Université De Montpellier - Faculté Des Sciences
Année Universitaire 2023 - 2024
M1 Génie Logiciel
HAI823I : TER De Master 1

MONITORING DES ARCHITECTURES À BASE DE MICROSERVICES : TÉLÉMÉTRIE ET RECONFIGURATION

Jeudi 2 mai 2024

Étudiants : :

Mohammed DAFAOUI
Adam DAIA
Aness RABIA

Encadrant :

Seriai-Abdelhak
Rima-Bachar

Table des matières

1 Monitoring des architectures : Contexte et état de l'art 3

1.1 Introduction et présentation du problème . . . 3

1.1.1 Architecture monolithique . . . 3

1.1.2 Architecture microservices . . . 4

1.1.3 Objectif du Projet . . . 4

1.2 Background et contexte . . . 5

1.2.1 Architecture monolithique . . . 5

1.2.2 Architecture microservices . . . 6

1.2.3 Observabilité et telemetrie . . . 6

1.2.4 Qualité de service et architectures logicielles . . . 8

1.2.5 Modèle ISO/IEC 25010 pour la qualité des produits logiciels . . . 8

1.2.6 Impact des architectures sur la qualité . . . 9

1.3 État de l'art . . . 9

2 Outils 14

2.1 OpenTelemetry . . . 14

2.2 Postman . . . 17

2.3 Artillery . . . 17

2.4 Optimisation et Monitoring de K6 . . . 18

3 Monitoring 19

3.1 Architecture monolithique de l'application . . . 19

3.1.1 Structure de l'application . . . 19

3.1.2 Implémentation de l'architecture . . . 20

3.2 Migration vers une architecture microservices . . . 21

3.3 Mise en place de la télémétrie . . . 22

3.3.1 Instrumentation avec OpenTelemetry . . . 22

3.3.2 Configuration de la collecte et de l'exportation . . . 23

3.3.3 Visualisation et analyse . . . 23

3.3.4 Tests préliminaires avec Postman . . . 24

3.3.5 Tests de charge avec Artillery . . . 24

3.3.6 Optimisation et monitoring avec K6 . . . 24

3.4 Comparaison et interprétation des résultats . . . 24

3.4.1 Tests avec 20 vus pendant 30 secondes . . . 24

3.4.2 Tests avec 60 vus pendant 30 secondes 25
3.4.3 Détails des requêtes pour 60 vues pendant 30 secondes 26
3.4.4 Tests avec 80 vues pendant 30 secondes 26
3.4.5 Tests avec 100 vus pendant 50 secondes 27
3.4.6 Tests avec 1 vu pendant 30 secondes 28
3.5 Conclusion sur les résultats 29
3.6 Réconfiguration : Quelques techniques 30
3.6.1 Ajout d'un API Gateway 30
3.6.2 Ajout d'un Load Balancer 30

4 Conclusion **31**

1 Monitoring des architectures : Contexte et état de l'art

1.1 Introduction et présentation du problème

Le sujet aborde le domaine du monitoring des architectures basées sur les microservices, en se concentrant particulièrement sur les aspects de télémétrie et de reconfiguration. Les microservices, une approche architecturale moderne, permettent de développer des applications comme une collection de petits services autonomes et spécialisés, chacun fonctionnant indépendamment [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.]. Le projet vise à mettre en œuvre une architecture microservices pour une application logicielle, en utilisant des outils d'observabilité comme OpenTelemetry pour surveiller et potentiellement améliorer les caractéristiques non fonctionnelles telles que les performances et la disponibilité. OpenTelemetry permet la collecte de données sur les performances, erreurs, et autres métriques importantes pour l'optimisation de l'architecture.

1.1.1 Architecture monolithique

Les applications monolithiques sont des systèmes logiciels construits comme une seule et unique unité. Toutes les fonctionnalités de l'application sont regroupées dans un seul et même programme, qui est déployé et mis à l'échelle dans son ensemble [Fowler, M. (2022). Monolith to Microservices : Evolutionary Patterns for Transforming Legacy Applications. O'Reilly Media, Inc.].
Dans une architecture monolithique, l'application est conçue comme un bloc unique, où les différentes couches sont étroitement liées et interdépendantes. Toute modification nécessite de recompiler et redéployer l'ensemble du système.

Les architectures monolithiques présentent des avantages comme une mise en place plus simple et une gestion plus aisée des transactions. Cependant, elles comportent également des inconvénients, notamment une scalabilité limitée, une évolutivité difficile et une résilience moindre.
Bien que largement utilisées, notamment pour des applications de taille modeste, les architectures à base de microservices gagnent en popularité grâce à leur plus grande flexibilité et leur capacité à s'adapter aux besoins évolutifs des entreprises.

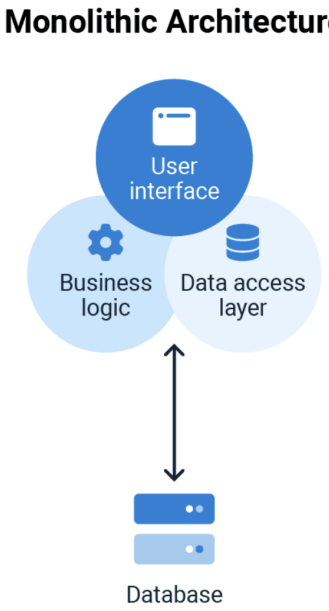


FIGURE 1 – Architecture monolithique

1.1.2 Architecture microservices

Les microservices sont une approche architecturale consistant à développer une application sous forme de petits services autonomes, indépendants, et spécialisés [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.].

Les architectures à base de microservices sont des systèmes logiciels construits en regroupant ces services, chaque service étant développé, déployé et mis à l'échelle de manière indépendante.

Les microservices offrent une scalabilité plus facile, une meilleure résilience, une plus grande flexibilité dans le déploiement, la possibilité de technologie polyglotte, une évolutivité aisée et une amélioration de la continuité de service.
Les architectures à base de microservices apportent une agilité et une flexibilité indéniables, mais cela vient avec le défi de contrôler des caractéristiques non fonctionnelles comme les performances, la sécurité et la disponibilité.

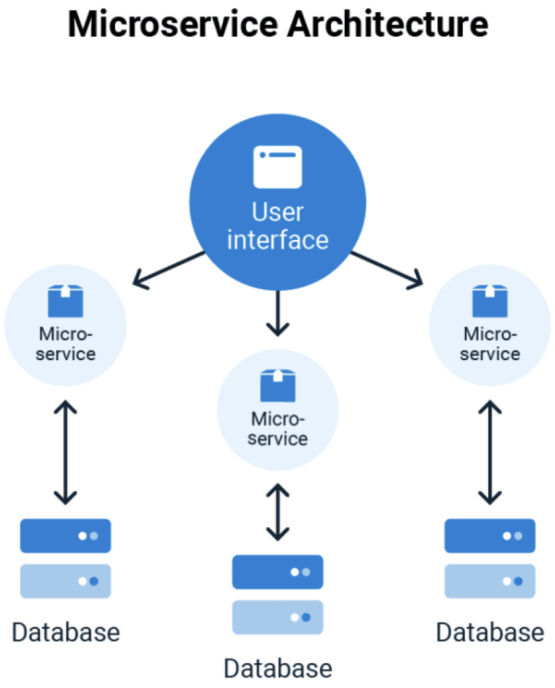
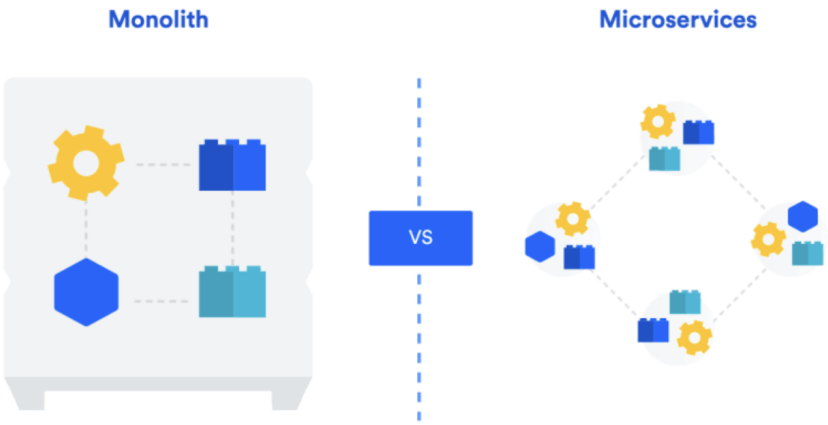


FIGURE 2 – Architecture microservice

1.1.3 Objectif du Projet

Ce projet vise à implémenter une architecture à base de microservices pour une application logicielle, en mettant en place une approche d'observabilité pour surveiller les caractéristiques non fonctionnelles (performances, disponibilité, etc.). En utilisant des outils comme OpenTelemetry, l'objectif est de mettre en œuvre un système de monitoring permettant de reconfigurer l'architecture pour améliorer ces caractéristiques, incluant l'ajout d'un API Gateway, des Load Balancers, etc.

1.2 Background et contexte



1.2.1 Architecture monolithique

Les applications monolithiques sont des systèmes logiciels construits comme une seule et unique unité, où toutes les fonctionnalités sont regroupées dans un même programme [Fowler, M. (2022). Monolith to Microservices : Evolutionary Patterns for Transforming Legacy Applications. O'Reilly Media, Inc.]. Dans cette architecture, l'application est conçue comme un bloc unique, avec des couches étroitement liées et interdépendantes (interface utilisateur, logique métier, accès aux données, etc.). Avec une architecture monolithique, toute modification ou évolution de l'application nécessite de recompiler et redéployer l'ensemble du système. Cela peut s'avérer complexe et coûteux, surtout pour des applications de grande taille et de plus en plus complexes.

Les principaux avantages des architectures monolithiques sont :

— Mise en place plus simple et rapide
— Gestion plus aisée des transactions et des interactions entre les différentes parties de l'application
— Déploiement et configuration plus simples

Bien que les architectures monolithiques présentent certains avantages en termes de simplicité de mise en place et de gestion des transactions, elles comportent également des inconvénients significatifs qu'il est important de prendre en compte. La nature même d'un bloc unique et indivisible rend ces architectures difficiles à faire évoluer et à mettre à l'échelle de manière optimale. De plus, une défaillance dans une partie de l'application peut impacter l'ensemble du système, réduisant ainsi sa résilience [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.].

Les principaux inconvénients des architectures monolithiques incluent :

— Scalabilité limitée : il est difficile de mettre à l'échelle sélectivement certaines parties de l'application
— Évolutivité difficile : toute modification nécessite de recompiler et redéployer l'ensemble du système
— Résilience moindre : une défaillance dans une partie de l'application peut impacter tout le système

Bien que largement utilisées, notamment pour des applications de taille modeste, les architectures monolithiques deviennent de plus en plus difficiles à maintenir et à faire évoluer face à la complexité croissante des systèmes modernes. C'est dans ce contexte que les architectures à base de microservices ont émergé comme une alternative prometteuse [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.].

1.2.2 Architecture microservices

Les microservices sont une approche architecturale consistant à développer une application sous forme de petits services autonomes, indépendants, et spécialisés [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.]. Les architectures à base de microservices sont des systèmes logiciels construits en regroupant ces services, chaque service étant développé, déployé et mis à l'échelle de manière indépendante. Contrairement à une architecture monolithique où l'application est conçue comme un bloc unique avec des couches étroitement liées, une architecture microservices divise l'application en une série de services déployables indépendamment qui communiquent via des API bien définies. Chaque service implémente une fonctionnalité unique dans un contexte borné.

