résultats sont différents de ceux présentés pour Autocannon. Cela peut être dû aux méthodes de calcul utilisées et à la pression exercée par les outils sur les services.

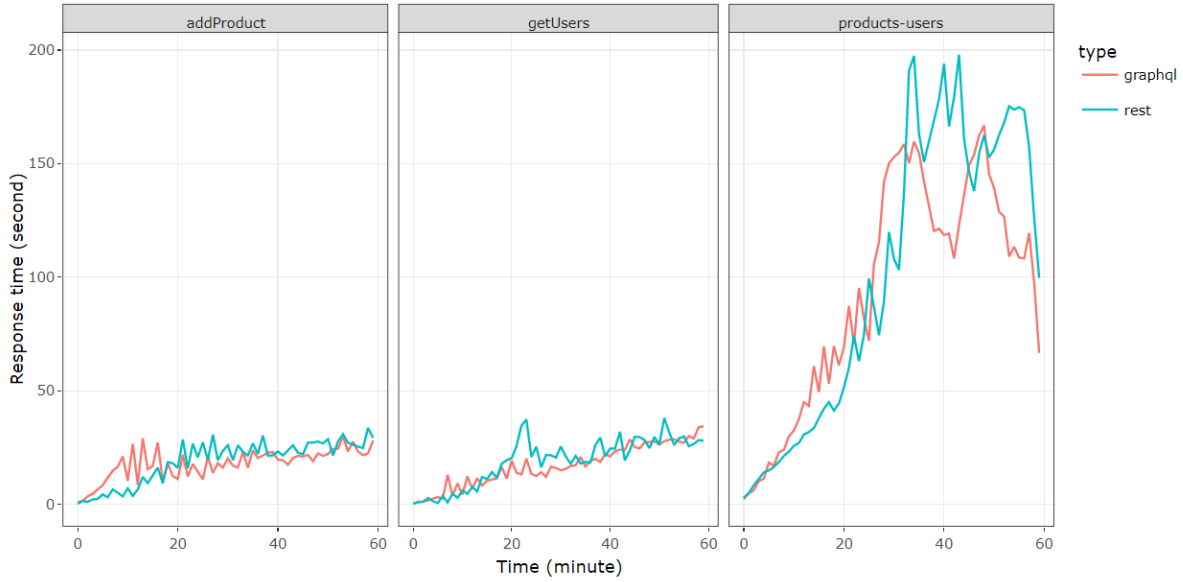

FIGURE 7.7 – Temps de réponse en fonction du temps - test de charge

Les graphiques de la figure 7.8 illustrent les temps de réponse en fonction du nombre d'utilisateurs simultanés. Les points représentent la moyenne des temps de réponse. La ligne est une courbe de tendance.

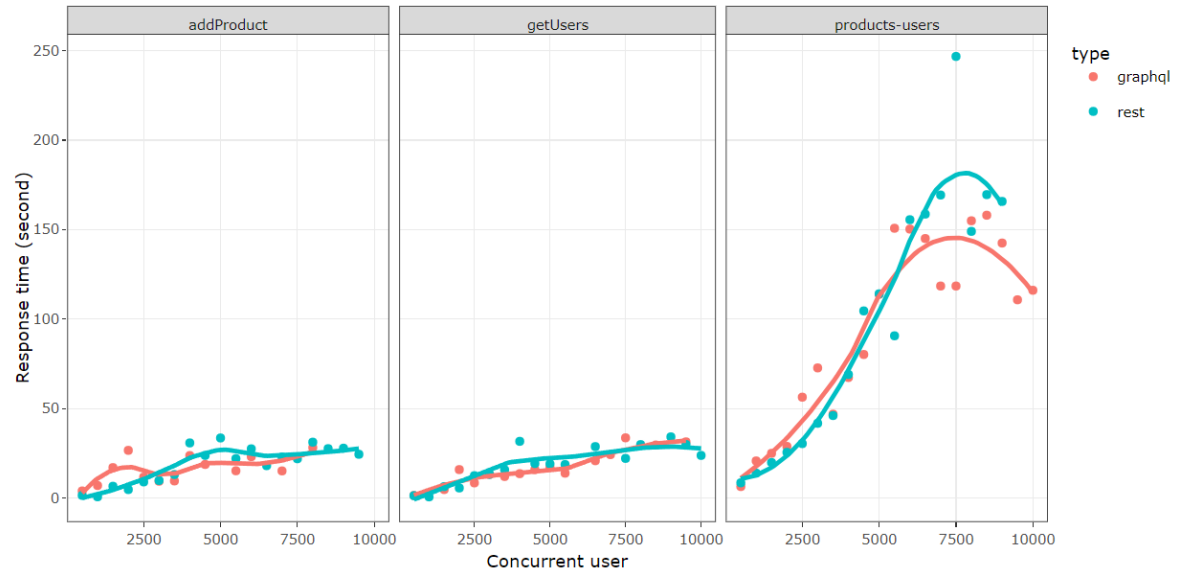

FIGURE 7.8 – Distribution du temps de réponse en fonction du nombre d'utilisateur concurrent - test de charge

Les fonctions getUsers et addProduct ne montrent pas de différence notable entre les temps de réponse de REST et GraphQL. Les deux API voient leur temps de réponse augmenter linéairement, passant de 3 secondes en moyenne pour 500 utilisateurs concurrents à 30 secondes pour 10'000 utilisateurs concurrents. Cependant, le temps de réponse de la fonction products-users augmente exponentiellement, passant de 8 secondes pour

500 utilisateurs à 170 secondes pour 7500 utilisateurs. Sur cette fonction, GraphQL a un temps de réponse inférieur à REST à partir de 6000 utilisateurs concurrents.

La différence entre les deux API n'est pas perceptible. Sur ce test, REST et GraphQL semblent similaires.

À retenir

Autocannon

- REST supporte mieux un nombre d'utilisateurs élevé.
- Meilleurs temps de réponse pour REST.

JMeter

- Sur des fonctions simples (getUsers et addProduct): temps de réponse similaires pour les deux API.
- Sur une fonction complexe (products-users): temps de réponse similaires jusqu'à 6000 utilisateurs simultanés, puis avantage pour GraphQL.

7.5 Test concurrent

Le test concurrent a été effectué avec JMeter. L'objectif est d'évaluer la performance globale du serveur. JMeter a été paramétré de façon à effectuer des requêtes sur les trois fonctions en simultané. La figure 7.9 présente le paramétrage de JMeter.

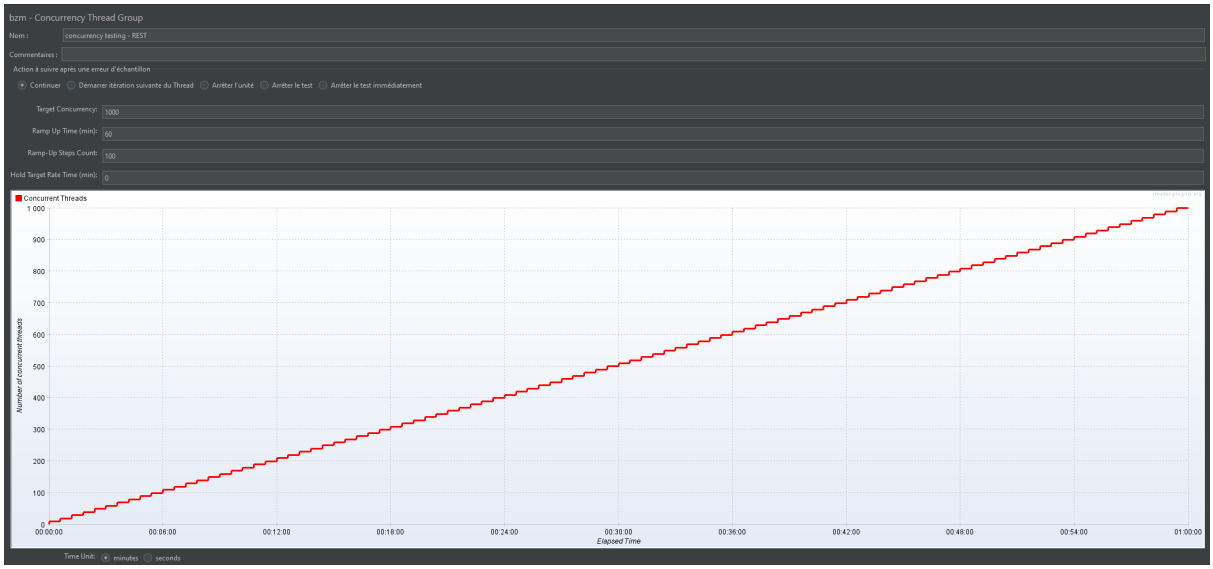

FIGURE 7.9 – Paramétrage de JMeter pour le test concurrent

L'objectif est de simuler la présence de 1'000 utilisateurs concurrents. Le benchmark tourne durant 60 minutes et effectue 100 paliers. Toutes les 36 secondes, l'outil simule la présence de 10 utilisateurs en plus. Des requêtes sont effectuées sur les trois fonctions en même temps.

