CentraleSupélec

MEMOIRE DE MASTERE SPECIALISE

ARCHITECTURE DES SYSTEMES D'INFORMATIONS
ARCHITECTURE LOGICIELLE

Promotion 2019

ETAT DE L'ART DES ARCHITECTURES CLOUD-NATIVES

MICROSERVICES ET API MANAGEMENT

Présenté par : Nancy NJEUNDJI
Encadrée par : Pierre-Frédéric ROUBERTIES

REMERCIEMENTS

Je tiens tout d'abord à remercier mes parents Bertrand NJEUNDJI et Gertrude NJEUNDJI NGASSA pour le soutien financier, moral et la confiance sans borne qu'ils m'ont donnés tout au long de mes études et de ce projet.

Je remercie du fond du cœur ma petite sœur Romy-Kelly NJEUNDJI et mon petit frère Krys-Warren NJEUNDJI, pour le soutien moral, la joie et la bonne humeur qu'ils m'ont apportés tout au long de mes recherches.

Je remercie particulièrement, M. Pierre-Frédérique ROUBERTIES pour le suivi, les remarques, les suggestions, la patience et la grande disponibilité dont il fait preuve tout au long de ce projet. Ainsi que toute l'équipe pédagogique de CentraleSupélec, pour les connaissances apportées tout au long de mon cursus MS ASI-AL.

Je remercie tous les architectes et les professionnels de l'IT que j'ai côtoyé tout au long de ce projet pour les précieux conseils, revues et observations qu'ils m'ont prodigués et pour avoir pris le temps de répondre à toutes mes questions.

Glossaire

A

API	Application Programming Interface : Interface de programmation applicative Management Gestion des API
APM	Applications Performance Monitoring : Surveillance de la performance des applications
Autoscaling	Scalabilité automatique : capacité d'un produit à s'adapter à un changement d'ordre de grandeur de la demande de façon automatique

B

B2B	Business To Business : vente de professionnel à professionnel
B2B2C	Business To Business To Customer : vente de professionnel à professionnel et particuliers
B2C	Business To Customer : vente de professionnel à particuliers
Backend	Partie applicative qui gère les fonctionnalités et les accès aux données
BASE	Basically Available Soft state Eventually consistent : groupe de propriété s'appliquant à une transaction de base de données.
BDD	Base de données
Build	Construire/compiler/empaqueter les fichiers d'un projet

C

Caching	Mise en cache : enregistrement en mémoire de certaines données pour améliorer la rapidité de consultation des données.
Commit	Enregistrement effectif d'une transaction dans une base de données.
Cookie	Témoin de connexion : suite d'informations envoyée par un serveur HTTP chaque fois que ce dernier est interrogé et sous certaines conditions
Cross-devices	Expérience de transfert de données en simultanées sur divers appareils (mobile, tablette, ordinateurs, objets connectés, etc.) via internet
CRUD	Opérations « Create Read Update Delete » sur une base de données
CI/CD	Continuous Intégration/Continuous Delivery : Intégration continue/ Livraison continue

D

DDD	Domain-Driven Design : Conception orientée design
Dos	Deny of Services – type d'attaque de piraterie informatique

E

Endpoint	Point de terminaison ou point d'accès à un service
ESB	Enterprise Service Bus : Bus d'échange de données (progiciel)

F

Framework	Ensemble de briques logicielles factorisant du code technique ou des modèles de conception
Frontend	Partie applicative qui gère les interactions Homme machine
FTP	File Transfert Protocol : Protocole de transfert de fichier

H, I, J, L

HTTP/HTTPS	Protocole de transfert hypertexte/ Protocole de transfert hypertexte sécurisé
IHM	Interface Homme-Machine - les pages d'applications
JSON	JavaScript Object Notation : format de données textuelles dérivé de la notation des objets du langage JavaScript
LDAP	Lightweight Directory Access Protocol : protocole permettant l'interrogation et la modification des services d'annuaire
Load-balancing	Technique de distribution des charges sur différents appareils d'un même réseau

M, O

Middleware	intergiciel – logiciel tiers qui crée un réseau d'échange d'informations entre plusieurs applications informatiques
Monitoring	surveillance des mesures d'une activité
MSA	Microservices Architecture : architecture de microservices
Multicanal	Utilisation simultanée de plusieurs moyens de communication
OAuth1/Oauth2	Protocole standardisés de sécurité et d'autorisation d'application tierce
Package	Dossier compressé contenant les archives d'une application
Payload	Contenu structuré du corps d'une requête
Pool	(de connexion). Lot de connexions à la base de données enregistrées en cache afin d'être réutilisées pour gagner en performance.