Les principaux avantages des architectures microservices sont :

— Scalabilité plus facile : les services peuvent être mis à l'échelle de manière indépendante
— Meilleure résilience : une défaillance dans un service n'impacte pas l'ensemble de l'application
— Flexibilité dans le déploiement : les services peuvent être déployés indépendamment
— Possibilité de technologie polyglotte : les services n'ont pas besoin de partager les mêmes technologies
— Évolutivité aisée : il est plus facile d'ajouter de nouvelles fonctionnalités en ajoutant de nouveaux services
— Amélioration de la continuité de service : les services peuvent être mis à jour sans impacter l'application entière.

Bien que les architectures microservices apportent de nombreux avantages en termes de scalabilité, de résilience et de flexibilité, elles présentent également des défis et des inconvénients à prendre en compte [Newman, S. (2015). Building Microservices : Designing Fine-Grained Systems. O'Reilly Media, Inc.]. La complexité inhérente à la distribution des services et à leur communication peut compliquer la gestion des caractéristiques non fonctionnelles comme les performances, la sécurité et la disponibilité du système. De plus, la mise en place des mécanismes nécessaires pour assurer le bon fonctionnement de l'architecture (découverte de services, routage, orchestration, etc.) ajoute une couche de complexité supplémentaire.

Les principaux inconvénients des architectures microservices incluent :

— Contrôle des caractéristiques non fonctionnelles (performances, sécurité, disponibilité) face à la complexité croissante des interactions entre services
— Gestion de la complexité liée à la distribution des services et à leur communication
— Nécessité de mettre en place des mécanismes de découverte de services, de routage, de sécurité et de résilience
— Difficulté de garantir la cohérence des données réparties entre les différents services
— Complexité accrue du déploiement et de l'orchestration des services
— Besoin de mettre en place des outils de monitoring et de diagnostic pour surveiller le fonctionnement global du système.

1.2.3 Observabilité et telemetrie

L'observabilité se rapporte à la compréhension de l'état interne d'un système par l'examen de ses sorties externes, en particulier ses données. Dans le contexte du développement d'applications modernes, l'observabilité fait référence à la collecte et à l'analyse de données (logs, indicateurs et traces) provenant d'une grande variété de sources, dans le but de fournir des informations détaillées sur le comportement des applications qui s'exécutent dans nos environnements. Elle peut s'appliquer à n'importe quel système que nous avons créé et que nous souhaitons monitorer.

L'observabilité est importante, car elle permet aux équipes d'évaluer, monitorer et améliorer les performances des systèmes informatiques distribués. Elle est bien plus efficace que les méthodes de monitoring traditionnelles. Une plateforme d'observabilité de bout en bout peut aider à décloisonner et à favoriser la collaboration. Nous pouvons diagnostiquer, analyser et remonter à l'origine des problèmes de façon proactive.

L'observabilité aide les organisations à :

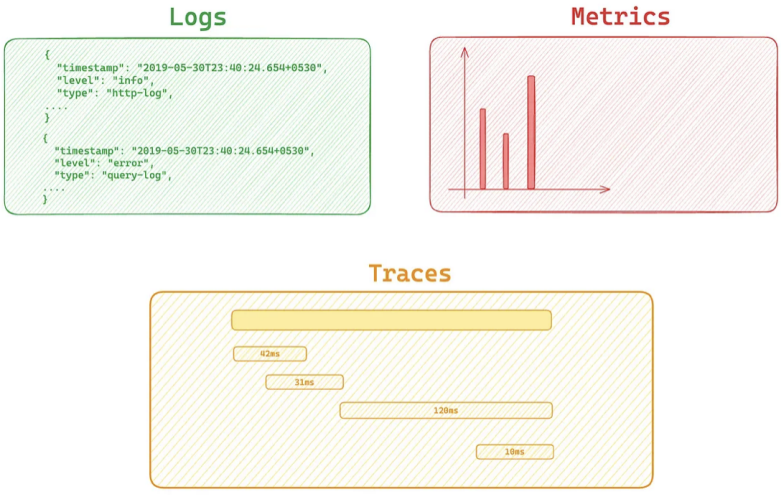
— découvrir et analyser la signification des incidents de performance pour leur entreprise ;
— améliorer l'efficacité des cycles de vie de développement logiciel ;
— accélérer la résolution des problèmes et l'analyse des causes profondes ;
— améliorer l'expérience des utilisateurs finaux ;
— renforcer la sécurité des applications.

Les principales classes de données utilisées dans l'observabilité sont les logs, les métriques et les traces. Ensemble, ils forment ce qu'on appelle souvent « les trois piliers de l'observabilité » :

Logs : un log est un enregistrement texte d'un événement qui s'est produit à un moment donné, accompagné d'un horodatage indiquant quand il s'est produit et de données contexte. Les logs existent sous trois formes : texte brut, structurés et binaires. Le texte brut est le plus courant, mais les logs structurés, qui incluent des données et des métadonnées supplémentaires et sont plus faciles à interroger, deviennent de plus en plus populaires. Dans le cadre de notre système de data observability (observabilité des données), bien souvent, les logs sont aussi le premier endroit où regarder lorsqu'un système ne fonctionne pas comme prévu.

Métriques : une métrique est une valeur numérique mesurée sur un intervalle de temps et qui inclut des attributs spécifiques : horodatage, nom, KPI et valeur, entre autres. Contrairement aux logs, les métriques sont structurées par défaut, ce qui facilite les requêtes et l'optimisation du stockage, permettant de les conserver pendant des périodes plus longues.

Traces : une trace représente le trajet de bout en bout d'une requête à travers un système distribué. Lorsqu'une requête se déplace dans le système hôte, chaque opération effectuée sur celui-ci, appelée « unité logique », est encodée avec des données importantes concernant le microservice qui réalise cette opération. En visualisant les traces, chacune comprenant une ou plusieurs unités logiques, nous pouvons suivre son parcours à travers un système distribué et identifier la cause d'un goulot d'étranglement ou d'une interruption.



1.2.4 Qualité de service et architectures logicielles

La qualité de service (QoS) est un concept clé dans le développement d'applications, qui vise à garantir que le système réponde aux exigences non fonctionnelles des utilisateurs et des parties prenantes. Ces exigences, définies dans des modèles comme ISO 25010, incluent des caractéristiques telles que la performance, la sécurité, la fiabilité, la maintenabilité et la portabilité [ISO/IEC 25010, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011].

Le choix d'une architecture logicielle, qu'il s'agisse d'une approche monolithique ou microservices, a un impact direct sur la capacité à atteindre ces différentes dimensions de qualité. Par exemple, une architecture monolithique peut faciliter la gestion des transactions et la cohérence des données, mais peine à offrir une scalabilité fine et une résilience élevée. À l'inverse, une architecture microservices permet une mise à l'échelle et une évolutivité plus poussées, mais nécessite de relever des défis en termes de performances et de complexité opérationnelle.

Dans les sections suivantes, nous allons examiner en détail comment les architectures monolithiques et microservices se comparent sur les différents aspects de la qualité de service, en nous appuyant sur le modèle ISO 25010 et les caractéristiques non fonctionnelles clés. Cela permettra de mieux comprendre les forces et faiblesses de chaque approche, et d'éclairer le choix architectural le plus adapté en fonction des priorités du projet.

1.2.5 Modèle ISO/IEC 25010 pour la qualité des produits logiciels

Le modèle ISO/IEC 25010 définit un cadre de référence complet pour spécifier, mesurer et évaluer la qualité des produits logiciels tout au long de leur cycle de vie [ISO/IEC 25010 :2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models]. Ce modèle identifie huit caractéristiques clés permettant d'analyser différents aspects du produit ou système, allant de sa capacité à répondre aux besoins des utilisateurs à sa facilité de maintenance et de déploiement dans divers environnements.

Figure 3 – Modèle ISO/IEC 25010

Les huit caractéristiques du modèle ISO/IEC 25010 sont les suivantes :

1. **Pertinence fonctionnelle** : Mesure dans quelle mesure le produit répond aux besoins déclarés et implicites. Inclut la complétude, la correction et l'adéquation fonctionnelles.
2. **Efficacité de performance** : Évalue les performances par rapport aux ressources utilisées. Comprend le comportement temporel, l'utilisation des ressources et la capacité. *Cette caractéristique*

mesure la capacité du système à répondre dans des délais acceptables, à gérer des charges utilisateurs et des volumes de transactions importants.

3. **Compatibilité** : Détermine la capacité à échanger des informations et à fonctionner avec d'autres systèmes. Inclut la coexistence et l'interopérabilité.
4. **Facilité d'utilisation** : Évalue la convivialité pour les utilisateurs en termes d'efficacité, d'efficience et de satisfaction. Inclut la reconnaissabilité, l'exploitabilité, la protection contre les erreurs, l'esthétique et l'accessibilité.
5. **Fiabilité** : Mesure la capacité à fonctionner conformément aux spécifications et à maintenir la performance. Comprend la maturité, la disponibilité, la tolérance aux pannes et la récupérabilité. *Cette caractéristique évalue la capacité du système à fonctionner sans défaillance, à gérer les erreurs, à être remis en état rapidement en cas de panne.*
6. **Sécurité** : Évalue la capacité à protéger les données contre les accès non autorisés. Inclut la confidentialité, l'intégrité, la non-répudiation, la responsabilité et l'authenticité.
7. **Maintenabilité** : Détermine la facilité et l'efficacité des modifications et corrections. Comprend la modularité, la réutilisabilité, l'analysabilité, la modificabilité et la testabilité. *Cette caractéristique mesure la facilité avec laquelle le système peut être modifié et corrigé, en termes de coût et de délai.*
8. **Portabilité** : Évalue la facilité de transfert d'un environnement à un autre. Inclut l'adaptabilité, l'installabilité et le remplaçabilité.

Ce modèle fournit un cadre de référence complet pour définir, mesurer et évaluer la qualité des produits logiciels, et est largement utilisé par les développeurs, acquéreurs et évaluateurs.

1.2.6 Impact des architectures sur la qualité

Le choix d'une architecture logicielle, monolithique ou microservices, a un impact direct sur la capacité à atteindre les différentes dimensions de qualité, fonctionnelles et non fonctionnelles :

— Une architecture monolithique facilite la gestion des transactions et la cohérence des données (caractéristiques fonctionnelles), mais peine à offrir une scalabilité fine et une résilience élevée (performances, fiabilité).
— Une architecture microservices permet une mise à l'échelle et une évolutivité plus poussées (performances, maintenabilité), mais nécessite de relever des défis en termes de complexité opérationnelle et de cohérence des données réparties.

Par exemple, ajouter de nouvelles fonctionnalités dans un monolithe peut impacter les performances globales, tandis que dans une architecture microservices, cela se limite au service concerné. Inversement, assurer la cohérence des données est plus complexe avec une architecture distribuée. En résumé, le choix architectural est un arbitrage entre différentes dimensions de qualité. Une approche monolithique favorise la facilité de développement et la cohérence fonctionnelle, tandis que les microservices apportent plus de flexibilité et d'évolutivité, au prix d'une complexité accrue.

1.3 État de l'art

Avant d'avancer notre TER, nous avons entrepris une analyse approfondie de plusieurs documents pertinents dans le domaine des architectures monolithiques et microservices. Ces articles ont été soigneusement sélectionnés pour fournir une vue d'ensemble complète des performances, des défis et des avantages associés à chaque architecture. Ces documents nous ont fourni une base solide pour comprendre les métriques, les défis et les meilleures pratiques associés aux architectures monolithiques et microservices. Ils ont éclairé notre compréhension des aspects à analyser et à observer dans le cadre de notre projet, ainsi que des stratégies de gestion que nous prévoyons d'adopter.