Résultat

Le tableau 7.6 présente, pour chaque fonction, la répartition de l'état des requêtes pour l'échantillon **few**.

Fonctions	API	Succès	Echec	Pourcentage succès (%)
global	graphql	417'174	993'396	29.57
global	rest	322'652	698'182	31.61
getUsers	graphql	204'741	265'037	43.58
getUsers	rest	155'965	184'681	45.79
addProduct	graphql	196'130	271'524	41.94
addProduct	rest	143'293	193'612	42.53
products-users	graphql	16'303	456'835	3.45
products-users	rest	23'394	319'889	6.81

TABLE 7.6 – Etats des requêts pour l'échantillon few

Comparé au tableau 7.5, le pourcentage de succès des différentes fonctions est considérablement plus élevé. Il est possible que cela soit dû à la simulation de seulement 1'000 utilisateurs concurrents contre 10'000 utilisateurs précédemment. La fonction products-users a un pourcentage de succès nettement inférieur aux deux autres fonctions. Une explication possible est que cette fonction implique la fusion de données de trois sources différentes, ce qui peut causer l'échec de la requête si l'une des sources est affectée par le nombre d'utilisateurs concurrents.

La fonction **global** représente les trois fonctions réunies. La différence entre les pourcentages de succès de GraphQL et de REST est d'environ 1.5%.

La figure 7.10 illustre le temps de réponse de l'API durant le test. Les temps de réponse des deux API sont instables. De fortes variations sont présentes. Ces fluctuations peuvent être causées par la difficulté des API à gérer un grand nombre d'utilisateurs et un nombre élevé de requêtes simultanées. Dans notre cas, trois fonctions sont appelées simultanément, ce qui semble déstabiliser les API.

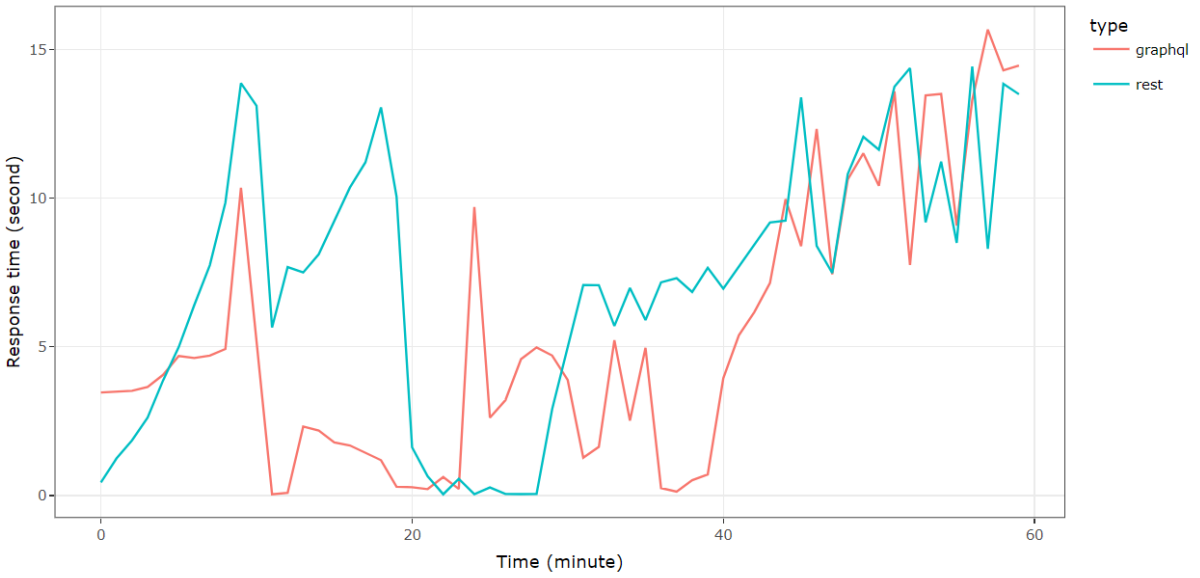

FIGURE 7.10 – Temps de réponse en fonction du temps - test concurrent

La figure 7.11 présente les mêmes données séparées par fonction.

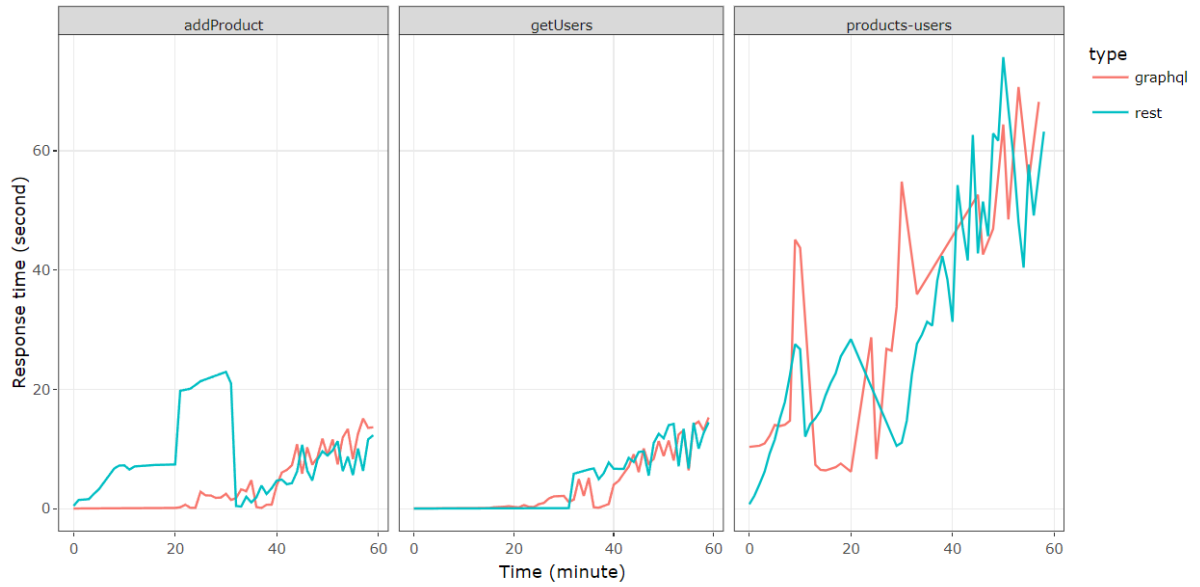

FIGURE 7.11 – Temps de réponse en fonction du temps et par la fonction - test concurrent

Pour la fonction **addProduct**, l'API REST subit une forte augmentation de son temps de réponse entre la 20ème et la 30ème minute. À partir de la 30ème minute, les deux API semblent se stabiliser et subissent une augmentation du temps de réponse presque linéaire, avec des fluctations mais moins importantes. La fonction getUsers ne semble montrer aucune variation jusqu'à la 30ème minute. Après cela, le temps de réponse augmente également de façon linéaire. La fonction **products-users**, quant à elle, reste instable tout le long du test. Le temps de réponse passe de 5 secondes en moyenne au début du test, à 60 secondes en moyenne après 60 minutes.

La figure 7.12 présente le temps de réponse global en fonction du nombre d'utilisateurs concurrents. Jusqu'à 500 utilisateurs concurrents, les temps de réponse de REST et GraphQL diminuent légèrement avant de remonter. Les résultats sont cohérents avec les figures précédentes, lesquelles ont également montré que le temps de réponse n'augmentait qu'après 30 minutes. Le temps de réponse de GraphQL reste plus bas que ceux de REST jusqu'à 750 utilisateurs. Au-delà de ce seuil, la tendance s'inverse.