Proxy	Pare-feu : logiciel intermédiaire de surveillance des échanges entre deux hôtes

R

Reporting	Ensemble de données de rapports et de statistiques
REST	REpresentational State Transfer: style d'architecture pour les systèmes hypermédia distribués

S

SLA	Service-level agreement : entente de niveau de service - document qui définit la qualité de service pour prestation de service d'un fournisseur à un client
SOA	Services-Oriented architecture : architecture orientée services
SOAP	Simple Objet Access Protocol protocole de messagerie écrit en XML et permettant à des systèmes d'exploitation distincts de communiquer via HTTP
SSL	(un type de connexion)
Dispatching	
Queuing	

T

Throttling	
Time-to-market	(delivery) Livraison juste à temps sur le marché commercial
TLS	Transport Layer Security – norme de sécurisation par chiffrement du transport de l'information au sein d'un réseau informatique

U, V, W

URL	Uniform Ressource Locator – nommage universel de la location d'une ressource
UUID	Universal Unique Identifier : identifiant universel unique
VM	Virtual Machine : Machine virtuelle
Workflow	Flux de travail – modélisation du processus de circulation des flux d'informations

Table des matières

Introduction générale

En raison du succès des géants du web tel que Google, Amazon, Facebook et notamment Uber lors de la conception de nouvelles architectures informatiques permettant de répondre à plusieurs problématiques liées à la gestion de volumes colossaux d'utilisateurs et de données, l'année 2015 a marqué ce que l'on pourrait appeler le pic d'une nouvelle ère digitale qui a contraint les entreprises traditionnelles à se remettre véritablement en questions quant à leur propre architectures et à enclencher divers chantiers de transformations digitales.

En effet, comment des entreprises telles que celles évoquées plus haut ont pu gérer des défis d'une telle ampleur alors que les moyennes et grandes entreprises ont du mal à le faire sur une base d'utilisateurs et de données 1000 fois moindre ?

Bien avant de se pencher sur les solutions architecturales dans les entreprises qui semblent ne pas être adaptées remarquons d'abord ceci : L'essor des applications et technologies mobiles qui a permis aux utilisateurs d'être ultra connectés, en tout lieu et à tout moment de la journée, a eu pour conséquence majeure la nécessité de mettre en place des systèmes très disponibles, distribués et sécurisés, ce qui a complément bousculé les entreprises sur de nombreux plans :

Sur le plan logiciel : La naissance de nombreux frameworks client offrant de meilleures expériences utilisateurs dont il faille absolument s'approprier pour fidéliser la clientèle. Les montées en versions drastiques des frameworks coté serveur pour redoubler de performance dans le traitement de données, de gestion de la mémoire, de la montée en charge, de sécurisation des accès aux ressources. La nécessité d'adopter les modèles de données NoSQL pour dépasser les limites des systèmes de gestion de base de données relationnels afin de pouvoir "passer à grande échelle".

Sur le plan de l'infrastructure : L'importance accrue de mettre en place des chaines d'intégration et livraison continues de plus en plus automatisées qui brisent les frontières entre les développeurs de solutions et les opérationnels chargés des mises en productions de ces dites solutions, créant ainsi le concept de DevOps. De plus à une certaine échelle, l'infrastructure est devenue une commodité, l'ouverture du marché du *Cloud Computing* offrant des formules de **Everything As A Service* est devenue une opportunité d'externaliser l'informatique et d'en confier la gestion à des spécialistes avec une promesse de fast-delivery et d'infrastructure élastique à la demande.

Sur le plan organisationnel et économique : La nécessité de mettre en place des techniques d'agilité pour travailler rapidement et efficacement dans un milieu de plus en plus concurrentiel, forçant à changer le modèle de gestion des projets et par conséquent celui de la gouvernance. Le business model

va au-delà du service métier principal de l'entreprise, le savoir-faire et l'accès aux ressources peut être proposée dans des échangent B2B et B2C monétisés.

En Bonus : Cette nouvelle ère du digital est à cheval sur une certaine ère de la piraterie informatique, demandant un niveau de vigilance un peu plus accru en ce qui concerne la sécurisation des données et des systèmes.

La rencontre de cet ensemble de contraintes aura permis aux géants de concevoir des solutions, concepts et pratiques qui ont changé l'horizon du digital tel que l'on le connait à l'heure actuelle. Mais au centre de cet écosystème, réside principalement les microservices et les interfaces de programmations (API) dont les caractéristiques et spécificités ont orchestré la mise en place d'un panel élevé de pratiques et de solutions.

Sachant qu'une grande partie des entreprises implémentent de la SOA et que quelques-unes ont dans leur portefeuille de projets réalisés, quelques monolithes, les DSI aimeraient prendre des décisions et mesures afin d'éviter d'éventuelles disruptions qui pourraient être fatale à leur entreprise. Plusieurs questions se posent :

- Quelles sont les spécificités de ce type de services qui permettent qu'on puisse mettre en place tout un écosystème capable de gérer des problématiques de distribution, de scalabilité, de sécurité, d'agilité et tout en effectuant des livraisons fréquentes permettant de rester compétitifs ?
- Quels sont les impacts, les possibilités, les avantages et les inconvénients de l'implémentation d'une architecture dite cloud-native au sein d'une entreprise ?
- Compte tenu du métier et des besoins d'une entreprise, comment jauger la nécessité de mettre en place cette architecture ?
- Et finalement, si l'entreprise prend la décision de mettre en place cette architecture, comment effectuer la transformation ?