Microservice Patterns

Auteurs :Chris Richardson
With examples in Java. Il a été publié en 2019 par Manning Publications.

Dans notre démarche de migration, nous nous sommes également appuyés sur les stratégies décrites dans le chapitre 13 du livre *Microservice Patterns* de Chris Richardson, qui explore les différentes approches pour passer d'une architecture monolithique à une architecture microservices. Ce chapitre présente les principales stratégies de migration :

1. **La création progressive d'une "strangler application"** : de nouveaux microservices sont développés pour remplacer progressivement les fonctionnalités du monolithe. L'objectif est de livrer de la valeur commerciale régulièrement tout en minimisant les modifications apportées au monolithe.
2. **La décomposition progressive du monolithe en microservices** : en identifiant des tranches verticales de fonctionnalités à extraire vers des services autonomes. Cette approche est plus complexe car elle nécessite de diviser le modèle de domaine et de résoudre les problèmes de dépendances.
3. **L'implémentation de nouvelles fonctionnalités en tant que services distincts** : plutôt que de les ajouter au monolithe existant. Cela permet d'accélérer le développement, les tests et le déploiement des nouvelles fonctionnalités.
4. **L'extraction progressive de fonctionnalités du monolithe** : pour les transformer en services autonomes, afin d'améliorer la modularité, l'évolutivité et la maintenabilité.

Ces différentes stratégies nous ont guidés dans notre réflexion sur la meilleure approche à adopter pour migrer notre application monolithique vers une architecture microservices.

Voici quelques résumés des autres documents que nous avons examinés pour orienter notre recherche et comprendre les tendances actuelles dans le domaine.

Evaluating the Performance of Monolithic and Microservices Architectures in an Edge Computing Environment

Auteurs : Nitin Rathore, Anand Rajavat
Date de publication : Il a été publié dans le "International Journal of Fog Computing", Volume 5, Issue 1, en 2022.

Ce document évalue l'efficacité des architectures de microservices par rapport aux architectures monolithiques dans les applications IoT, en se concentrant particulièrement sur les environnements de calcul en périphérie (edge computing). Il propose une architecture de microservices utilisant des technologies telles qu'Apache Jena pour la gestion des données, MQTT pour la messagerie, Docker pour la conteneurisation, et WEKA pour le traitement des données. Un prototype a été développé et testé dans les domaines de la santé, de la maison intelligente et du bureau intelligent, pour démontrer la flexibilité et l'adaptabilité de cette approche.

Les performances des deux architectures sont comparées en termes de temps de réponse, de débit, et d'utilisation des ressources. Les résultats montrent des améliorations significatives dans tous ces aspects avec l'architecture de microservices, soulignant ainsi sa supériorité en termes d'efficacité opérationnelle dans des environnements exigeant une réponse rapide et une gestion efficace des ressources. Ces améliorations sont particulièrement pertinentes pour les applications IoT où la rapidité et l'efficacité sont cruciales pour le traitement des données en temps réel.

Monolithic vs. Microservice Architecture : A Performance and Scalability

Auteurs : Grzegorz Blinowski, Anna Ojdowska, Adam Przybylek
Date de publication : 18 février 2022.

Cette étude approfondit la comparaison entre les architectures monolithiques et les microservices dans le contexte d'Azure App Service, mettant en œuvre des technologies Java et .NET pour développer et tester les applications. Les résultats mettent en lumière que, bien que les microservices offrent une modularité et une adaptabilité accrues, dans un environnement contraint à une seule machine, les architectures monolithiques démontrent une supériorité notable. Cela s'explique principalement par les surcoûts générés par le transfert de requêtes entre les composants des microservices, qui peuvent impacter négativement la performance globale.

L'étude révèle que sur des configurations à ressource unique, les systèmes monolithiques optimisent mieux l'utilisation des ressources, évitant les latences additionnelles engendrées par la communication inter-services des microservices. Cependant, il est crucial de noter que la décision entre choisir une architecture monolithique ou microservices ne doit pas seulement se baser sur la performance brute. Elle doit également prendre en compte les besoins spécifiques de l'application en termes de scalabilité, de maintenance et de développement futur, ainsi que la nature de la charge de travail prévue.

En définitive, cette recherche éclaire les décideurs techniques sur le choix de l'architecture la plus adaptée selon le contexte spécifique de leur application, soulignant que les microservices, bien que performants sous plusieurs aspects, peuvent ne pas être la solution optimale dans des contextes de ressources limitées ou de charges de travail non distribuées.

Stepwise Migration of a Monolith to a Microservices Architecture : Performance and Migration Effort Evaluation

Auteurs : Diogo Faustino, Nuno Gonçalves, Manuel Portela, António Rito Silva
Date de publication : Janvier 2022

Ce document explore en détail la migration progressive d'un système monolithique vers une architecture de microservices, mettant un accent particulier sur une étape intermédiaire de modularisation du monolithe. L'étude analyse minutieusement les performances comparatives des deux architectures—monolithique et basée sur les microservices—et aborde les défis spécifiques liés à la cohérence des données durant la transition. En utilisant une approche de modularisation comme étape intermédiaire, le document identifie les avantages de cette stratégie pour faciliter la migration complète vers les microservices.

Les résultats montrent que la modularisation intermédiaire peut réduire l'effort de migration et préparer le terrain pour une transition plus fluide vers les microservices. Toutefois, la décomposition augmente la complexité de gestion des interfaces et de la communication entre les services, ce qui peut impacter la performance globale. L'analyse révèle que bien que les microservices offrent une scalabilité et une flexibilité accrues, ils introduisent également des surcoûts en termes de performances dues aux communications inter-services et à la gestion décentralisée des données.

En conclusion, le document offre un cadre précieux pour les architectes logiciels et les gestionnaires de projet envisageant la migration d'architectures monolithiques vers des microservices, en soulignant l'importance d'une étape de modularisation pour mitiger les risques et les coûts associés à cette transformation numérique.

Assessing the impacts of decomposing a monolithic application for microservices

Auteurs : Tulio Ricardo Hoppen Barzotto, Kleinner Farias
Date de publication : 25 mars 2022. Le langage de programmation utilisés dans l'étude incluent Java.

Cette étude présente un cadre d'évaluation rigoureux, basé sur des métriques quantitatives, pour examiner les impacts de la transformation d'applications monolithiques en architectures de microservices. Focalisée sur une opération spécifique de retrait effectuée par une institution financière, l'analyse détaillée aborde les changements dans la modularité du logiciel, ainsi que dans les consommations de CPU et de mémoire avant et après la migration vers les microservices.

Le document explore comment la décomposition en microservices peut résoudre des problèmes inhérents aux structures monolithiques, telles que la difficulté de maintenance, la complexité accrue et les coûts élevés de mise à l'échelle. En utilisant des métriques ciblées, l'étude révèle une amélioration significative de la modularité du logiciel, illustrant une meilleure séparation des responsabilités et une réduction des dépendances complexes qui caractérisent souvent les architectures monolithiques. Ces changements se traduisent par une maintenance plus facile et une évolutivité améliorée.

Quant aux performances, les résultats montrent une diminution de la consommation de CPU et de mémoire, soutenant l'argument que les microservices peuvent mener à une utilisation plus efficace des ressources. Cela est particulièrement pertinent dans des contextes où la performance et la réactivité des systèmes sont critiques.

En conclusion, l'étude fournit des preuves empiriques que la migration vers des microservices peut non seulement améliorer l'efficacité opérationnelle, mais aussi optimiser les coûts liés aux ressources matérielles. Ce cadre d'évaluation sert donc d'outil précieux pour les décideurs en TI, leur permettant de baser leurs stratégies de transformation numérique sur des données solides et des analyses détaillées.

From monolithic systems to Microservices : An assessment framework

Auteurs : Florian Auer, Valentina Lenarduzzi, Michael Felderer, Davide Taibi
Date de publication : Avril 2021

Ce travail académique propose un cadre d'évaluation robuste et basé sur des données empiriques, spécifiquement conçu pour assister les entreprises dans la prise de décision concernant la migration de systèmes monolithiques vers des architectures de microservices. Ce cadre se concentre sur l'analyse détaillée de diverses caractéristiques et métriques essentielles, telles que le temps de réponse des applications, l'utilisation du CPU, et le volume de requêtes traitées par unité de temps (minute ou seconde). L'objectif est de fournir aux décideurs une méthode systématique pour évaluer si la transition vers les microservices améliorera la performance et l'efficacité opérationnelle de leur système informatique.

Le document détaille la méthode utilisée pour développer ce cadre d'évaluation, qui inclut des entretiens avec des professionnels de l'informatique et l'analyse de cas d'utilisation réels. En se fondant sur la théorie ancrée, les auteurs ont identifié un ensemble de métriques pertinentes qui permettent une évaluation objective des avantages et des inconvénients de la migration vers les microservices. Ces métriques incluent, mais ne sont pas limitées à, la modularité du système, la capacité de montée en charge, et la tolérance aux pannes.

Le document insiste sur l'importance de ne pas se baser uniquement sur des "impressions intestinales" ou des tendances du marché, mais plutôt sur une approche rigoureuse et factuelle. Pour les entreprises envisageant la migration, le cadre propose une série d'étapes structurées pour collecter et analyser les données nécessaires, permettant ainsi de prendre une décision éclairée. En évaluant minutieusement chaque aspect du système existant et en le comparant aux potentialités offertes par les microservices, les entreprises peuvent mieux juger de la pertinence de cette transition technologique.

En conclusion, ce cadre d'évaluation offre une ressource précieuse pour les architectes logiciels, les gestionnaires de projet et les décideurs techniques, leur fournissant les outils nécessaires pour naviguer dans le paysage complexe et en évolution rapide des architectures de systèmes. Il met l'accent sur une approche mesurée et orientée données pour s'assurer que les transitions vers les microservices sont non seulement techniquement viables, mais aussi alignées avec les objectifs stratégiques à long terme de l'entreprise.

Does migrating a monolithic system to microservices decrease the technical debt ?

Auteurs : Valentina Lenarduzzi, Francesco Lomio, Nyyti Saarimäki, Davide Taibi
Date de publication : juillet 2020.

Ce document examine l'impact de la migration d'un système monolithique vers des microservices sur la dette technique. Utilisant des outils d'analyse de code tels que SonarQube et des entretiens qualitatifs avec les membres de l'équipe, l'étude évalue l'évolution de la dette technique avant et après la migration. Les résultats montrent que, malgré une augmentation initiale de la dette technique due au développement des nouveaux microservices, celle-ci a tendance à croître plus lentement qu'avec le système monolithique sur le long terme. Cette étude fournit une preuve empirique que, bien que la migration initiale puisse augmenter la dette technique due aux coûts de développement et d'intégration, les avantages à long terme en matière de maintenabilité et de flexibilité du système justifient cette transition pour les entreprises visant à améliorer leurs pratiques de développement logiciel.

Conclusion de l'état de l'art

L'analyse approfondie de ces différents documents scientifiques nous a permis de mieux comprendre les enjeux liés aux architectures monolithiques et microservices. Nous avons pu identifier les principaux critères de comparaison, tels que les performances, la scalabilité, la maintenabilité et les efforts de migration. Les études montrent que le choix entre les deux approches dépend fortement des besoins spécifiques de l'application et de son contexte d'utilisation .

Les architectures monolithiques présentent l'avantage d'une mise en œuvre plus simple et d'une gestion des transactions facilitée, mais peinent à offrir une scalabilité fine et une évolutivité aisée. À l'inverse, les microservices permettent une mise à l'échelle et une évolutivité plus poussées, au prix d'une complexité accrue en termes d'opérations et de cohérence des données réparties .