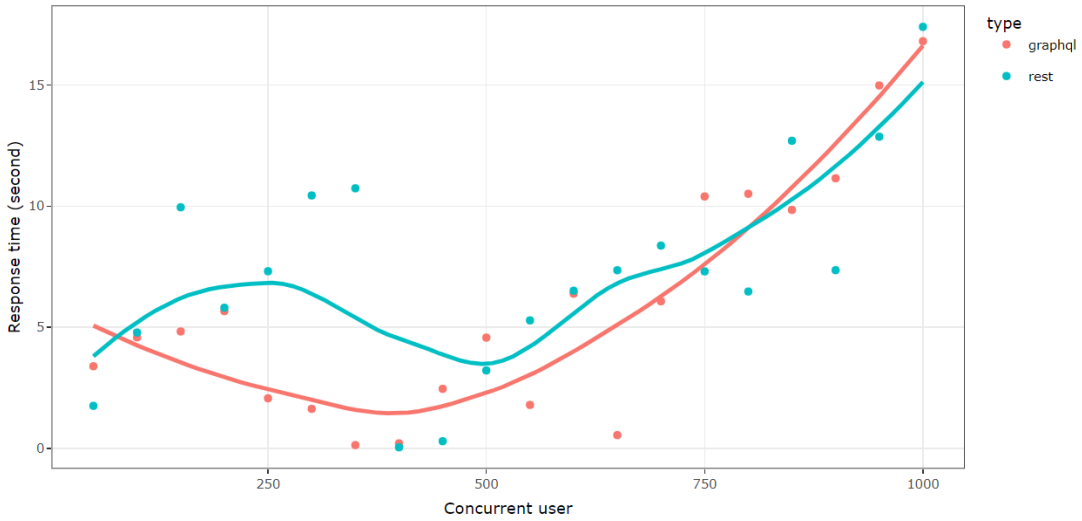

FIGURE 7.12 – Temps de réponse en fonction du nombre d'utilisateurs concurrents - test concurrent

Les graphiques de la figure 7.13 illustrent le temps de réponse par fonction. Les observations précédentes sont également visibles sur cette figure. Les temps de réponse des fonctions **addProduct** et **getUsers** atteignent au maximum 20 secondes tandis que ceux products-users montent jusqu'à 80 secondes.

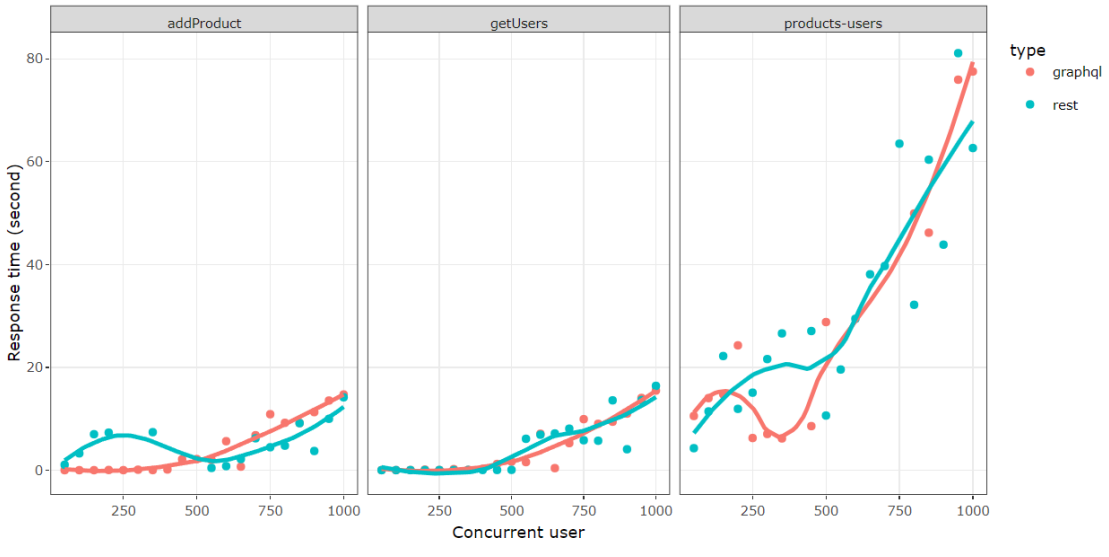

FIGURE 7.13 – Temps de réponse en fonction du nombre d'utilisateurs concurrents et par fonction - test concurrent

À retenir

- Les deux API sont instables sur les 30 premières minutes. Ensuite augmentation du temps de réponse de façon linéaire.
- sur la fonction addProduct, REST subit une forte augmentation de son temps de réponse entre la 20ème et la 30ème minute.
- Pas de différence significative entre REST et GraphQL.

7.6 Synthèse

L'utilisation de trois types de test, de deux échantillons et de trois fonctions nous a permis de montrer qu'il n'y a pas de réelle différence entre les deux API. En effet, la méthode séquentielle a montré des temps de réponse favorables pour REST, sur l'échantillon few. Cependant, sur l'échantillon lot, GraphQL a su se démarquer. Cela confirme les résultats du chapitre 5. GraphQL semble être plus performant sur les données imbriquées.

Le test de charge et le test concurrent n'ont également pas pu départager les deux API. Autocannon montre des résultats avantageux pour REST tandis que JMeter ne départage pas les deux API. Le choix de l'API ne doit donc pas se faire sur les critères de temps de réponse ni d'utilisateurs concurrents mais plutôt sur la finalité de l'API. Si le but est de retourner des grandes quantités de ressources imbriquées, alors GraphQL serait la meilleure option. Dans le cas contraire, REST est optimal pour la plupart des situations.

8
Conclusion

Cette thèse de master s'est intéressée à l'analyse de l'architecture, des API et des workflows utilisés dans la partie serveur d'une application web. L'objectif de cette recherche a été de répondre à la question suivante: **"Quelles sont les architectures et les API que les entreprises doivent privilégier pour s'adapter à leur croissance et à l'évolution technologique ?"**. À travers cette thèse, nous avons étudié l'architecture des services web, les workflows et les API. Nous avons également créé une application web pour l'entreprise Stock&Co, afin de mettre en pratique les notions apprises.

Dans une première partie, nous avons abordé l'impact de la digitalisation sur les entreprises et l'importance des applications web. Ces applications web sont devenues le point de contact entre les clients et les entreprises. En plus de favoriser l'innovation et la création de nouveaux marchés, les applications web permettent aux entreprises d'attirer de nouveaux clients.

Puis, nous avons présenté l'histoire des applications web, leur évolution et les défis qu'elles présentent pour les entreprises. Parmi ces défis, deux sont particulièrement importants. Le premier est l'impact des applications web sur l'économie des entreprises. Le deuxième est l'adaptation des applications web aux entreprises, notamment lorsqu'elles sont en phase de croissance. Cette dernière problématique est le sujet de recherche de cette thèse. L'adaptation d'une application web concerne plusieurs composants distincts que nous avons étudiés dans les chapitres suivants.

Dans la deuxième partie a été consacrée aux architectures monolithiques, SOA et en microservices. L'architecture monolithique a été, durant de nombreuses années, un standard de développement web. Cependant, avec la croissance des entreprises et l'évolution du web, cette architecture a été abandonnée au profit des SOA et microservices. Les avantages et les inconvénients de chacune de ces architectures ont été discutés. Par exemple, l'architecture monolithique convient aux petites applications, mais devient complexe à mesure que l'application se développe. Les architectures SOA et en microservices facilitent le développement de services indépendants, mais cette indépendance peut entraîner un manque de coordination entre les services. Pour résoudre ce problème, deux types de workflow ont été proposés: les workflows en orchestration et les workflows chorégraphiques. Les workflows en orchestration sont gérés à l'aide d'un workflow engine et permettent de visualiser l'avancée des processus. Les workflows chorégraphiques utilisent une application externe comme courtier et communiquent via un système d'événements. L'utilisation de l'un ou l'autre dépend des besoins de l'entreprise. Nous avons constaté que la mise en place d'un workflow, en particulier un workflow chorégraphique, peut être une tâche

complexe. L'utilisation de AsyncAPI peut s'avérer utile pour résoudre ces problèmes et simplifier la mise en place du workflow.