Autant de questions pour lesquelles un ensemble de réponses seront apportées tout au long de ce document. Tout d'abord avec une approche explorative allant de la phase de conception à la phase d'intégration en passant par plusieurs notions clés et bonnes pratiques spécifiquement liées à cette architecture. Ensuite l'accent sera mis sur l'API management afin d'analyser la mise en place de la gouvernance des microservices. Au fur et à mesure, un état des lieux des bénéfices et contraintes devrait permettre certaines remises en question.

Pour démontrer par l'exemple la mise en place de cette architecture, une étude de cas de l'application mobile CoMove, qui fournit un service de covoiturage dans l'agglomération de Toulouse, sera présentée

dans le document. Et enfin, une synthèse de tout l'écosystème des architectures cloud-native permettra de prendre une décision quant à la nécessité d'amorcer une éventuelle migration d'architecture.

1. Contexte

Comme évoqué succinctement en introduction de ce document, un ensemble d'industries leaders du logiciel, plus communément connus sur l'appellation de géants du web, ont entrepris des transformations architecturales profondes de leurs structures sur tous les plans (entreprise, logiciel et infrastructure). Ces transformations ont eu un impact qui a littéralement changé l'aspect de l'industrie du web tel qu'on le connait aujourd'hui et de ce fait marqué une nouvelle ère digitale portée par le cloud.

A la question qui consiste à se demander pourquoi ces entreprises ont t'elles opérées de tels changements pour la mise en place d'applications dites cloud-native, on pourrait observer un ensemble de motivations qu'elles ont toutes en commun :

- La rapidité : Dans un marché du service, relativement concurrentiel, la capacité de mettre son produit à disposition de l'utilisateur final le plus vite possible est une valeur clé du succès de vente et de fidélisation. En essayant de répondre à la question « comment devenir ready-to-market », elles ont été capable d'innover, d'expérimenter et de proposer des solutions logicielles et des techniques de gestion de projet qui ne servaient qu'à cet objectif. L'agilité et l'automatisation des chaines de production, ont permis d'augmenter la fréquence des déploiements à une centaine de fois par jour, réduisant par cette occasion les marges de d'erreurs, favorisant la prise de risque et la réutilisabilité au travers des APIs.

- La sécurité : Les architectures basées sur le cloud on permit de démontrer qu'il est tout à fait possible d'allier vitesse et sécurité en mettant en place un ensemble de procédures et technique permettant entre autres :
    - D'avoir de la visibilité à l'aide de métriques permettant de corriger des erreurs, combler des failles et contre-attaquer (cf. chapitre 7)
    - De faire de l'isolation afin de contrôler et limiter les risques associés aux échecs en cascade notamment à l'aide d'une conception du système en microservices (cf. chapitre 2)
    - D'être tolérant aux pannes qui est également un avantage d'avoir un système décomposé en composants indépendamment déployables (microservices).
    - De faire de la récupération automatique après avoir pu identifier les échecs et de les avoir traités.

- La mise à l'échelle horizontale : plutôt que d'avoir des applications de plus en plus grosses entrainant ainsi l'achat de plus gros serveurs afin de supporter de plus grands calculs et de montées en charge, ces entreprises ont trouvé plus judicieux de réduire la taille des applications et de repartir les instances de ces dernières sur un grand nombre de serveurs peu couteux, facile à acquérir et rapide à déployer. De plus, cette conception a permis d'améliorer les techniques de virtualisation et d'optimiser les ressources des serveurs avec la conteneurisation (cf. point 6.3.). L'infrastructure externalisée sur le cloud est devenue un service hautement commercialisé proposant aux entreprises plus petites de déployer leur logiciel à moindre coût.

- Des applications centrées sur le mobile : Les entreprises ont vite compris qu'avec un taux de pénétration du mobile très élevé à l'échelle planétaire et avec la performance de plus en plus élevée dans la conception de ces derniers, permettant aux utilisateurs d'accéder aux ressources à n'importe quel moment et cela de n'importe où, l'ère des applications de bureau était révolue. Une approche mobile-first a eu un impact très structurant sur la façon d'aborder les architectures, du fait de la diversité des plateformes mobiles et de la nécessité d'avoir un élément telle qu'une API Gateway (voir point 2.3.3) dans son modèle de conception qui va faire le routage des demandes et l'agrégation des requêtes, afin de fournir au plus vite les informations demandées par l'utilisateur pour le fidéliser.

- L'innovation : L'idée derrière une architecture de microservices est la capacité à ajouter de fonctionnalités sans impacter l'existant. La modularité de cette architecture permet de s'engager sur de nouveaux produits, langages, frameworks et marchés sans risquer l'effondrement du système. La capacité à innover est l'un des éléments importants ayant permis à ces entreprises à se placer en tête de l'industrie.