Ces enseignements tirés de l'état de l'art vont nous guider dans la suite de notre projet, en nous aidant à définir une méthodologie d'évaluation pertinente et à interpréter lMisees résultats obtenus. Ils nous permettront également de formuler des recommandations éclairées sur le choix architectural le plus adapté en fonction des contraintes et des priorités du projet.

2 Outils

2.1 OpenTelemetry

OpenTelemetry est une solution open source qui prend en charge l'intégralité du processus de télémétrie, de l'extraction des données jusqu'à leur stockage final. Cette solution présente trois avantages majeurs qui, combinés, en font une option extrêmement puissante :

— **Tout-en-un** : OpenTelemetry gère toutes les étapes du processus de télémétrie, éliminant ainsi le besoin d'outils tiers.
— **Universel** : Compatible avec tous les langages et backends.
— **Infiniment personnalisable** : Adaptable à absolument tous les besoins spécifiques.

FIGURE 4 – OpenTelemetry

Avant la création d'OpenTelemetry, deux principaux projets dominaient le domaine de la télémétrie :

OpenTracing : Développé par la CNCF (Cloud Native Computing Foundation), OpenTracing offrait une bibliothèque de spécifications très flexible, pouvant être implémentée dans n'importe quel langage. Cependant, son implémentation demandait souvent un travail de développement considérable ainsi que l'intégration de composants externes.

OpenCensus : Développé par Google, OpenCensus proposait un ensemble d'outils et de frameworks très complets, capables de prendre en charge le processus de télémétrie d'une application de A à Z. Néanmoins, ces outils étaient spécifiques à certains langages et ne répondaient pas toujours aux besoins les plus précis.

FIGURE 5 – OpenTracing + OpenCensus = OpenTelemetry

Ces deux projets présentaient des approches différentes : OpenTracing offrait une grande flexibilité mais était trop minimaliste, tandis qu'OpenCensus était très puissant mais manquait de flexibilité. Pour combiner les avantages des deux solutions, la CNCF et Google ont décidé de fusionner leurs projets respectifs, aboutissant à la création d'OpenTelemetry.

OpenTelemetry se compose de plusieurs éléments clés :

Spécification/API

La spécification est un ensemble de règles permettant d'uniformiser la manière dont OpenTelemetry est implémenté, assurant ainsi une cohérence du processus de télémétrie, quel que soit le système ou le langage utilisé. Il s'agit du niveau d'abstraction le plus élevé, consistant en une documentation et une API soigneusement élaborées pour garantir les mêmes fonctionnalités et un même niveau d'extensibilité dans toutes les implémentations.

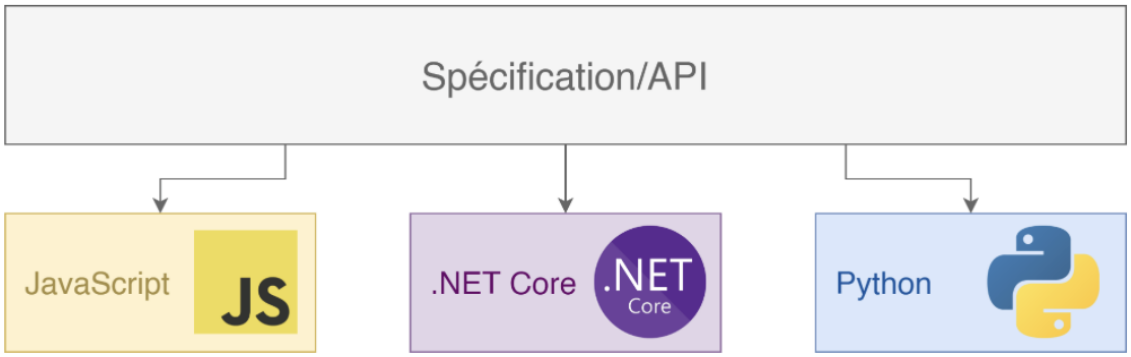


FIGURE 6 – Specifications

SDK

Les SDK sont des implémentations concrètes d'OpenTelemetry développées par le CNCF (le créateur) et la communauté, afin de fournir une base de départ robuste. Ces SDK contiennent divers outils qui, une fois assemblés, prennent en charge l'ensemble du processus de télémétrie. Les principaux composants incluent :

— **Traceur/Activité** : Instrumentalise le code et extrait les données.
— **Processeurs** : Traite les données avant transmission.
— **Échantillonneur** : Filtre les données à extraire.
— **Exportateur** : Transmet les données vers le backend.

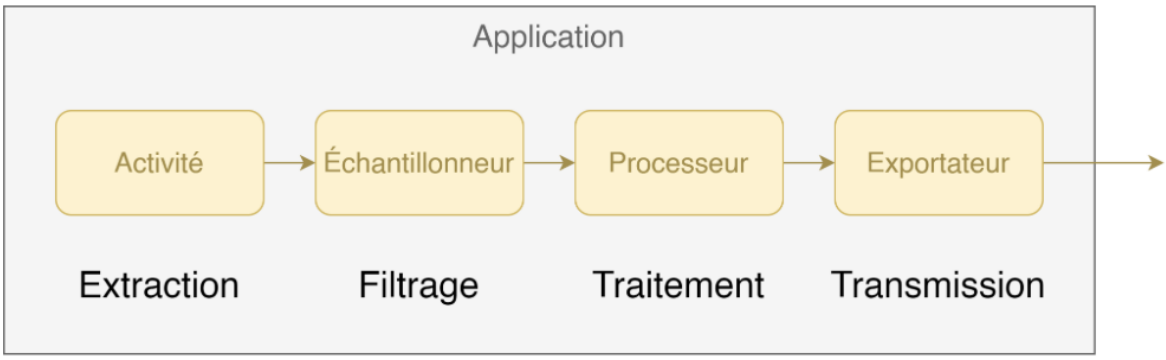


FIGURE 7 – SDK

La grande force de ces SDK réside dans leur conformité rigoureuse à la spécification, permettant ainsi une extension, modification ou substitution aisée de ces éléments pour créer un système sur mesure, sans nécessiter des procédures complexes. Tant que la spécification est respectée, toutes les pièces s'emboîtent comme des LEGO.

Collecteur

Le collecteur est une composante centrale qui tourne sur l'un de nos serveurs, traitant les données après transmission pour ensuite les réexporter vers le backend. Il est particulièrement utile pour filtrer, agréger ou regrouper les données. Bien que son utilisation ne soit pas obligatoire, il est bon de savoir qu'il existe.

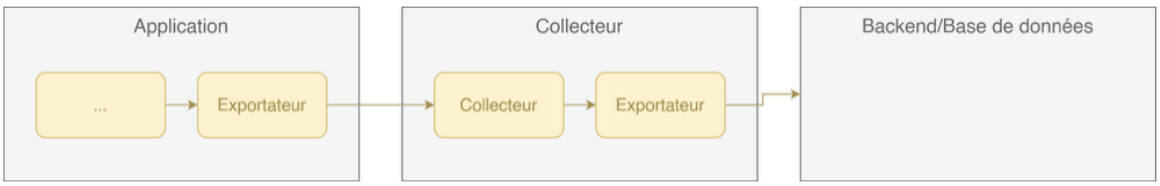


FIGURE 8 – Collecteur

Dans notre projet, nous avons intégré OpenTelemetry pour la collecte de données de télémétrie. Pour commencer, nous avons utilisé l'instrumentation automatique d'OpenTelemetry, ce qui a ajouté des points d'instrumentation à notre code sans nécessiter de modifications manuelles importantes. Cela nous permet de capturer automatiquement des informations telles que les traces, les mesures et les journaux à partir de nos applications sans avoir à écrire de code spécifique à chaque fois.

Pour la gestion des traces, nous avons choisi Jaeger comme fournisseur. Ce système open source de traçabilité distribuée nous permet de visualiser et d'analyser les traces générées par nos applications. En utilisant Jaeger avec OpenTelemetry, nous avons pu obtenir une vision détaillée du comportement de nos services et identifier rapidement les problèmes de performance ou les goulots d'étranglement dans notre architecture distribuée.

Pour la collecte et la visualisation des métriques, nous avons opté pour Prometheus. Ce système de surveillance open source nous permet de collecter des métriques à partir de nos applications et de notre infrastructure. En intégrant Prometheus avec OpenTelemetry, nous avons pu exporter les métriques enregistrées vers Prometheus, où elles sont stockées et visualisées à l'aide de Grafana.

Enfin, pour l'affichage et l'analyse des données de télémétrie, nous avons utilisé Grafana. Cette plate-forme open source de visualisation et d'analyse des données nous a permis de créer des tableaux de bord personnalisés pour surveiller les performances de nos applications et identifier les tendances ou anomalies potentielles.

En résumé, l'intégration d'OpenTelemetry avec une instrumentation automatique, des fournisseurs tels que Jaeger et Prometheus, et l'utilisation de Grafana pour l'affichage des données nous a permis de mettre en place une infrastructure complète de collecte, de surveillance et d'analyse des données de télémétrie sur notre projet. Cela a amélioré la performance et la fiabilité de nos applications.

2.2 Postman

Après avoir mis en place OpenTelemetry pour la collecte de données de télémétrie, nous avons élargi nos efforts pour inclure des outils de test de charge et de performance dans notre processus de développement. Dans un premier temps, nous avons utilisé Postman pour effectuer des tests d'API et de services individuels. Postman nous a permis de valider le bon fonctionnement de nos endpoints et de nos microservices en simulant des requêtes et en vérifiant les réponses, ce qui était crucial pour assurer la qualité de nos applications.

2.3 Artillery

Cependant, alors que notre application évoluait et que nos exigences en matière de performance devenaient plus complexes, nous avons décidé d'intégrer Artillery à notre processus de test. Artillery est un outil open source de test de charge et de performance qui nous a permis de simuler des charges de travail réalistes et de mesurer la réactivité et la stabilité de nos applications sous différentes conditions de charge.

En utilisant Artillery en conjonction avec OpenTelemetry, nous avons pu non seulement évaluer les performances de nos applications du point de vue des utilisateurs finaux, mais aussi collecter des données de télémétrie détaillées pendant les tests de charge. Cela nous a permis de comprendre le comportement de nos applications sous stress et d'identifier les éventuels problèmes de performance ou de mise à l'échelle avant qu'ils ne deviennent des problèmes critiques en production.

Pour lancer les tests de charge avec Artillery, nous avons utilisé un fichier de configuration YAML qui définissait les scénarios de test, les charges de travail à simuler et les métriques à collecter. Voici un exemple de code utilisé pour tester la création de produits dans notre application monolithique :

```yaml
config:
  target: "http://localhost:8001"
  phases:
    - duration:  30
      arrivalRate: 1
  defaults:
    headers:
      Content-Type: "application/json"
scenarios:
  - flow:
      - post:
          url: "/product/create"
          json:
            name: "Kesar Mango"
            desc: "grande qualité de mangue"
            type: "fruits"
            unit: 1
            price: 170
            available: true
            suplier: "Golden seed firming"
      - get:
          url: "/category/fruits"
```

On peut le lancer directement dans le terminal en utilisant un script similaire :

```
artillery run batteryTest.yml
```

2.4 Optimisation et Monitoring de K6

Lors d'une conférence organisée par le CNCF (créateur de OpenTelemetry) et Crédit Agricole, nous avons eu l'opportunité d'approfondir nos connaissances sur OpenTelemetry et son intégration avec d'autres outils de développement. L'événement, intitulé "Optimisation et Monitoring de K6 à OpenTelemetry", s'est tenu le jeudi 18 avril 2024. Après avoir assisté à la conférence sur l'optimisation et le monitoring de K6 avec OpenTelemetry, nous avons décidé d'intégrer K6 à notre suite d'outils de test de charge et de performance. K6 est un outil open source puissant et moderne qui nous permet de simuler des charges de travail réalistes et de mesurer les performances de nos applications avec précision.