La troisième partie s'est focalisée sur trois technologies facilitant le développement des API: REST, GraphQL et gRPC. Ces trois API ont des modes de fonctionnement différents et ne sont pas utilisées pour résoudre le même problème. REST est un standard pour le développement des API. Il fournit quelques contraintes pour uniformiser l'architecture. GraphQL est un langage de requête qui simplifie la récupération de données imbriquées. Il permet également de filtrer les attributs des ressources. Finalement, gRPC facilite la communication entre les composants indépendants, tels que les microservices, en utilisant le langage protoBuffer pour générer les services et en proposant plusieurs modes de communication.

Dans la suite de ce travail, nous avons comparé les trois APIs en nous basant sur la littérature et les recherches scientifiques. Les recherches montrent qu'il n'y a pas une API qui soit performante dans toutes les situations. Les résultats montrent que GraphQL excelle lors de la récupération de ressources imbriquées et pour le traitement des réponses volumineuses. gRPC semble être le choix idéal pour les petits appareils et pour les données légères. Enfin, REST semble être la technologie la plus appropriée pour la plupart des situations.

Dans la dernière partie de ce travail dédiée à l'application des principes théoriques appris, nous avons développé un prototype pour l'entreprise Stock&Co. L'objectif est de démontrer que les éléments abordés dans les chapitres précédents peuvent être intégrés ensemble. Nous avons opté pour une architecture en microservices, une API en GraphQL et un workflow en orchestration car ces choix répondent aux exigences de Stock&Co.

Finalement, nous avons évalué les performances de notre API GraphQL en testant la partie serveur de notre application. Pour effectuer une comparaison avec une autre technologie, nous avons créé trois endpoints pour REST et les avons exposés. Nous avons ensuite comparé les performances de ces deux API sur trois tests différents: test séquentiel, test de charge et test concurrent. Ces tests n'ont pas permis de départager les deux API. Les résultats varient en fonction du nombre d'utilisateurs simultanés simulés et des endpoints utilisés. Par conséquent, nous avons conclu que le choix de l'API ne doit pas se baser uniquement sur les performances mais plutôt sur l'utilité. Si l'objectif est de récupérer de grandes quantités de données imbriquées, GraphQL est le choix optimal. Sinon, REST est suffisant pour la plupart des situations.

Ce travail a permis d'avoir une comparaison théorique et pratique de trois composants utilisés régulièrement par les entreprises dans le développement des applications web. Pour les futures recherches, nous pourrions développer un service gRPC. La littérature comparant les trois API étant peu fréquente, nos travaux pourraient apporter une contribution. De plus, nous pourrions également comparer les performances entre une installation local sur l'ordinateur et Docker.

9 Annexe

Ce chapitre contient toutes les informations nécessaires pour lancer l'application et effectuer les tests.

9.1 Informations

Les microservices ont été développés avec Javascript et le framework Express. Le client a été développé avec ReactJS. Pour simplifier l'utilisation, le prototype fonctionne sur Docker. Docker et Docker Desktop doivent être installés.

9.2 Détails techniques

9.2.1 Les services

Le tableau 9.1 présente les 15 services qui composent l'application :

Service	**Type**	**Port**
mysql-camunda	Workflow	Port 3310
camunda	Workflow	http://localhost:8080/camunda
mongo-logs-microservice	Logs	Port 9006
mongo-user-microservice	DB	Port 9000
mongo-express-user-microservice	Interface de la DB user	http://localhost:9001
user-microservice	GraphQL	http://localhost:8082/graphql
mongo-order-microservice	DB	Port 9002
mongo-express-order-microservice	Interface de la DB order	http://localhost:9003
order-microservice	GraphQL	http://localhost:8083/graphql
mongo-product-microservice	DB	Port 9004
mongo-express-product-microservice	Interface de la DB product	http://localhost:9005

product-microservice	GraphQL	http://localhost:8084/graphql
gateway	Sert à déployer le modèle BPMN	Port 8085
mesh	Reverse proxy Gateway	http://localhost:4000/graphql
client	Client	http:/localhost:3000

TABLE 9.1 – Liste des services

Les services en gras sont obligatoires pour le bon fonctionnement de l'application. Les autres peuvent être activés ou désactivés. Ces services sont définis dans le fichier docker-compose.yml.

9.2.2 Script de lancement

Plusieurs services sont interdépendants et doivent être lancés dans un certain ordre pour le bon fonctionnement. Les services suivants contiennent chacun un script **start.sh** qui vérifie, grâce à des commandes curl, si leurs dépendances sont actives avant de démarrer:

- gateway
- mesh
- order-microservice
- product-microservice
- user-microservice
- client

Note: Si lors du lancement, les services cessent de fonctionner et vous recevez l'erreur "**/start.sh: No such file or directory**":

- Ouvrez les script start.sh dans un IDE, par exemple Visual Studio Code.
- Vérifiez que le fichier est en LF (en bas à droite dans Visual Studio Code).
- Si ce n'est pas le cas, changez CRLF en LF.
- Supprimer tous les containers, images et volumes et relancer

9.2.3 Données fictives

Les services order-microservice, product-microservice et user-microservice contiennent des fonctions qui permettent de créer des données fictives. Si ces données ne sont pas nécessaires, il suffit de commenter :

- dans order-microservice: la ligne 84 du fichier index.js (load_orders)
- dans user-microservice: la ligne 119 du fichier index.js (load_users)
- dans product-microservice: la ligne 92 du fichier index.js (load_database)

9.3 Téléchargement et installation du project Stock&Co

Vérifier que les ports présents dans le tableau 9.1 soit libres.

1. Cloner le répertoire https://github.com/KesThav/stock-and-co
2. Ouvrir le terminal à la racine du dossier
3. Dans le terminal, écrire "docker compose up".

Avant chaque nouveau lancement, il faut:

- supprimer les containers, images et volumes créés par l'instance précédente.
- Effacer les données (token et basket) se trouvant dans le localstorage du navigateur.

9.4 Architecture des services

Les figures 9.1 à 9.5 présentent les dépendances entre les fichiers de chaque service. En violet, ce sont les fichiers dépendants d'autres fichiers. En vert, ce sont des fichiers indépendants. Les figures ont été générées grâce à la librairie Madge (`https://github.com/pahen/madge`).