- L'API Economy : l'idée avant la mise en place de ce nouveau modèle économique basée sur la monétisation des APIs, était de pouvoir utiliser des fonctionnalités existantes chez des tiers, dans un souci de réutilisabilité et de rapidité. L'émergence des réseaux sociaux a fortement contribué à la montée de ce modèle économique très B2B, ce qui a permis de structurer et normaliser les APIs pour mieux gérer l'accès aux ressources. Les notions d'API RESTful, publiques et privées et d'API management ont été une nécessité dans la réussite de stratégie économique

Afin d'obtenir les résultats présentés ci-dessus, les géants du web ont dû se délester d'applications traditionnelles monolithique, constituées d'une grande base de code, difficile à appréhender, à maintenir et à faire évoluer. De plus, les déploiements fréquents rendaient les applications indisponibles quand survenaient le moindre échec, cela devenait une manœuvre risquée et couteuse. En termes de scalabilité, chacune des instances ayant accès aux données, les stratégies de mise en cache de d'une partie des données augmentait considérablement la consommation de mémoire et de trafic d'E/S. Sans compter que le load-balancing n'était pas assez optimal car ne permettait pas de différencier les composants ayant des besoins en ressources qui nécessitaient une utilisation intensive de la mémoire de ceux nécessitant une utilisation intensive du processeur…

D'après ces observations, on pourrait s'intéresser aux architectures cloud-native dans le but :

- D'être plus rapide dans la mise en production des solutions
- Avoir des applications plus sécurisées avec l'analyse des flux de données
- De profiter des avantages d'une scalabilité horizontale
- D'avoir une approche conceptuelle centrée sur le mobile
- D'avoir la capacité d'innover grâce à l'aspect modulaire de l'architecture.
- D'avoir une stratégie économique dans la commercialisation des APIs

La mise en place d'une architecture basée sur le cloud nécessite de comprendre les fondamentaux de cette architecture, en commençant par la conception et l'implémentation des microservices et des APIs, la communication entre ces services et l'exposition des APIs qui les regroupent, les différentes stratégies de déploiement actuelles et la gouvernance au travers de la notion d'API management. Tous ces concepts sont autant d'élément à prendre en compte pour aider à la prise de décision d'une éventuelle migration vers ces systèmes.

2. Architecture de microservices

2.1. Définitions microservice et API

Microservices et architecture

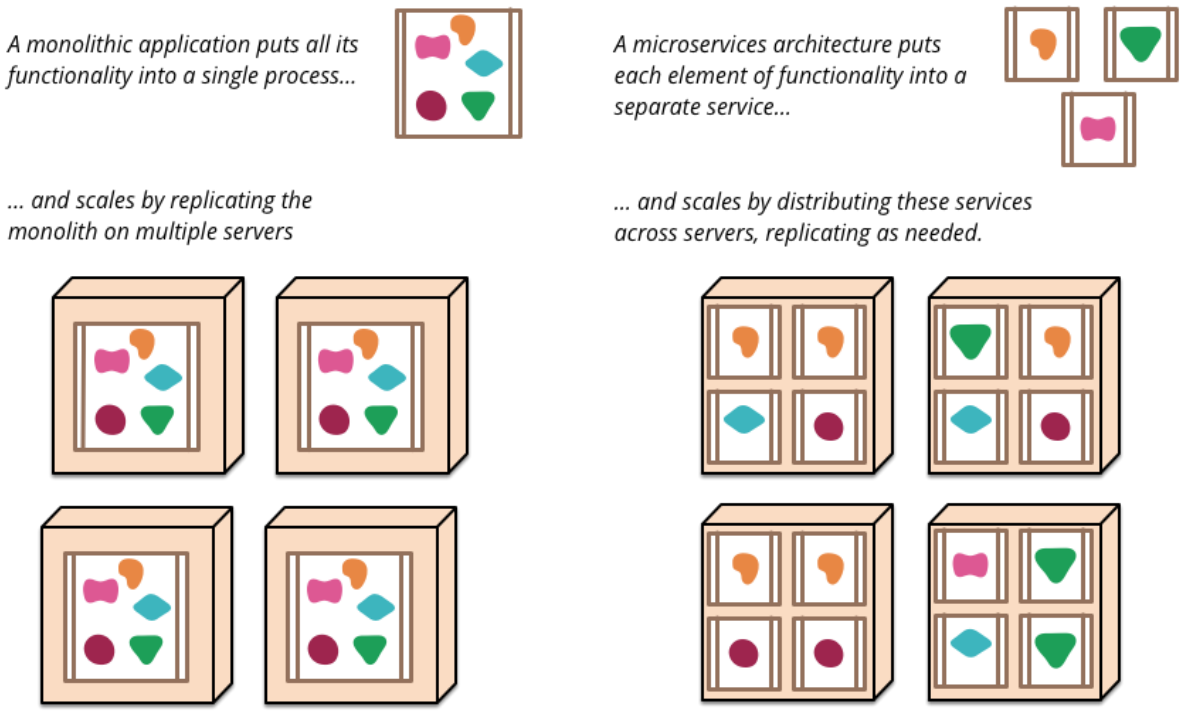
Dans son blog, Martin Fowler, auteur conférencier et porte-parole du développement logiciel, explique que le terme « Architecture de microservices » apparu ces dernières années pour décrire une manière particulière de concevoir les applications logicielles est une approche permettant de développer une application sous forme d'une suite de petits services.