Pour commencer, nous avons utilisé K6 pour créer des scénarios de test de charge qui reflétaient de manière réaliste le comportement attendu de nos utilisateurs. Nous avons défini des scénarios pour simuler des actions telles que la navigation sur notre application, l'envoi de formulaires, et la récupération de données à partir de nos services. En utilisant le langage de script simple mais expressif de K6, nous avons pu définir des scénarios complexes et réalistes en quelques lignes de code. En intégrant K6 avec OpenTelemetry, nous avons pu collecter des données de télémétrie détaillées pendant nos tests de charge. OpenTelemetry nous a permis de capturer des métriques précises sur les performances de nos applications, telles que les temps de réponse, les taux d'erreur et les latences, et de les exporter vers des systèmes de surveillance tels que Prometheus et Grafana pour une analyse ultérieure.

En utilisant K6 avec OpenTelemetry, nous avons pu obtenir une vision approfondie des performances de nos applications sous différentes charges de travail. Cela nous a permis d'identifier les goulets d'étranglement et les points faibles de nos systèmes, et de prendre des mesures pour les optimiser et les améliorer. En fin de compte, l'intégration de K6 avec OpenTelemetry a joué un rôle crucial dans l'amélioration de la fiabilité et de la performance de nos applications, en nous permettant de détecter et de résoudre les problèmes de performance dès les premières étapes du développement.

Voici un exemple de script :

Example script

```js
import http from "k6/http";
import { check, sleep } from "k6";

// Test configuration
export const options = {
  thresholds: {
    // Assert that 99% of requests finish within 3000ms.
    http_req_duration: ["p(99) < 3000"],
  },
  // Ramp the number of virtual users up and down
  stages: [
    { duration: "30s", target: 15 },
    { duration: "1m", target: 15 },
    { duration: "20s", target: 0 },
  ],
};

// Simulated user behavior
export default function () {
  let res = http.get("https://test-api.k6.io/public/crocodiles/1/");
  // Validate response status
  check(res, { "status was 200": (r) => r.status == 200 });
  sleep(1);
}
```

On peut le lancer directement dans le terminal en utilisant un script similaire :

```
k6 run --vus 10 --duration 30s loadtest.js
```

3 Mise en place du Monitoring d'architecture

3.1 Architecture monolithique de l'application

3.1.1 Structure de l'application

Sur GitHub, nous avons trouvé une application Node.js monolithique prête à l'emploi. Nous avons donc travailler dessus pour la transformer en une application à micro-services. Cette application est une plateforme de commande en ligne d'aliments. Étant déjà construite dans une architecture monolithique, nous prévoyons de la décomposer en micro-services. Ses exigences fonctionnelles comprennent :

Liste des produits disponibles Inscription/Connexion de l'utilisateur Ajout de produits au panier Ajout de produits à la liste de souhaits Passer une commande Visualiser les commandes

Cette application monolithique offre une gamme complète de fonctionnalités pour une plateforme de commerce électronique, comme illustré dans le screen des use cases ci-dessous.

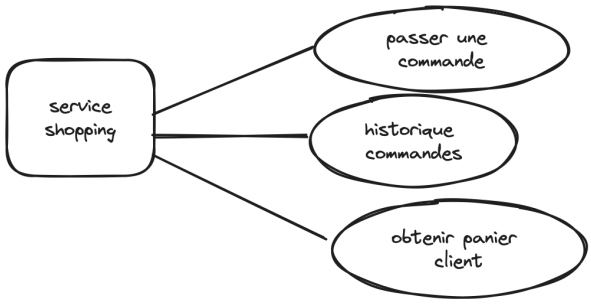


FIGURE 9 – Shopping Use Cases

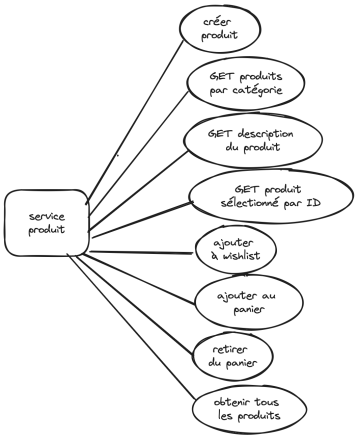


FIGURE 10 – Product Use Cases

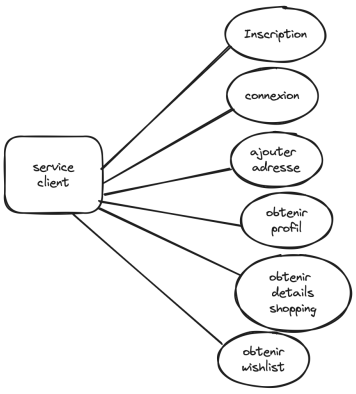


FIGURE 11 – Customer Use Cases

3.1.2 Implémentation de l'architecture

L'application web côté client est accessible aux utilisateurs via une connexion HTTP à notre système backend monolithique. Celui-ci est composé des éléments suivants (voir Figure 12) :

Une base de données MongoDB pour stocker les données des produits, des utilisateurs, des commandes, etc. Un serveur web gérant les requêtes HTTP entrantes.

Différents modules implémentant la logique métier :

— Module Produits : gestion du catalogue produits
— Module Client : gestion des comptes utilisateurs
— Module Panier : gestion du panier d'achat et des commandes

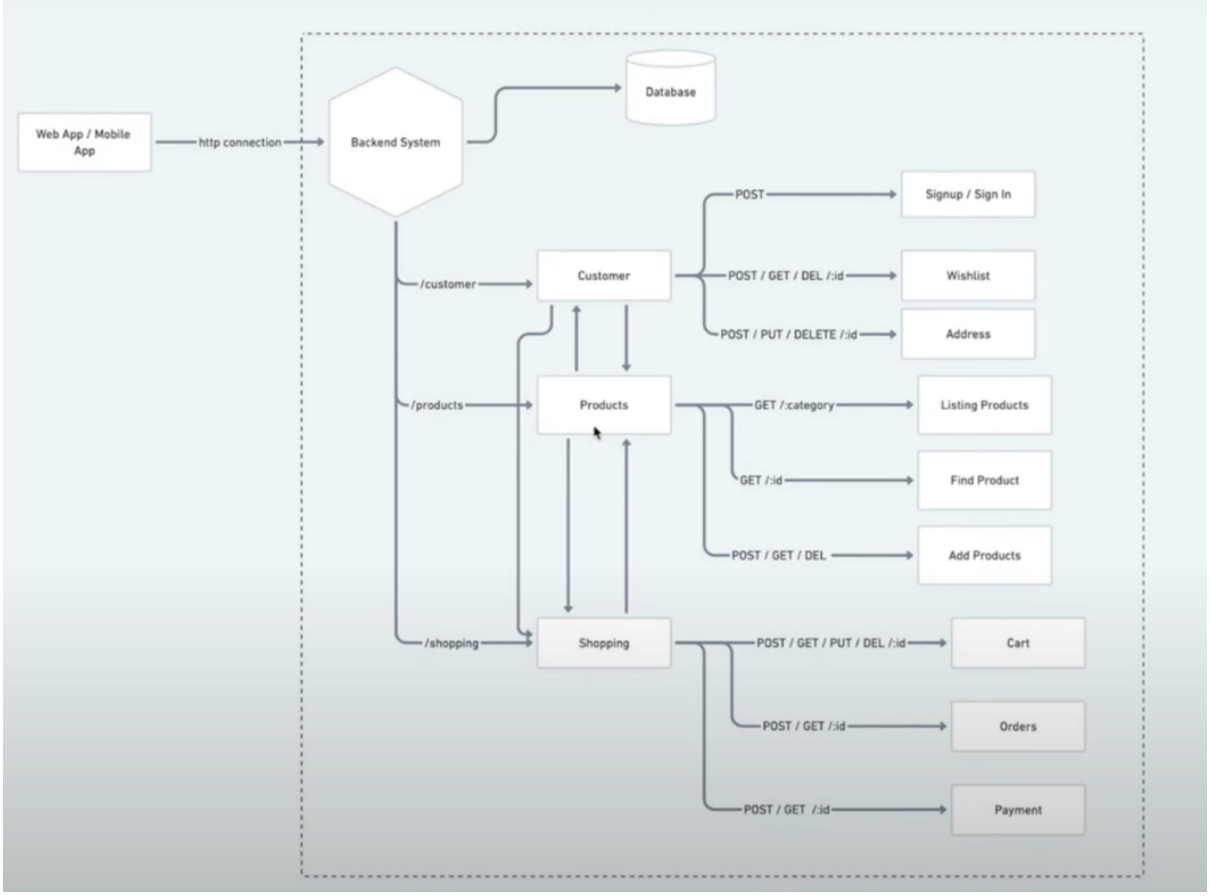


FIGURE 12 – Architecture monolithique de l'application

Bien que cette architecture monolithique présente l'avantage d'une mise en œuvre simple, elle comporte des limites en termes de scalabilité et de maintenabilité. En effet, tous les modules sont fortement couplés et interdépendants. Toute modification ou évolution de l'application nécessite de redéployer l'ensemble du monolithe.

Dans la section suivante, nous détaillerons comment nous avons migré cette application monolithique vers une architecture microservices, afin de résoudre ces défis et tirer parti des avantages des microservices en termes de flexibilité, d'évolutivité et de maintenabilité.

3.2 Migration vers une architecture microservices

Comme nous l'avons vu dans la section d'état de l'art, le chapitre 13 du livre "Microservice Patterns" de Chris Richardson présente plusieurs stratégies clés pour migrer d'une architecture monolithique vers une architecture microservices.Dans le cadre de notre projet, nous avons retenu la stratégie de "décomposition progressive du monolithe en microservices".