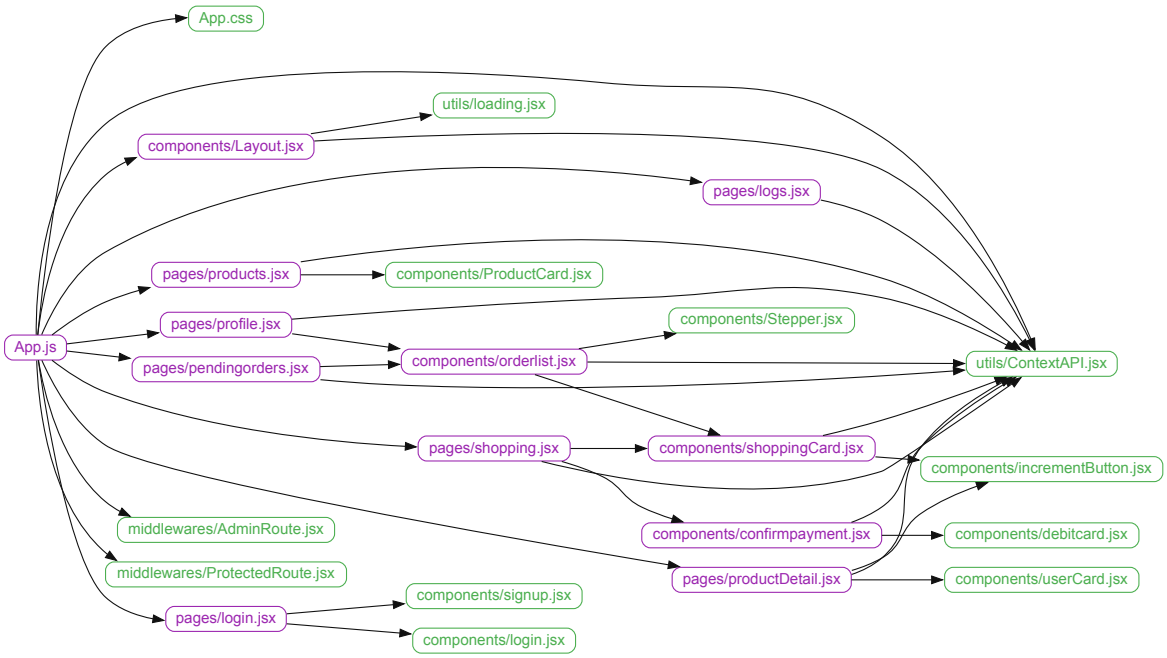

FIGURE 9.1 – Architecture du client

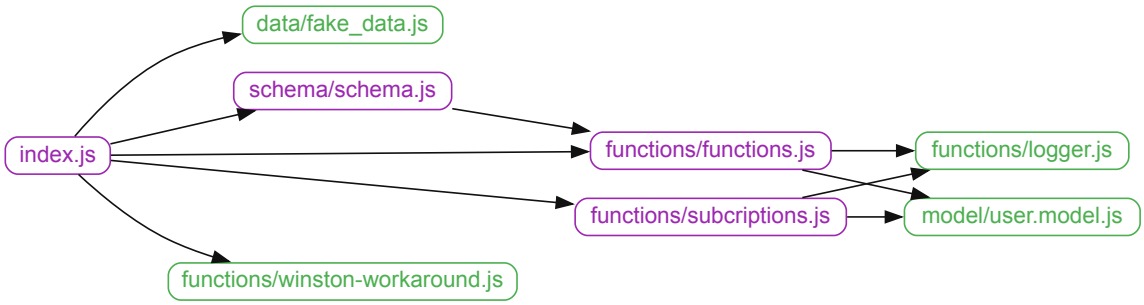

FIGURE 9.2 – Architecture du microservice utilisateur

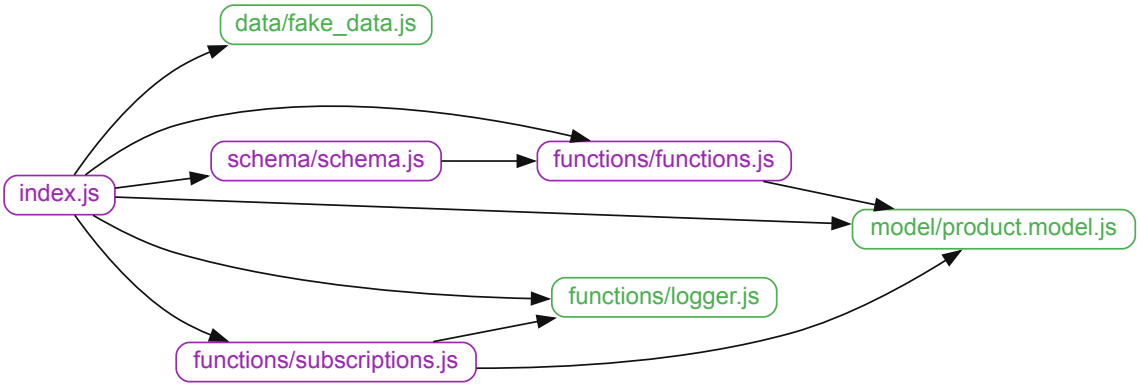

FIGURE 9.3 – Architecture du microservice produit

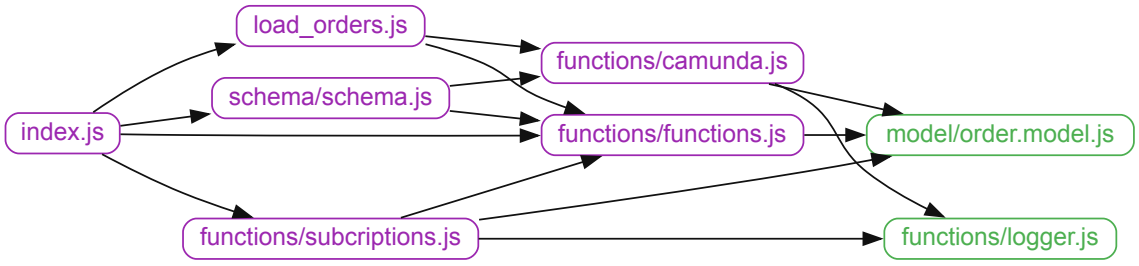

FIGURE 9.4 – Architecture du microservice commande

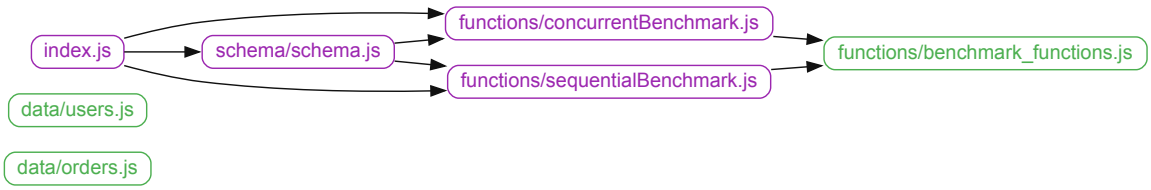

FIGURE 9.5 – Architecture du microservice benchmark (sequentiel + concurrent/auto-cannon)

9.5 Liste des Queries et Mutations

Le tableau 9.2 contient la liste des Queries et Mutations implémentées.

Type	**Nom**	**Arguments**	**Description**
Query	users		Retourne tous les utilisateurs.
	user	id	Retourne un utilisateur
	products		Retourne tous les produits.
	product	id	Retourne un produit.
	orders		Retourne toutes les commandes.
	order	id	Retourne une commande
	orderByUser	userid	Retourne toutes les commandes d'un utilisateur.
	orderByProduct	id	Retourne toutes les commandes contenant un produit.
	queryLogs		Retourne l'historique de Camunda.
	userTasksAndRelatedOrders	id	Retourne toutes les tâches ouvertes dans Camunda avec l'utilisateur concerné.
Mutation	register	name email password	Crée un utilisateur et retourne ses informations.
	login	email password	Retourne un token jwt.
	addProduct	name description type averageRating quantity price images	Crée un produit.
	updateProduct	id name description type averageRating quantity price images	Met à jour un produit.
	deleteProduct		Supprime un produit.

	createOrder	userid products total status orderid discount	Crée une commande.
	updateOrderStatus	id status	Met à jour le statut d'une commande.
	startOrder	userid order ptype orderid discount	Démarre une instance Camunda.
	completeTask	taskid	Termine une instance Camunda.

TABLE 9.2 – Liste des Queries et Mutations

9.6 Benchmark

Le service benchmark permet d'effectuer les tests de performance. Il se lance sur le port 10000. Il est préférable de désactiver la connexion avec le workflow. Pour ce faire :

1. Aller dans order-microservice/load_order.js
2. Mettre à jour les lignes 129 à 133 :
   - Avant

```
1  //await createOrder(myorder);
2  console.log("Order " + order + " loaded !");
3  order++;
4  await startInstance(data_camunda);
5  await sleep(1000);
```

   - Après

```
1  await createOrder(myorder);
2  console.log("Order " + order + " loaded !");
3  order++;
4  //await startInstance(data_camunda);
5  //await sleep(1000);
```

9.6.1 Les échantillons

L'échantillon **few** est généré par les données fictives de la section 9.2.3. Pour générer l'échantillon **lot**, effectuer une requête POST sur les endpoints suivants:

- http://localhost:8082/users/load
- http://localhost:8084/orders/load

9.6.2 Les endpoints

Les endpoints suivants permettent de démarrer les benchmarks (port 10000):

- Pour REST
  - Méthode séquentielle
    - /benchmarks/sequential/users (manuel)
    - /benchmarks/sequential/product (manuel)
    - /benchmarks/sequential/products/users (manuel)
  - Méthode concurrente
    - /benchmarks/load/users (Autocannon)
    - /benchmarks/load/product (Autocannon)
    - /benchmarks/load/products/users (Autocannon)
- Pour GraphQL
  - /graphql

Les résultats sont sauvegardés, sous forme de fichier JSON, dans benchmark/results

9.6.3 Benchmark JMeter

Pour effectuer le benchmark JMeter, l'outil JMeter est le plugin bzm sont nécessaires.