Chaque service s'exécutant dans son propre processus, serait capable de communiquer à l'aide d'un protocole léger, le plus souvent à base de ressources HTTP. Le service est construit autour d'une capacité métier bien définie et peut être déployé indépendamment par des machines de déploiement entièrement automatisées. Grace à une gestion centralisée d'un microservice, ce dernier peut respecter le principe de responsabilité unique, en étant écrit dans un langage de programmation qui lui est propre et utiliser différentes technologies de stockage. Ceci permet de donner libre choix quant aux outils et langages les plus pertinents à même de répondre à l'objectif auquel doit tenir le microservice.

Bien que faisant parti d'un ensemble, il est important de comprendre qu'un microservice est avant tout un **service métier** regroupant des *services techniques* (REST, SOAP...) qui ensemble fournissent une fonctionnalité précise, cohérente et logique qui a un **sens métier**.

Une architecture de microservices serait donc un ensemble de **composants** indépendamment remplaçables et maintenables qui communiquent via des mécanismes tels que les requêtes de services web ou les appels de procédures à distance. L'utilisation du terme "composant" pour désigner le microservice permet de souligner dans un premier temps son unicité et par la même occasion, le fait qu'il fasse partie d'un système global fonctionnel et cohérent.

Cette architecture à l'échelle du service a été créé pour pallier les difficultés liées au systèmes monolithiques rencontrés sur les grands projets :



*Source : **www.martinfowler.com***

- *Une application monolithique met toute sa fonctionnalité dans un unique processus…et évolue en le répliquant le monolithe sur plusieurs serveurs*
- *Une architecture de microservices met chaque fonctionnalité dans un service séparé …et évolue en répartissant ces services sur plusieurs serveurs en les répliquant au besoin*

En somme, un microservice peut être défini comme un service métier autonome, ayant son propre code, son propre cycle de vie, gérant ses propres données sans les partager directement avec les autres services et basé sur une technologie permettant de répondre à l'objectif métier de ce dernier. Ainsi une architecture de microservices devrait permettre d'améliorer la performance d'un système par :

- L'évolutivité et la fiabilité
- La scalabilité horizontale
- L'innovation technologique
- L'innovation métier
- Une distribution plus large du système
- Le time-to-market delivery (la livraison juste à temps)

Les microservices ayant un accès direct aux ressources d'une organisation, il est nécessaire de les regrouper et de faciliter leur exploitation au travers d'une interface de programmation applicative, API, pour gérer les accès à ces derniers, masquer leur complexité et améliorer la communication quant à leur utilité.

APIs et gestion

"Dans l'industrie contemporaine du logiciel, une interface de programmation applicative (souvent désignée par le terme API pour application programming interface) est un ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. Elle est offerte par une bibliothèque logicielle ou un service web, le plus souvent accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur."

*Source : **Wikipédia***

Autrement dit, une API peut être considérée comme un contrat proposant les fonctionnalités d'une solution logicielle sans avoir à implémenter ni même connaître le processus de réalisation et la complexité de ces dernières.

Les bonnes pratiques suggèrent qu'une API spécifie, au travers d'une page web statique ou tout autre outil visuel, une description de l'usage pour chaque microservices. Cette mise en place de la documentation des services confirme que ces derniers sont viables et propice à être utilisé pour accéder aux ressources de l'entreprise, on parle donc "*d'exposition de services*".