Cette approche nous a permis d'identifier les différentes fonctionnalités métier de notre application (gestion des produits, des clients, du panier, etc.) et de les extraire progressivement du monolithe pour les transformer en microservices autonomes. Bien que plus complexe que certaines autres stratégies, cette décomposition verticale du monolithe nous a offert plusieurs avantages :

— Une meilleure modularité et maintenabilité, en découplant les différentes fonctionnalités dans des services indépendants
— Une plus grande flexibilité technologique, chaque microservice pouvant utiliser la stack la plus adaptée
— Une scalabilité améliorée, en permettant de mettre à l'échelle uniquement les services les plus sollicités

Nous avons ainsi migré notre application monolithique vers une nouvelle architecture microservices, qui se compose des éléments suivants (voir Figure 13) :

— Une base de données MongoDB distribuée, avec une instance dédiée pour chaque microservice.
— Plusieurs microservices indépendants, chacun responsable d'une fonctionnalité métier spécifique :
  • Service Produits : gestion du catalogue produits
  • Service Client : gestion des comptes utilisateurs
  • Service Panier : gestion du panier d'achat et des commandes
— Une passerelle API (API Gateway) qui fait office de point d'entrée unique pour les clients et redirige les requêtes vers les microservices appropriés
— Un système de messagerie asynchrone (par exemple, RabbitMQ) pour permettre la communication entre les microservices et gérer les événements

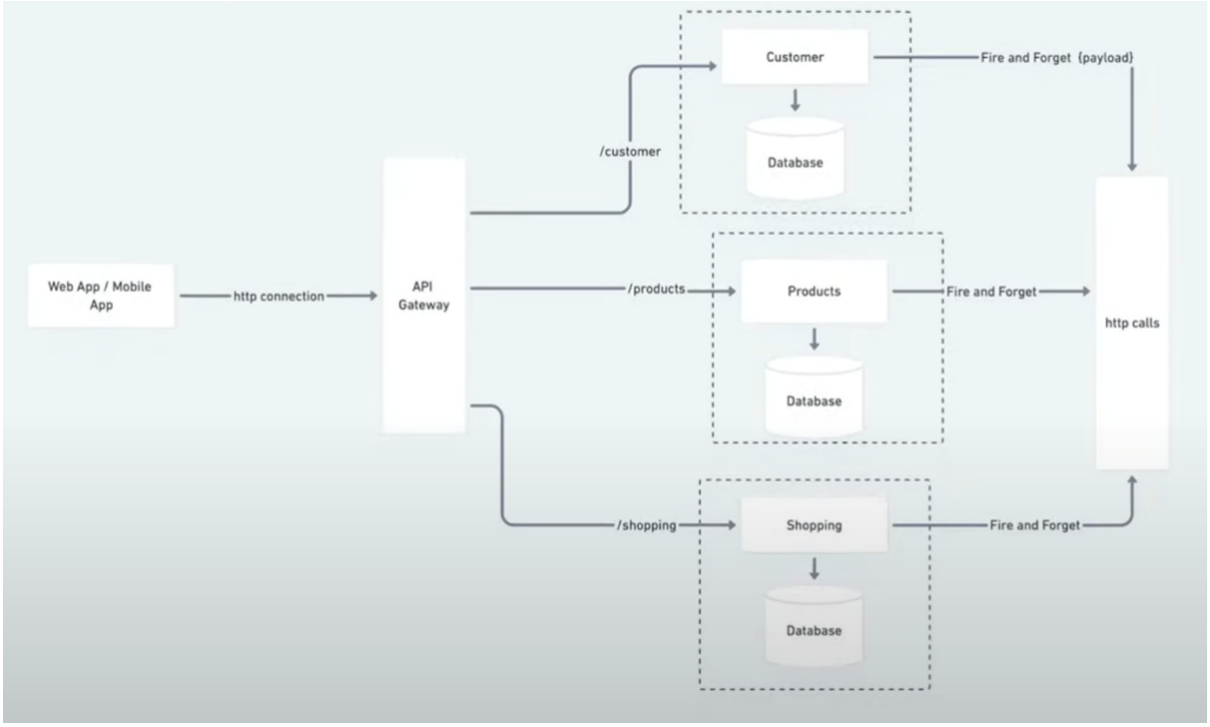


Figure 13 – Architecture microservices de l'application

Dans l'architecture monolithique actuelle, l'application web côté client est accédée par les utilisateurs via une connexion HTTP au système backend. Ce backend est composé d'une base de données MongoDB, avec une hiérarchie de services comprenant :

— Les clients
— Les produits
— Le panier

Chaque service a ses propres opérations et composants. Une caractéristique essentielle de cette architecture est l'interdépendance entre les services : le service Client dépend du service Produits, et vice versa. Cette interconnexion peut devenir un défi en termes de maintenance et d'évolutivité.

Cette nouvelle architecture microservices offre plusieurs avantages :

— Scalabilité améliorée : chaque microservice peut être mis à l'échelle indépendamment en fonction des besoins
— Maintenabilité facilitée : les modifications et les mises à jour peuvent être effectuées de manière isolée sur chaque microservice
— Flexibilité technologique : chaque microservice peut utiliser la technologie la plus adaptée à ses besoins spécifiques
— Résilience accrue : en cas de défaillance d'un microservice, les autres peuvent continuer à fonctionner

Cependant, cette architecture introduit également de nouveaux défis, tels que la gestion de la cohérence des données réparties, la complexité de l'orchestration et de la surveillance des microservices.

3.3 Mise en place de la télémétrie

3.3.1 Instrumentation avec OpenTelemetry

OpenTelemetry est un framework open source qui permet de collecter de manière standardisée des données de télémétrie (traces, métriques, logs) à partir d'applications distribuées. Nous avons intégré OpenTelemetry dans nos deux architectures de la manière suivante :

Architecture monolithique Dans notre application monolithique, nous avons ajouté l'agent OpenTelemetry Java au sein de notre conteneur Docker. Cet agent est responsable de l'instrumentation automatique de notre application Node.js, en capturant les informations sur les requêtes, les appels de méthodes, les erreurs, etc.

Voici un extrait de notre Dockerfile montrant comment nous avons ajouté l'agent OpenTelemetry Java :

```
# Téléchargez et copiez le JAR de l'exportateur Jaeger
ADD https://repo1.maven.org/maven2/io/opentelemetry/opentelemetry-exporter-jaeger/
1.34.1/opentelemetry-exporter-jaeger-1.34.1.jar .

# Téléchargez et copiez l'agent OpenTelemetry Java
ADD https://repo1.maven.org/maven2/io/opentelemetry/javaagent/opentelemetry-javaagent/
1.29.0/opentelemetry-javaagent-1.29.0.jar .

# Spécifiez l'utilisation de l'agent OpenTelemetry Java avec l'exportateur Jaeger
ENV JAVA_TOOL_OPTIONS "-javaagent:./opentelemetry-javaagent-1.29.0.jar
```

Dans cet extrait, nous téléchargeons d'abord le JAR de l'exportateur Jaeger et l'agent OpenTelemetry Java. Ensuite, nous configurons l'option *JAVA TOOL OPTIONS* pour spécifier l'utilisation de l'agent OpenTelemetry Java avec l'exportateur Jaeger lors de l'exécution de notre application Node.js dans le conteneur Docker.

Et voici un extrait de notre fichier Docker Compose montrant la configuration des variables d'environnement pour l'instrumentation OpenTelemetry :

```
services:
  monolith-ter:
    environment:
      - OTEL_SERVICE_NAME=test-ter
      - OTEL_METRICS_EXPORTER=prometheus
      - OTEL_TRACES_EXPORTER=jaeger
      - OTEL_EXPORTER_JAEGER_ENDPOINT=http://jaeger:14250
    # ...
```

Dans cet extrait, nous définissons les variables d'environnement pour configurer l'instrumentation OpenTelemetry dans notre service 'monolith-ter'. Nous spécifions le nom du service, les exportateurs à utiliser (Prometheus pour les métriques et Jaeger pour les traces), ainsi que l'endpoint de l'exportateur Jaeger.

Architecture microservices Pour notre architecture microservices, nous avons déployé un agent OpenTelemetry Java dans chacun de nos microservices. Cela nous permet de collecter de manière indépendante les données de télémétrie de chaque service, tout en conservant une vue d'ensemble grâce à l'orchestration centralisée.

3.3.2 Configuration de la collecte et de l'exportation

Afin de centraliser et d'exploiter les données de télémétrie collectées, nous avons configuré les éléments suivants :

— Un collecteur OpenTelemetry, déployé sous forme de conteneur Docker, qui reçoit les données des agents OpenTelemetry et les redirige vers les systèmes d'analyse.
— Un exportateur Jaeger, qui permet de stocker et de visualiser les traces distribuées.
— Un exportateur Prometheus, qui collecte et agrège les métriques pour une analyse plus approfondie.

La configuration de ces différents composants est gérée via des variables d'environnement dans nos fichiers Docker Compose, assurant ainsi une mise en place homogène de la télémétrie sur l'ensemble de nos architectures.

3.3.3 Visualisation et analyse

Une fois la collecte des données de télémétrie mise en place, nous avons déployé les outils suivants pour visualiser et analyser ces informations :

— Jaeger, pour l'exploration et la visualisation des traces distribuées
— Prometheus, pour l'agrégation et la visualisation des métriques
— Grafana, pour la création de tableaux de bord personnalisés combinant traces, métriques et logs

Cette instrumentation complète nous a permis de disposer d'une vue d'ensemble de l'état de nos applications, facilitant ainsi l'analyse des performances, l'identification des goulots d'étranglement et la résolution des problèmes.

3.3.4 Tests préliminaires avec Postman

Nous avons débuté notre démarche de test en utilisant l'outil Postman. Cela nous a permis de tester individuellement les différents endpoints de notre application, en définissant des scénarios avec des itérations de requêtes. Cette étape initiale nous a aidés à valider le bon fonctionnement de chaque fonctionnalité de manière isolée.

3.3.5 Tests de charge avec Artillery

Après ces tests préliminaires, nous avons ressenti le besoin de simuler des charges plus conséquentes sur notre application. Nous avons donc adopté l'outil Artillery, qui nous a permis d'écrire des scénarios de test plus élaborés. Nous avons notamment fait varier progressivement le taux d'arrivée des requêtes (arrival rate) afin d'observer l'évolution des différentes métriques de performance.

3.3.6 Optimisation et monitoring avec K6

Lors d'une conférence organisée par le Crédit Agricole et Cloud Native Montpellier, nous avons eu l'opportunité d'approfondir nos connaissances sur l'OpenTelemetry et son intégration avec d'autres outils de développement. L'événement, intitulé "Optimisation et Monitoring de K6 à OpenTelemetry", nous a permis de découvrir l'outil K6. Nous avons donc décidé d'adopter K6 pour effectuer des tests de charge supplémentaires, en augmentant également de manière progressive le taux de requêtes. Dans la suite de ce rapport, nous analyserons les résultats obtenus avec ces différents outils de test, en comparant les performances de l'architecture monolithique et de l'architecture microservices.

3.4 Comparaison et interprétation des résultats

Dans le cadre de notre projet, nous avons réalisé une série de tests de performance avec l'outil k6, en ciblant à la fois notre architecture monolithique et notre architecture microservices. Pour ces tests, nous avons exploré diverses vues offertes par k6 afin d'obtenir une analyse approfondie de nos applications.

Nous avons choisi de présenter les résultats du monolithe à gauche et ceux des microservices à droite sur les captures d'écran. Cette disposition nous a permis de faciliter la comparaison entre les deux architectures et d'identifier plus aisément leurs forces et faiblesses respectives.

3.4.1 Tests avec 20 vus pendant 30 secondes

Nous avons effectué des tests de performance avec k6 en utilisant **20 vus pendant 30 secondes**, à la fois sur notre architecture monolithique et sur notre architecture microservices. Les résultats de ces tests initiaux nous ont fourni des informations précieuses.

Nos tests ont montré que le nombre total de requêtes était quasiment le même entre l'architecture monolithique et l'architecture microservices, atteignant environ 1,8k requêtes. Cependant, nous avons observé que les microservices offraient des temps de réponse légèrement plus rapides, avec une moyenne de **206 millisecondes** contre **233 millisecondes** pour le monolithe.