- Télécharger JMeter: https://jmeter.apache.org/
- Copier le fichier Benchmark/plugin_jmeter/lib/jmeter-plugins-cmn-jmeter-0.6.jar dans apache-jmeter/lib
- Copier le fichier Benchmark/plugin_jmeter/lib/etc/jmeter-plugins-casutg-2.10 dans apache-jmeter/lib/etc
- Copier le fichier Benchmark/plugin_jmeter/lib/etc/jmeter-plugins-manager-1.6 dans apache-jmeter/lib/etc
- Lancer JMeter en cliquant sur apache-jmeter/bin/ApacheJMeter.jar
- Ouvrir le fichier Loading_testing.jmx se trouvant dans Benchmark/plugin_jmeter.

Lancez un benchmark à la fois. Désactivez les autres en effectuant un clic-droit.

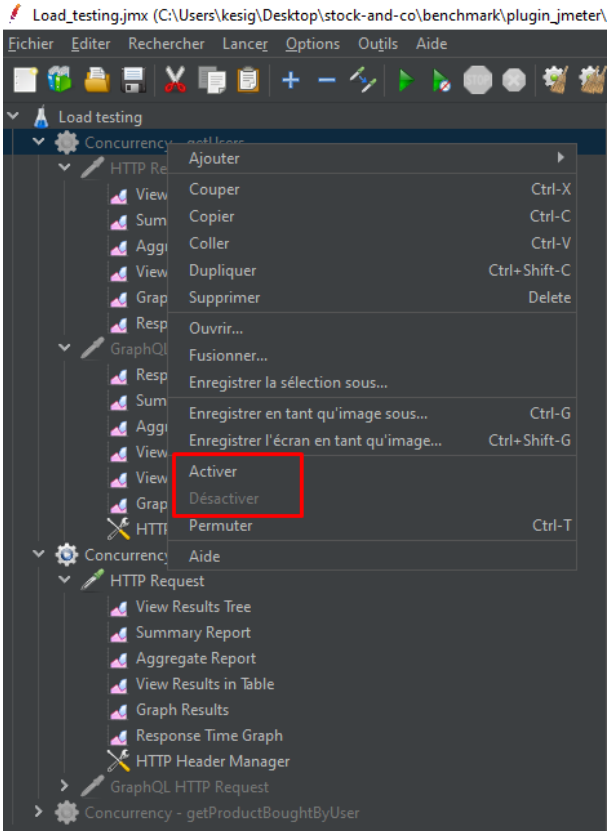

FIGURE 9.6 – Activer/désactiver un benchmark

Pour enregistrer les données en csv, changez le chemin d'accès:

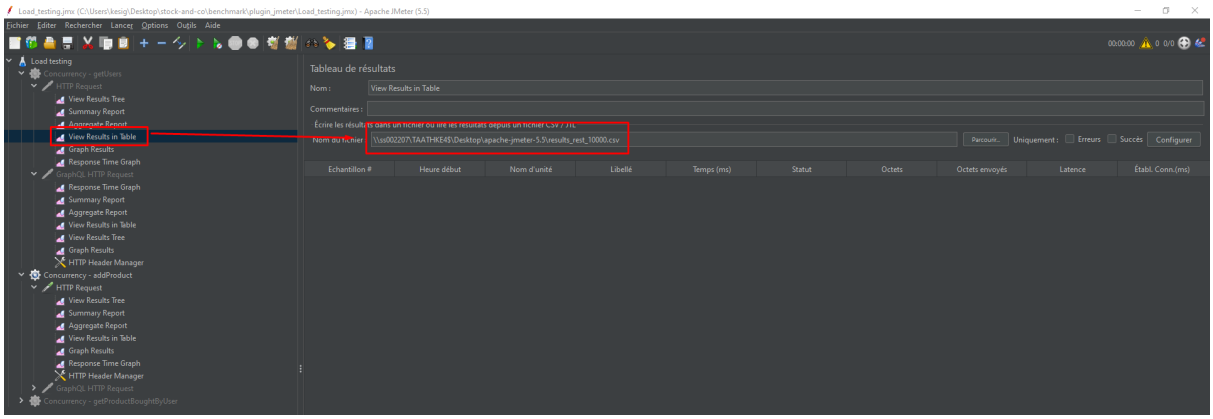

FIGURE 9.7 – Chemin d'accès JMeter

9.7 Graphiques et analyses

La liste des graphiques du chapitre 7 se trouve dans Benchmark/results/dashboard.html. Le code source se trouve dans le même dossier dans le fichier dashboard.Rmd.

Les résultats de JMeter ont été compressés dans un fichier .Rds. Si le dashboard doit être regénéré, téléchargez le fichier df_jm.Rds depuis le lien ci-dessous et placez le dans le dossier Benchmark/results.

df_jm.Rds: `https://www.dropbox.com/s/vgzxhrq269htrl3/df_jm.rds?dl=0`

Bibliographie

[1] 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, November 2018. ISSN: 2471-9269. vi, 14, 15

[2] Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-Driven RESTful Service Composition. In E. Michael Maximilien, Gustavo Rossi, Soe-Tsyr Yuan, Heiko Ludwig, and Marcelo Fantinato, editors, *Service-Oriented Computing*, volume 6568, pages 111–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science. 26

[3] Marek Bolanowski, Kamil Żak, Andrzej Paszkiewicz, Maria Ganzha, Marcin Paprzycki, Piotr Sowiński, Ignacio Lacalle, and Carlos E. Palau. Eficiency of REST and gRPC Realizing Communication Tasks in Microservice-Based Ecosystems. In Hamido Fujita, Yutaka Watanobe, and Takuya Azumi, editors, *Frontiers in Artificial Intelligence and Applications*. IOS Press, September 2022. vii, 50

[4] Martine Brasseur and Fatine Biaz. L'impact de la digitalisation des organisations sur le rapport au travail : entre aliénation et émancipation. *Question(s) de management*, 21(2):143–155, 2018. Place: Caen Publisher: EMS Editions. 2

[5] Angular Core Razor Pages and Anthony Giretti. Beginning grpc with asp .net core 6. vii, 34, 35, 36, 37, 39, 41, 49

[6] Francisco Curbera, William Nagy, and Sanjiva Weerawarana. Web Services: Why and How. December 2001. 8

[7] Asit Dan, Robert D. Johnson, and Tony Carrato. SOA service reuse by design. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 25–28, Leipzig Germany, May 2008. ACM. 12

[8] Thomas Erl. *Service-oriented architecture: analysis and design for services and microservices*. Prentice Hall, 2016. vi, 10

[9] Bob Familiar. What Is a Microservice ? In Bob Familiar, editor, *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*, pages 9–19. Apress, Berkeley, CA, 2015. 11

[10] Robin Flygare and Anthon Holmqvist. Performance characteristics between monolithic and microservice-based systems, 2017.

[11] Florian Gerlinghoff. Vergleich von Introspected REST mit alternativen Ansätzen für die Entwicklung von Web-APIs hinsichtlich Performance, Evolvierbarkeit und Komplexität. May 2021.

[12] Dewi Ayu Hartina, Armin Lawi, and Benny Leonard Enrico Panggabean. Performance Analysis of GraphQL and RESTful in SIM LP2M of the Hasanuddin University. In *2018 2nd East Indonesia Conference on Computer and Information Technology (EIConCIT)*, pages 237–240, November 2018. vii, 50

[13] Łukasz Kamiński, Maciej Kozłowski, Daniel Sporysz, Katarzyna Wolska, Patryk Zaniewski, and Radosław Roszczyk. Comparative review of selected Internet communication protocols, December 2022. arXiv:2212.07475 [cs]. vii, 50, 51

[14] Benjamin Mueller, Goetz Viering, Christine Legner, and Gerold Riempp. Understanding the Economic Potential of Service-Oriented Architecture. *Journal of Management Information Systems*, 26(4):145–180, April 2010. 9

[15] San Murugesan. Web Application Development: Challenges And The Role Of Web Engineering. In Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 7–32. Springer, London, 2008. 5, 6

[16] Brad A. Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, May 2016. 24

[17] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media, 2019. 9

[18] Naghmeh Niknejad, Waidah Ismail, Imran Ghani, Behzad Nazari, Mahadi Bahari, and Ab Razak Bin Che Hussin. Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation. *Information Systems*, 91:101491, July 2020. 9, 10, 12, 13

[19] Eve Porcello and Alex Banks. *Learning GraphQL: declarative data fetching for modern web apps*. " O'Reilly Media, Inc.", 2018. 28, 29, 32, 33, 46, 49

[20] Sascha Preibisch. *API Development*. Springer, 2018. 23, 24

[21] Mark J Price. *Apps and Services with .NET 7: Build practical projects with Blazor, .NET MAUI, gRPC, GraphQL, and other enterprise technologies*. Packt Publishing, Birmingham, England, 2022.