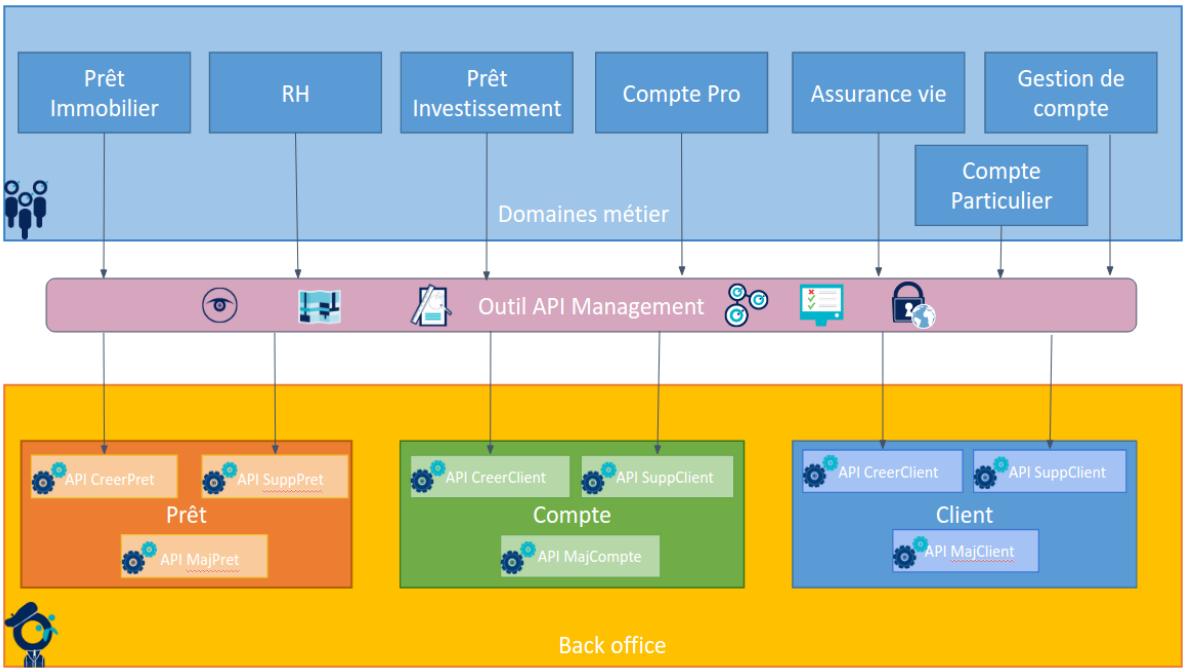
En règle générale, dans l'industrie informatique, les utilisateurs finaux des APIs sont les développeurs, c'est cette raison qui définit l'état d'esprit et l'ensemble des normes qui régissent la création des services à exposer. Ils doivent être plus ou moins simples afin d'offrir un champ de possibilité assez large pour les processus que le développeur souhaite réaliser et incorporer dans ses applications. Les APIs exposent des services à l'extérieur de leur système souvent dans une stratégie B2B, mais les développeurs peuvent aussi créer et utiliser des APIs pour leur propre besoin.

La gestion des APIs ou API management, est un processus de gouvernance qui permet à une entreprise de publier des APIs et de superviser leur cycle de vie au sein d'un environnement sécurisé et évolutif. L'implémentation d'un tel processus devrait permettre entre autres :

- L'exposition et la documentation des services, définition de l'interface et des règles de nomenclatures appliquées.
- L'implémentation de la sécurité
- La gestion des authentifications et habilitations (en création directe ou en s'appuyant sur une brique existante telle qu'un LDAP par exemple)
- La supervision des services exposés (monitoring) au travers de métriques techniques telles que : les nombres d'appels et de réponses KO, les temps de réponses, les taux d'utilisations de la mémoire ou du cache.
- La gestion du trafic (quotas, caching, arrêt rapide, limitations/throttling)
- L'application de la stratégie de gouvernance des services fondée sur la granularité et la politique de gestion des versions de ces derniers
- L'enrôlement vers des consoles interactives via un portail pour les développeurs et un autre pour les administrateurs

Il est à noter que rajouter un tel outil à son architecture nécessite au préalable d'avoir étudié le système auquel on souhaite l'appliquer, d'avoir posé une stratégie claire de l'architecture et défini les services répondant aux besoins métiers.

Il existe sur le marché, plusieurs plateformes ayant des fonctionnalités répondant à différents besoins (type d'expositions, monitoring ; cache, outil de transformation...) et cette opération en mode Bottom-top (les APIs d'abord et l'outil d'API management ensuite) facilite grandement le choix des outils, surtout si l'on souhaite implémenter une solution externe.



*Source : **blog.octo.com***

Mise en situation d'un outil d'Api management au sein d'une architecture, qui gère l'accès aux APIs de gestion de Prêt, de Compte et de Client. Chacune de ces APIs expose des microservices qui permettent d'effectuer des opérations sur des ressources (CréerPret, SupprClient, MajCompte, etc.)

Sachant qu'un microservice est un composant métier d'une application et qu'il est défini par un ou plusieurs services techniques appelés **opérations**, une API est une interface permettant d'accéder à un ou plusieurs microservices et par conséquent à leur opération. De ce fait, un Microservice pourrait ne pas être accessible via une API si cette dernière ne l'expose pas.

Les APIs devenant ainsi des points d'accès aux ressources d'une organisation, l'importance d'avoir un outil d'API management au sein de cette organisation permet d'instaurer une stratégie de gouvernance, d'autant plus si le nombre de microservices est élevé. L'outil devrait faciliter des réponses et des prises de décisions concernant des questions telles que celles-ci :

- Quelle API exposera quels microservices ?
- Dans quel objectif ?
- Qui y aura accès ?
- Quelles sont les informations que révèlent le trafic au sein de cette API
- Quel est le niveau de sécurité de l'API par rapport à la criticité des ressources ? Etc.