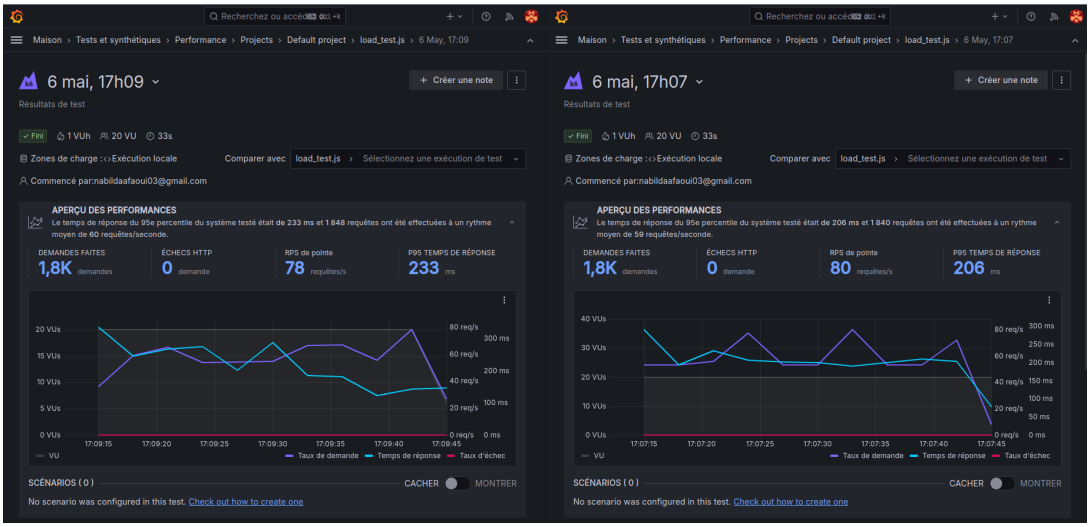


FIGURE 14 – Résultats des tests avec 20 vues pendant 30 secondes

3.4.2 Tests avec 60 vus pendant 30 secondes

Nous avons effectué des tests de performance avec k6 en utilisant **60 vus pendant 30 secondes**, à la fois sur notre architecture monolithique et sur notre architecture microservices. Les résultats de ces tests initiaux nous ont fourni des informations précieuses.

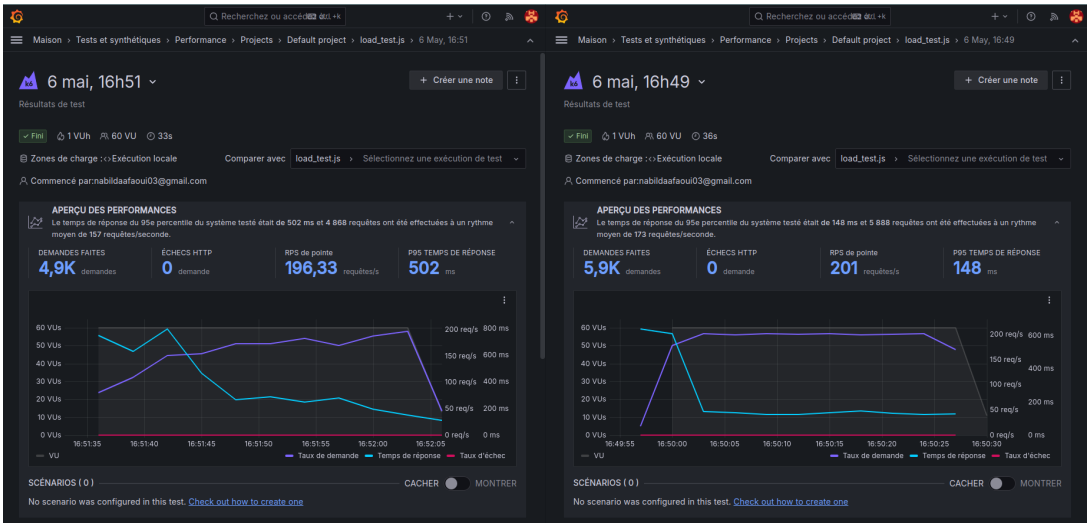


FIGURE 15 – Résultats des tests avec 60 vus pendant 30 secondes

Lors de nos tests suivants, nous avons observé que le nombre total de requêtes était de **4,9k pour l'architecture monolithique** et de **5,9k pour l'architecture microservices**. Cela montre que les microservices ont pu gérer une charge de travail plus importante que le monolithe. De plus, nous avons constaté que le temps de réponse moyen du **microservice** était beaucoup plus faible, avec **148 millisecondes**, contre **502 millisecondes** pour le **monolithe**.

Cette différence de performance suggère que la modularité et la flexibilité des microservices leur permettent de mieux répondre aux fluctuations de la charge de travail.

Fait intéressant, le microservice est resté stable et a maintenu un débit élevé de requêtes par seconde, tandis que les performances du monolithe se sont dégradées de manière significative lorsque nous avons augmenté la charge. Cela démontre que l'architecture microservices est mieux adaptée pour gérer des charges de travail importantes et variables, offrant ainsi une meilleure expérience utilisateur. Ces résultats confirment les avantages des microservices en termes de passage à l'échelle et de résilience par rapport à l'approche monolithique, qui peut rapidement atteindre ses limites lorsque la demande augmente.

3.4.3 Détails des requêtes pour 60 vues pendant 30 secondes

Lors de ces tests, nous avons pu observer en détail les performances des différents services composant nos architectures. Pour l'architecture microservices, nous avons constaté que le service "**customer**" était celui qui prenait le plus de temps, avec des temps de réponse moyens de **103 ms** et **152 ms**. En comparaison, le service "**product**" affichait des temps beaucoup plus faibles, de l'ordre de **9,9 ms** et **1,5 ms**. De même, pour l'architecture monolithique, le service "**customer**" semblait être le point faible, avec des temps de réponse plus élevés que les autres composants.

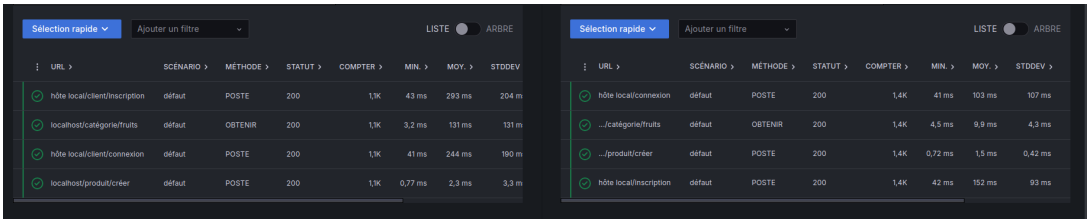


FIGURE 16 – Détails des requêtes pour 60 vues pendant 30 secondes

Ces observations suggèrent que l'optimisation du service "**customer**" pourrait permettre d'obtenir de meilleures performances globales, aussi bien pour l'architecture microservices que pour l'architecture monolithique. En améliorant les temps de réponse de ce service critique, nous devrions pouvoir réduire les temps de réponse moyens et offrir une meilleure expérience utilisateur, quel que soit le type d'architecture choisi.

3.4.4 Tests avec 80 vues pendant 30 secondes

Nous avons effectué des tests de performance avec k6 en utilisant **80 vues pendant 30 secondes**, à la fois sur notre architecture monolithique et sur notre architecture microservices.

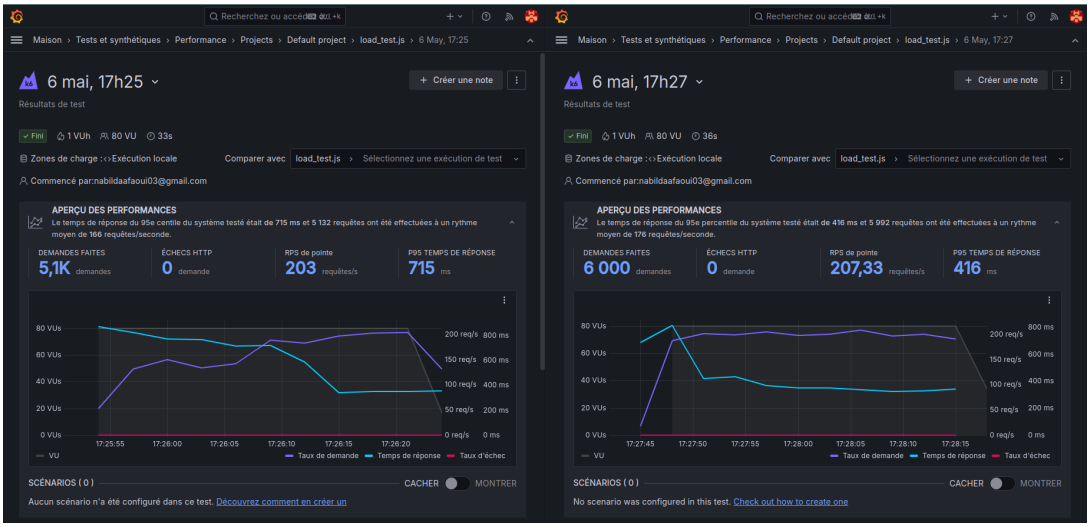


FIGURE 17 – Résultats des tests avec 80 vus pendant 30 secondes

Lors de nos tests avec **80 vus pendant 30 secondes**, nous avons observé que l'architecture microservices a pu supporter une charge de travail plus importante, avec un total de **6 000 requêtes**, contre seulement **5 100** pour **l'architecture monolithique**. De plus, le temps de réponse moyen du **microservice** était de **416 millisecondes**, contre **700 millisecondes** pour le **monolithe**.

3.4.5 Tests avec 100 vus pendant 50 secondes

Nous avons effectué des tests de performance avec k6 en utilisant **100 vus** pendant **50 secondes**, à la fois sur notre architecture monolithique et sur notre architecture microservices.

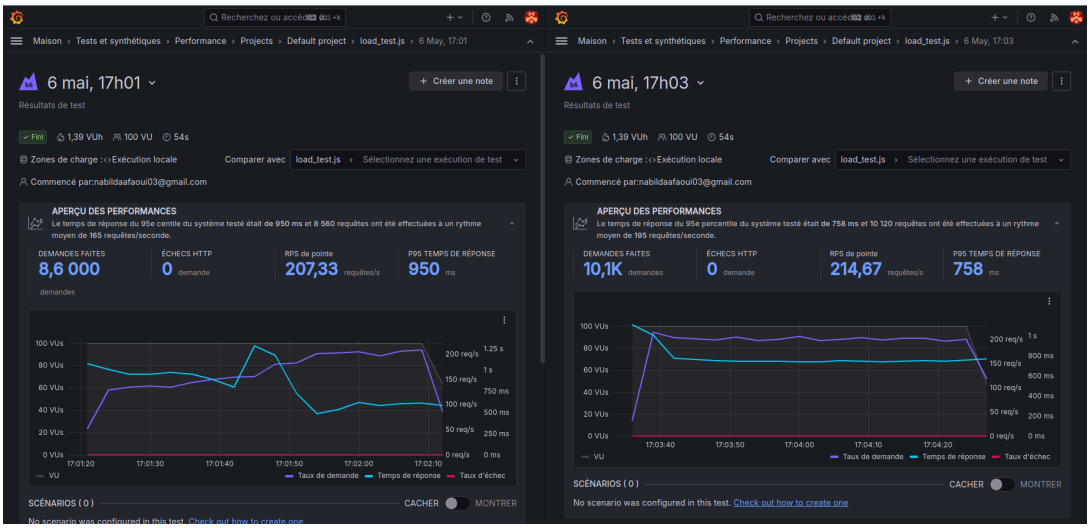


FIGURE 18 – Résultats des tests avec 100 vus pendant 50 secondes

Lors de ces tests sous une charge encore plus importante, nous avons constaté une différence significative entre les deux architectures.

Pour l'architecture **monolithique**, le nombre total de requêtes atteignait seulement **5 300**, avec un temps de réponse moyen de **922 millisecondes**. En revanche, l'architecture **microservices** a pu gérer **une charge beaucoup plus élevée, avec** 6 000 requêtes au total. De plus, le temps de réponse moyen est resté relativement stable, à **799 millisecondes**.

Ces résultats démontrent clairement que l'architecture microservices est beaucoup mieux adaptée pour supporter des charges de travail importantes et variables. Alors que le monolithe a montré des signes de saturation, les microservices sont restés stables et ont continué à offrir de meilleures performances.