[22] Vinay Raj and Sadam Ravichandra. A service graph based extraction of microservices from monolith services of service-oriented architecture. *Software: Practice and Experience*, 52(7):1661–1678, 2022. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3081. 8

[23] Chris Richardson. *Microservices patterns: with examples in Java*. Simon and Schuster, 2018. 9, 12, 13, 14

[24] Bernd Ruecker. *Practical Process Automation*. " O'Reilly Media, Inc.", 2021. 17, 18, 19, 21

[25] F. Sazanavets. *Microservices Communication in .NET Using gRPC: A practical guide for .NET developers to build efficient communication mechanism for distributed apps*. Packt Publishing, 2022. 34

[26] Oliver Schmid, Agnes Lisowska Masson, and Béat Hirsbrunner. Real-time collaboration through web applications: an introduction to the Toolkit for

Web-based Interactive Collaborative Environments (TWICE). *Personal and Ubiquitous Computing*, 18(5):1201–1211, June 2014. 2

[27] Shakirat Sulyman. Client-Server Model. *IOSR Journal of Computer Engineering*, 16:57–71, January 2014. 4

[28] Sri Lakshmi Vadlamani, Benjamin Emdon, Joshua Arts, and Olga Baysal. Can GraphQL Replace REST ? A Study of Their Efficiency and Viability. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, pages 10–17, June 2021. vii, 50

[29] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in practice: Hypermedia and systems architecture.* " O'Reilly Media, Inc.", 2010. 24, 25, 49

[30] Luis Weir. *Enterprise API Management: Design and deliver valuable business APIs.* Packt Publishing Ltd, 2019. 39, 46, 47, 48, 49

[31] Tetiana Yarygina and Anya Helene Bagge. Overcoming Security Challenges in Microservice Architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 11–20, March 2018. 14

[32] Mariusz Śliwa and Beata Pańczyk. Performance comparison of programming interfaces on the example of REST API, GraphQL and gRPC. *Journal of Computer Sciences Institute*, 21:356–361, December 2021. 50

Sites Web

[33] 1. Introduction to gRPC - gRPC: Up and Running [Book]. https://www.oreilly.com/library/view/grpc-up-and/9781492058328/ch01.html (dernière consultation le Février 11, 2023). vii, 45

[34] API Documentation & Design Tools for Teams | Swagger. https://swagger.io/ (dernière consultation le Février 11, 2023).

[35] API Showdown: REST vs. GraphQL vs. gRPC – Which Should You Use ? https://www.infoq.com/podcasts/api-showdown-rest-graphql-grpc/ (dernière consultation le Novembre 19, 2022). 39, 40, 41

[36] Best tools and methods for designing RESTful APIs | TechTarget. https://www.techtarget.com/searchapparchitecture/tip/Best-tools-and-methods-for-designing-RESTful-APIs (dernière consultation le Mars 01, 2023). 46

[37] Chapter 1 - Reliable, Scalable and Maintainable Applications. https://timilearning.com/posts/ddia/part-one/chapter-1/ (dernière consultation le Novembre 11, 2022). 7

[38] Core concepts, architecture and lifecycle. Section: docs. 47

[39] A deep dive into caching REST APIs. https://stellate.co/blog/deep-dive-into-caching-rest-apis (dernière consultation le Mars 02, 2023). 48

[40] GraphQL | A query language for your API. https://graphql.org/ (dernière consultation le Janvier 04, 2023). 29

[41] GraphQL & Caching: The Elephant in the Room. https://www.apollographql.com/blog/backend/caching/graphql-caching-the-elephant-in-the-room/ (dernière consultation le Mars 02, 2023). 48

[42] GraphQL Best Practices | GraphQL. https://graphql.org/learn/best-practices/ (dernière consultation le Mars 02, 2023). 49

[43] gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design. https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them (dernière consultation le Janvier 05, 2023). 35

[44] How to Improve Application Reliability by Using ButterCMS. https://buttercms.com/blog/improve-application-reliability/ (dernière consultation le Novembre 11, 2022). 7

[45] Importance Of Code Reusability In Software Development. https://browserstack.wpengine.com/guide/importance-of-code-reusability/ (dernière consultation le Février 24, 2023). 11

[46] Introducing GraphiQL. https://www.gatsbyjs.com/docs/how-to/querying-data/running-queries-with-graphiql/ (dernière consultation le Février 11, 2023). 42

[47] Introduction to Event-driven Architectures With RabbitMQ | Theodo. https://blog.theodo.com/2019/08/event-driven-architectures-rabbitmq/ (dernière consultation le Décembre 30, 2022).

[48] Introduction to the gRPC-Gateway. https://grpc-ecosystem.github.io/grpc-gateway/docs/tutorials/introduction/ (dernière consultation le Février 11, 2023). vii, 45

[49] La naissance du web | CERN. https://home.cern/fr/science/computing/birth-web (dernière consultation le Novembre 16, 2022). 5

[50] Maintainability Guide · Jens Oliver Meiert. https://meiert.com/en/blog/maintainability-guide/ (dernière consultation le Février 24, 2023). 11

[51] Microservices Architecture | Microservices Solutions. https://www.chakray.com/expertise/microservices/ (dernière consultation le Février 11, 2023). vi, 9, 11

[52] Microservices Orchestration vs Choreography: What should you prefer ? https://www.accionlabs.com/microservices-orchestration-vs-choreography-what-to-prefer (dernière consultation le Février 11, 2023). 16, 17

[53] Microservices vs monolithic architecture: How each of them can impact your business ? https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (dernière consultation le Novembre 21, 2022). 9

[54] Monolithic vs microservice architecture: Which is best ? | DigitalOcean. https://www.digitalocean.com/blog/monolithic-vs-microservice-architecture (dernière consultation le Février 11, 2023). 16

[55] Orchestration vs Choreography - Which is better ? [U+2753]. https://www.wallarm.com/what/orchestration-vs-choreography (dernière consultation le Janvier 30, 2023). 22

[56] Orchestration vs Choreography, which one should you use ? The pros and cons. https://www.wallarm.com/what/orchestration-vs-choreography (dernière consultation le Janvier 30, 2023). 22

[57] Project-specific Swagger Documentation - CUBA Platform. Developer's Manual. https://doc.cuba-platform.com/manual-7.0/rest_swagger.html (dernière consultation le Février 11, 2023).

[58] SOA vs. Microservices: A Head-to-Head Comparison | Scout APM Blog. https://scoutapm.com/blog/soa-vs-microservices (dernière consultation le Février 11, 2023). 12, 16

[59] Testing Strategies in Monolithic vs Microservices Architecture. https://browserstack.wpengine.com/guide/testing-strategies-in-microservices-vs-monolithic-applications/ (dernière consultation le Février 11, 2023).

[60] To GraphQL or not to GraphQL ? Pros and Cons. https://slicknode.com/blog/graphql-or-not-graphql-pros-and-cons (dernière consultation le Février 11, 2023). 42

[61] Tutorial | Lighthouse. https://lighthouse-php.com/tutorial/ (dernière consultation le Février 11, 2023). vii, 43

[62] Understanding RPC, REST and GraphQL | APIs You Won't Hate. https://apisyouwonthate.com/blog/understanding-rpc-rest-and-graphql (dernière consultation le Janvier 05, 2023). 34

[63] Web Application | Definition, History, Development, Examples, Uses, & Facts | Britannica. https://www.britannica.com/topic/Web-application (dernière consultation le Novembre 16, 2022). 5, 6

[64] Web application vs. website: finally answered. https://www.scnsoft.com/blog/web-application-vs-website-finally-answered (dernière consultation le Janvier 11, 2023). 4

[65] What Is A Workflow Engine ? (With Six Key Benefits). https://in.indeed.com/career-advice/career-development/workflow-engine (dernière consultation le Février 11, 2023).