2.2. Conception d'une architecture de microservices

Lors de la mise en place d'un projet informatique, la première phase consiste à la conception de ce dernier. En fonction de l'architecture, de la structure d'un projet et des objectifs que l'on souhaite atteindre, il existe plusieurs modèles de conception possibles. Concernant une architecture de microservices, la conception orientée domaine, encore appelée DDD s'est démarquée par les techniques de séparations de modèle qu'elle propose, permettant ainsi de décomposer un domaine d'affaire en plusieurs sous domaines indépendants. Ce patron de conception est fortement recommandé, notamment pour les systèmes complexes de par sa similarité conceptuelle avec la notion même de microservices.

2.2.1. Le Domain-Driven-Design

Initialement introduit et rendu populaire par Eric Evans en 2003 dans son livre « ***Domain-Driven Design : Tackling Complexity in the Heart of Software*** », la conception pilotée par domaine, DDD, est une approche de conception préconisée pour le développement de systèmes complexes axés sur la cartographie d'activités, de tâches, d'événements et de données d'un domaine métier.

Le Domain-Driven Design met l'accent sur la compréhension du domaine afin de créer un modèle abstrait de celui-ci pouvant ensuite être mis en œuvre dans un ensemble particulier de technologies. Cette méthodologie fournit des indications sur la manière dont le développement de modèle et le développement de technologie peuvent aboutir à un système qui répond aux besoins de ceux qui l'utilisent tout en restant robuste face aux changements dans le dit domaine.

Les processus du Domain-Driven Design impliquent une étroite collaboration entre les experts du domaine (les personnes connaissant les problématiques liées au domaine) et des experts en conception / architecture/ développement (les personnes connaissant les solutions pouvant être apportées pour résoudre les problématiques liées au domaine). La finalité étant de disposer d'un modèle commun avec un langage commun de sorte que les différents interlocuteurs quel que soit leur domaine d'expertise (développeurs, analystes, commerciaux, clients etc.) puissent échanger sur une base de connaissance partagées avec des concepts partagés.

Le Domain Driven Design permet de remédier à toute difficultés profondes liées à la compréhension du domaine d'étude lors de la conception d'application complexe en connectant des éléments connexes

du logiciel dans un modèle en constante évolution. C'est plus que d'avoir un modèle objet, l'accent est mis sur la communication partagée et l'amélioration de la collaboration afin que les besoins réels dans le domaine étudié puissent être découverts et qu'une solution appropriée soit créée pour répondre à ces besoins. Ceci de façon agile.

Le DDD se repose sur un ensemble de principes parmi lesquels nous pouvons citer :

- Le langage omniprésent : ce principe de DDD met l'accent sur le fait de briser les barrières de la communication entre les experts du domaine et les spécialistes de l'informatique en les faisant travailler ensemble sur le modèle du domaine. Ceci devrait permettre de surmonter la différence de style de communication, d'échanger sur les éléments impliqués dans le modèle, de la manière dont ils sont reliés les uns aux autres et de la pertinence de certains. La création du langage omniprésent devrait permettre aux différents interlocuteurs quel que soit leur domaine d'expertise (développeurs, analystes, commerciaux, clients etc.) d'échanger sur une base de connaissance partagées avec des concepts partagés