3.4.6 Tests avec 1 vu pendant 30 secondes

Pour compléter notre analyse, nous avons également effectué des tests avec **1 vu pendant 30 secondes**, afin d'évaluer les performances de nos architectures dans un scénario de faible charge.

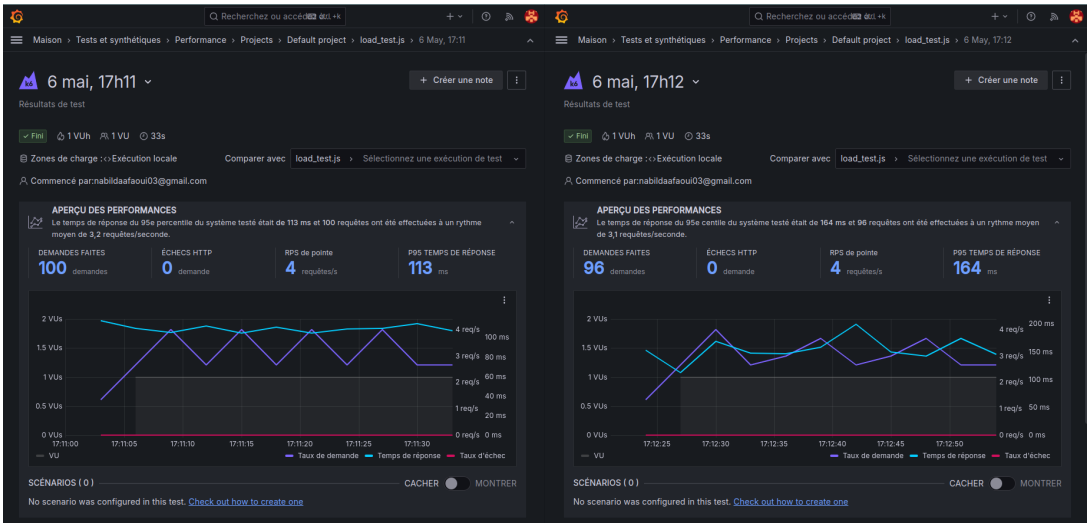


FIGURE 19 – Résultats des tests avec 1 vu pendant 30 secondes

Lors de nos tests avec **1 vue pendant 30 secondes**, nous avons observé que dans ce scénario de faible charge, l'architecture monolithique offrait de meilleures performances que l'architecture microservices.

En effet, le **monolithe** a pu exécuter **100 requêtes**, contre seulement **96** pour les **microservices**. De plus, le temps de réponse moyen du **monolithe** était de **113 millisecondes**, légèrement inférieur aux **164 millisecondes** enregistrées pour les **microservices**.

Ces résultats suggèrent que pour des applications prévues pour gérer un nombre relativement faible de requêtes, l'architecture monolithique peut s'avérer plus intéressante, non seulement en termes de performances, mais aussi en termes de facilité de mise en place. La simplicité inhérente au monolithe peut en effet s'avérer un avantage dans ce type de scénario. Cependant, comme nous l'avons vu précédemment, lorsque la charge augmente, les microservices deviennent généralement plus adaptés grâce à leur meilleure évolutivité et résilience.

3.5 Conclusion sur les résultats

Les tests de charge réalisés avec l'outil K6 ont permis de mettre en évidence les forces et faiblesses respectives des architectures monolithique et microservices dans notre projet.
De manière générale, les résultats ont clairement démontré que l'architecture microservices offre de meilleures performances et une meilleure capacité à supporter des charges de travail importantes et variables. Lorsque nous avons augmenté progressivement le nombre d'utilisateurs virtuels, les microservices ont pu maintenir un débit élevé de requêtes et des temps de réponse stables, contrairement à l'architecture monolithique qui a montré des signes de saturation.

Cette différence de comportement s'explique par la modularité et la flexibilité inhérentes aux microservices. Chaque service pouvant être mis à l'échelle indépendamment, l'architecture microservices est mieux à même de s'adapter aux fluctuations de la charge.

Cependant, nos tests ont également révélé que pour des scénarios de faible charge, l'architecture monolithique peut s'avérer plus performante et plus simple à mettre en place. Le monolithe a en effet affiché de meilleurs temps de réponse lorsque nous n'avons simulé qu'un seul utilisateur virtuel.

Ces résultats suggèrent que le choix entre une architecture monolithique ou microservices dépendra fortement des besoins spécifiques de notre application, notamment en termes de volume de trafic attendu. Pour des applications amenées à gérer des charges de travail importantes et variables, l'architecture microservices semble être la meilleure option. Mais pour des applications de plus petite envergure, le monolithe pourrait s'avérer plus adapté.

3.6 Réconfiguration : Quelques techniques

3.6.1 Ajout d'un API Gateway

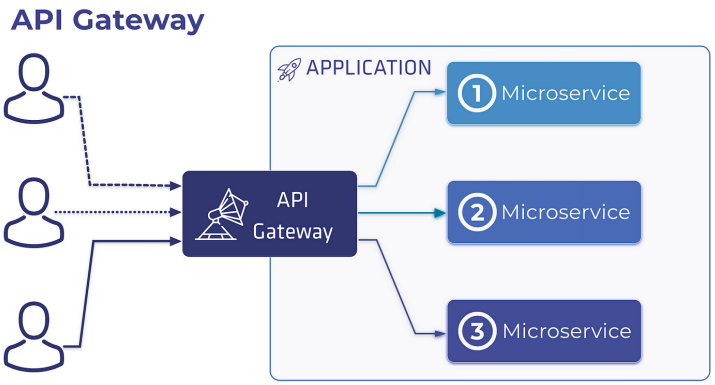


FIGURE 20 – API Gateway

L'introduction d'un API Gateway représente une étape décisive dans la reconfiguration d'une architecture monolithique ou de microservices. En agissant comme une porte d'entrée centrale, l'API Gateway rationalise la gestion des requêtes en fournissant un point d'accès unique aux services backend. Cette consolidation simplifie non seulement le processus de développement et de maintenance, mais permet également de mettre en œuvre des politiques de sécurité, d'authentification et de contrôle d'accès de manière uniforme. De plus, l'API Gateway offre une couche d'abstraction qui facilite l'évolutivité en permettant l'ajout ou la modification de services sans perturber les clients existants. En résumé, l'intégration d'un API Gateway optimise la performance, la sécurité et la flexibilité du système dans son ensemble.

3.6.2 Ajout d'un Load Balancer

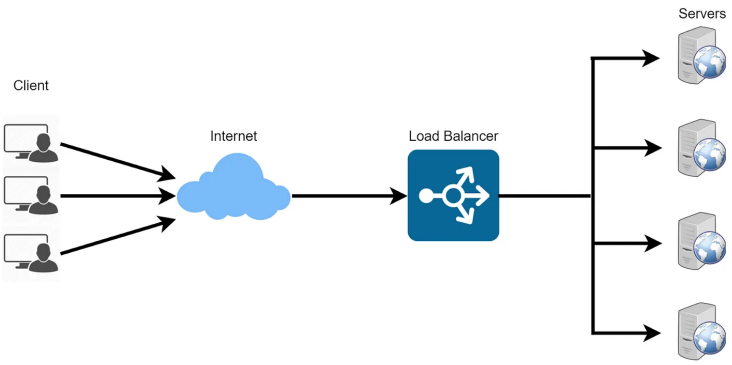


FIGURE 21 – Load Balencer

L'ajout d'un Load Balancer constitue une stratégie essentielle pour améliorer la disponibilité, la résilience et les performances d'une infrastructure informatique. En distribuant de manière équilibrée la charge de travail entre les différentes instances de serveurs, le Load Balancer permet d'optimiser l'utilisation des ressources et d'éviter les engorgements. Cette répartition intelligente du trafic améliore la réactivité du système en garantissant une réponse rapide aux requêtes des utilisateurs, tout en minimisant les temps d'attente et les risques de défaillance. De plus, en surveillant en temps réel l'état de santé des serveurs et en redirigeant automatiquement le trafic en cas de défaillance, le Load Balancer renforce la fiabilité de l'infrastructure et assure une expérience utilisateur fluide et sans interruption.

4 Conclusion

Au travers de ce projet, nous avons pu explorer en détail les différences entre les architectures monolithiques et microservices, en nous appuyant sur une application de commande en ligne pour aliments comme cas d'étude.
L'analyse approfondie de l'architecture monolithique initiale nous a permis de mettre en évidence ses principaux avantages, tels que la simplicité de mise en œuvre et de déploiement, mais aussi ses limites en termes de scalabilité et de maintenabilité. En effet, la forte interdépendance entre les différents services du monolithe rend les évolutions et les mises à jour complexes.

Nous avons ensuite migré cette application vers une architecture microservices, en identifiant les différents services métier (gestion des produits, des clients, du panier, etc.) et en les découplant pour les déployer de manière indépendante. Cette nouvelle architecture offre une meilleure scalabilité, une plus grande flexibilité technologique et une résilience accrue, au prix d'une complexité accrue en termes d'orchestration et de gestion de la cohérence des données réparties.
L'évaluation comparative des performances et de la scalabilité des deux architectures a montré que le choix entre monolithe et microservices dépend fortement des besoins spécifiques de l'application et de son contexte d'utilisation. Certaines charges de travail peuvent bénéficier davantage de l'approche microservices, tandis que d'autres seront mieux adaptées à une architecture monolithique.
Au-delà des aspects techniques, ce projet nous a également permis de prendre en compte les enjeux organisationnels et de gouvernance liés à la migration vers une architecture microservices. La nécessité de mettre en place des équipes dédiées, des processus de collaboration et de gestion des dépendances entre services s'est avérée être un défi majeur à relever.

En conclusion, ce travail nous a offert une vision globale des avantages et des inconvénients des architectures monolithiques et microservices. Il nous a également sensibilisés aux différents facteurs à prendre en compte lors du choix d'une architecture adaptée aux besoins de l'application et de l'organisation. Ces enseignements seront précieux pour guider nos futurs projets de développement et d'évolution architecturale.

Références

[1] ISO/IEC 25010 :2011 Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

[2] Chris Richardson, *Microservice Patterns*, Manning Publications, 2019.

[3] Miell, I., & Shubin, A. (2020). *Observability Engineering : Achieving Production Excellence*. O'Reilly Media, Inc.

[4] Rapport de projet TER, fourni dans les documents de recherche.

[5] Rathore, N., Rajavat, A. (2022). Evaluating the Performance of Monolithic and Microservices Architectures in an Edge Computing Environment. *International Journal of Fog Computing*, 5(1), 1-18.

[6] Blinowski, G., Ojdowska, A., Przybylek, A. (2022). Monolithic vs. Microservice Architecture : A Performance and Scalability. arXiv preprint arXiv :2202.08651.

[7] Faustino, D., Gonçalves, N., Portela, M., Rito Silva, A. (2022). Stepwise Migration of a Monolith to a Microservices Architecture : Performance and Migration Effort Evaluation. *IEEE Access*, 10, 16524-16541.

[8] Barzotto, T. R. H., Farias, K. (2022). Assessing the impacts of decomposing a monolithic application for microservices. *Journal of Software : Evolution and Process*, e2451.

[9] Auer, F., Lenarduzzi, V., Felderer, M., Taibi, D. (2021). From monolithic systems to Microservices : An assessment framework. *Information and Software Technology*, 129, 106405.

[10] Lenarduzzi, V., Lomio, F., Saarimäki, N., Taibi, D. (2020). Does migrating a monolithic system to microservices decrease the technical debt ?. *Journal of Systems and Software*, 169, 110710.