[66] What is Web Application (Web Apps) and its Benefits. https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app (dernière consultation le Janvier 29, 2023). 4, 6

[67] When to use GraphQL, gRPC, REST, and Webhooks. https://fauna.com/blog/when-to-use-graphql-grpc-rest-webhooks (dernière consultation le Novembre 16, 2022).

[68] Why You Should Disable GraphQL Introspection In Production – GraphQL Security. https://www.apollographql.com/blog/graphql/security/why-you-should-disable-graphql-introspection-in-production/ (dernière consultation le Janvier 12, 2023). 44

[69] A Lack of API Documentation Considered Harmful, May 2014. https://blog.carbonfive.com/a-lack-of-api-documentation-considered-harmful/ (dernière consultation le Mars 01, 2023). 46

[70] The benefits of using web-based applications | Geeks Insights, December 2019. https://www.geeks.ltd.uk/insights/blog/the-benefits-of-using-web-based-applications (dernière consultation le Janvier 29, 2023). 6

[71] Part 1: RabbitMQ for beginners - What is RabbitMQ ?, September 2019. https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html (dernière consultation le Février 26, 2023). vi, 18, 19

[72] Securing the Microservices Architecture: Decomposing the Monolith Without Compromising Security, March 2019. https://securityintelligence.com/securing-the-microservices-architecture-decomposing-the-monolith-without-compromi (dernière consultation le Février 11, 2023).

[73] We've done SOA, why should we care about Microservices ?, April 2019. https://www.axoniq.io/blog/soa-versus-microservices (dernière consultation le Février 11, 2023). 8

[74] How to Use Google's Protocol Buffers in Python, May 2020. https://www.freecodecamp.org/news/googles-protocol-buffers-in-python/ (dernière consultation le Février 11, 2023). vii, 36

[75] Monolithic vs Microservices architecture, March 2020. Section: Computer Subject. 13

[76] Qu'est-ce qu'un serveur web ? - Apprendre le développement web | MDN, July 2020. https://developer.mozilla.org/fr/docs/Learn/Common_questions/Web_mechanics/What_is_a_web_server (dernière consultation le Janvier 25, 2023). 4

[77] Digitization and Simplification, January 2021. https://www.bcg.com/publications/2014/digitization-simplification-getting-best-both (dernière consultation le Janvier 20, 2023). 2

[78] SOA vs microservices: going beyond the monolith, October 2021. https://circleci.com/blog/soa-vs-microservices/ (dernière consultation le Novembre 30, 2022). 12

[79] SOA vs. Microservices: What's the Difference ?, May 2021. https://www.ibm.com/cloud/blog/soa-vs-microservices (dernière consultation le Janvier 30, 2023). 9, 12

[80] Web Application Architecture Importance for Business, September 2021. Section: Blog. 2

[81] API Composition Toolkit | WaveMaker Docs, November 2022. https://www.wavemaker.com/learn/app-development/services/java-services/api-composer-toolkit/ (dernière consultation le Mars 01, 2023). 47

[82] AsyncAPI Initiative for event-driven APIs, March 2023. https://www.asyncapi.com/ (dernière consultation le Mars 23, 2023). 19

[83] Client–server model, February 2023. https://en.wikipedia.org/w/index.php?title=Client%E2%80%93server_model&oldid=1138040571 (dernière consultation le Février 11, 2023). vi, 5

[84] Introduction to mesh, January 2023. https://the-guild.dev/graphql/mesh/docs (dernière consultation le Mars 01, 2023). 47

[85] Marwen Abid. Four REST API Versioning Strategies, August 2019. https://www.xmatters.com/blog/blog-four-rest-api-versioning-strategies/ (dernière consultation le Mars 02, 2023). 49

[86] European Central Bank. Digitalisation and its impact on the economy: insights from a survey of large companies, November 2018. https://www.ecb.europa.eu/pub/economic-bulletin/focus/2018/html/ecb.ebbox201807_04.en.html (dernière consultation le Février 11, 2023). 2

[87] James Brannan. How to Expose API in Java: Expose Your Application With a REST API | RapidAPI, June 2020. https://rapidapi.com/blog/expose-api-with-java/ (dernière consultation le Février 11, 2023).

[88] Thomas Bush. Which Is More Secure: Monolith or Microservices ? | Nordic APIs |, September 2020. https://nordicapis.com/which-is-more-secure-monolith-or-microservices/ (dernière consultation le Février 25, 2023). 14

[89] Luca Christen. Using HATEOAS with REST APIs, April 2021. https://engineering.3ap.ch/post/using-hateoas-with-rest/ (dernière consultation le Mars 01, 2023). 46

[90] Marc DiPasquale. AsyncAPI Code Generation: Microservices Using Spring Cloud Stream, April 2020. https://solace.com/blog/asyncapi-codegen-microservices-using-spring-cloud-stream/ (dernière consultation le Mars 23, 2023). vi, 20, 21

[91] Ashley Dotterweich. The design principles behind scalable web apps, August 2022. https://mattermost.com/blog/design-principles-for-scalable-web-apps/ (dernière consultation le Février 24, 2023). 11

[92] Lokesh Gupta. REST Architectural Constraints, May 2018. https://restfulapi.net/rest-architectural-constraints/ (dernière consultation le Février 11, 2023). 26

[93] Sanjay K. 5 Web Application Testing Challenges | GlowTouch, March 2021. https://www.glowtouch.com/5-web-application-testing-challenges/ (dernière consultation le Novembre 11, 2022). 7

[94] Charley Mann. Testing Entire Process Paths, October 2020. https://camunda.com/blog/2020/10/testing-entire-process-paths/ (dernière consultation le Février 11, 2023). vi, 18

[95] peter czibik. Swagger for Node.js HTTP API Design, August 2015. https://blog.risingstack.com/swagger-nodejs/ (dernière consultation le Mars 08, 2023). vii, 42

[96] Team rédac. Workflow : qu'est-ce que c'est ?, March 2022. https://datascientest.com/workflow-tout-savoir (dernière consultation le Février 11, 2023). 16

[97] Lior Shalom. The Principles of Planning and Implementing Microservices, September 2020. https://medium.com/swlh/the-principles-of-planning-and-implementing-microservices-3cb0eb76c172 (dernière consultation le Février 11, 2023). 13

[98] Appexive IT Solutions. Top 5 Key Roles of Web Application Development Become Vital for Businesses | Appexive. https://appexive.com/blog/top-5-key-roles-of-web-application-development-become-vital-for-businesses (dernière consultation le Janvier 21, 2023). 2

[99] Temporal. Message-driven Microservice Orchestration with Serverless Workflow and AsyncAPI — AsyncAPI Conf, November 2021. https://www.youtube.com/watch?v=GmSMTrR6sEE (dernière consultation le Mars 23, 2023). 19

[100] Alyssa Walker. SOA vs Microservices – Difference Between Them, March 2020. https://www.guru99.com/microservices-vs-soa.html (dernière consultation le Février 11, 2023). 16

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Faculté des sciences économiques et sociales
Wirtschafts- und sozialwissenschaftliche Fakultät
Boulevard de Pérolles 90
CH-1700 Fribourg

D E C L A R A T I O N

Par ma signature, j'atteste avoir rédigé personnellement ce travail écrit et n'avoir utilisé que les sources et moyens autorisés, et mentionné comme telles les citations et paraphrases.

J'ai pris connaissance de la décision du Conseil de Faculté du 09.11.2004 l'autorisant à me retirer le titre conféré sur la base du présent travail dans le cas où ma déclaration ne correspondrait pas à la vérité.

De plus, je déclare que ce travail ou des parties qui le composent, n'ont encore jamais été soumis sous cette forme comme épreuve à valider, conformément à la décision du Conseil de Faculté du 18.11.2013.

……, le ..... 20.....

.....
(signature)