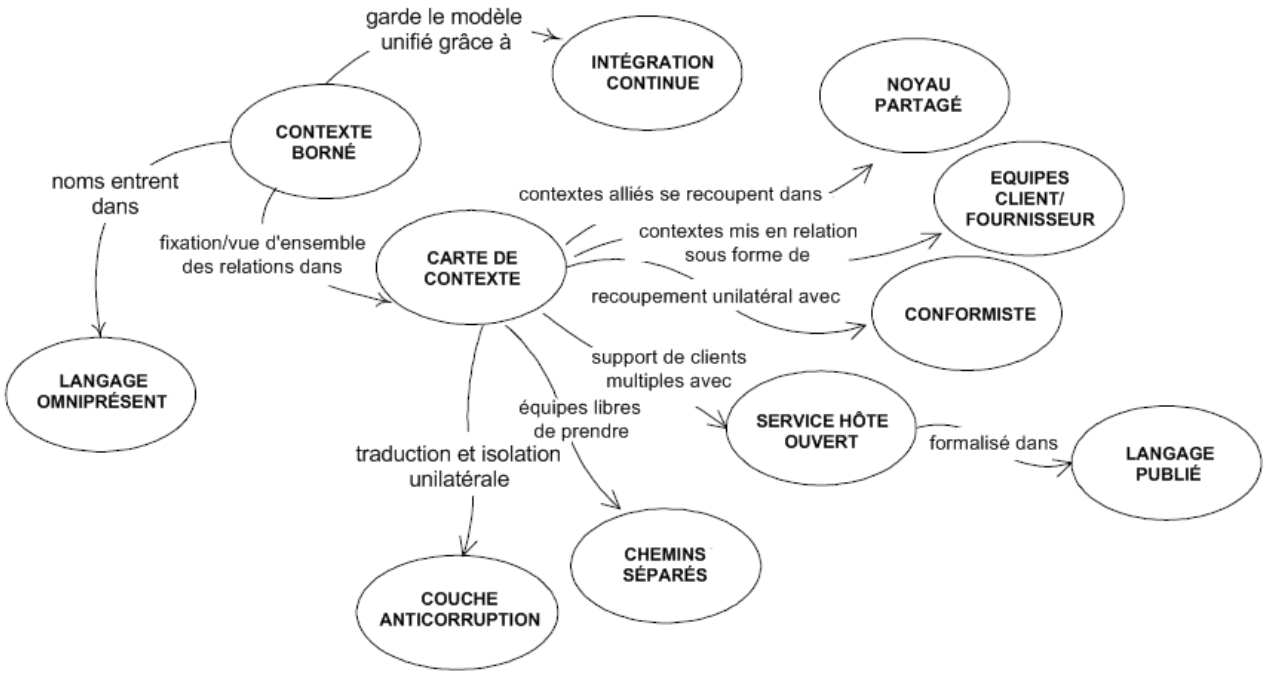
- La conception dirigée par le modèle : Dans la continuité du point précédent, ce principe énonce les avantages à ce que tous les acteurs du projet travaillent ensemble sur le modèle de conception. Car il arrive que les analystes détaillent beaucoup certains aspects du modèle et pas suffisamment d'autres. A ajouter qu'à cela les développeurs ont tendance à utiliser le modèle d'analyse comme une référence pour créer un modèle de conception qui s'éloigne de l'analyse avec le temps. L'objectif de rassembler tous les acteurs durant cette phase étant de combler les idées des uns et des autres tant sur le point de l'analyse que de la conception du modèle. Cette phase devrait permettre d'obtenir un modèle utile suite à l'obtention d'éléments de conception tels que :

    - Les blocs de constructions d'une conception orientée modèle
    - L'architecture en couches
    - Les entités
    - Les objets-Valeurs
    - Les services
    - Les modules
    - Les agrégats
    - Les Fabriques
    - Les entrepôts

- Le refactoring profond : L'auteur traite de ce principe en énonçant que le refactoring est quelque chose d'inévitable lors de la phase de conception d'un logiciel. Il est important de s'arrêter régulièrement pour inspecter le code et de le reconcevoir en vue de l'améliorer sans introduire de bugs. Le refactoring ajoute de la clarté au design et des conditions pour une avancée majeure (modification impactant fortement le modèle), qui est une source de grand progrès pour un projet. Toute avancée majeure implique un vaste refactoring et par conséquent du temps et des ressources (pas toujours disponible) et c'est de là que vient la notion de refactoring profond dans la mesure où il faut se repencher sur le modèle et rendre explicite tous les éléments implicites qui ont permis de conclure à un changement drastique de la conception. À savoir que cela rajoutera de la souplesse et de la modularité au logiciel.

- La préservation de l'intégrité du modèle : Ce principe est particulièrement applicable aux projets où l'on a confié à plusieurs équipes dans des conditions de management, de coordination et de technologies diverses, la tâche de développer une solution logicielle. Chaque équipe développe du code en parallèle en se voyant assigner une partie du modèle. Chacune de ses parties doivent pouvoir communiquer avec les autres sans altérer les données. L'ensemble des techniques suggérées par le DDD à combiner pour maintenir l'intégrité du modèle sont :

    - Les définitions des contextes bornés
    - L'intégration continue
    - La mise en place de la carte des contextes
    - Le noyau partagé
    - La compréhension du client-fournisseur
    - Le conformisme



*Source : « **Domain-Driven Design : Tackling Complexity in the Heart of Software** », Eric Evans.*

Ce schéma crée par l'auteur, permet de montrer l'ensemble des techniques de DDD et les interrelations entre elles.

Le DDD étant particulièrement vaste, les projets en architecture de microservices ont une nécessité à implémenter le point sur la *préservation de l'intégrité du modèle* car il répond aux différentes problématiques liées à ce type d'architecture et spécifiquement à la difficulté de découper le modèle aux bons endroits afin de limiter les dépendances. Dans le cadre de l'étude ce document, il est deux notions clés sur lesquels les experts de ces architectures s'accordent à dire qu'il ne faudrait pas faire d'impasse, il s'agit des notions de contexte borné et carte de contextes.

2.2.2. Les contextes bornés

Chaque domaine réfère à un contexte précis, c'est-à-dire à un ensemble de conditions qu'on doit appliquer pour s'assurer que les termes, notions et interrelations utilisées prennent un sens précis.

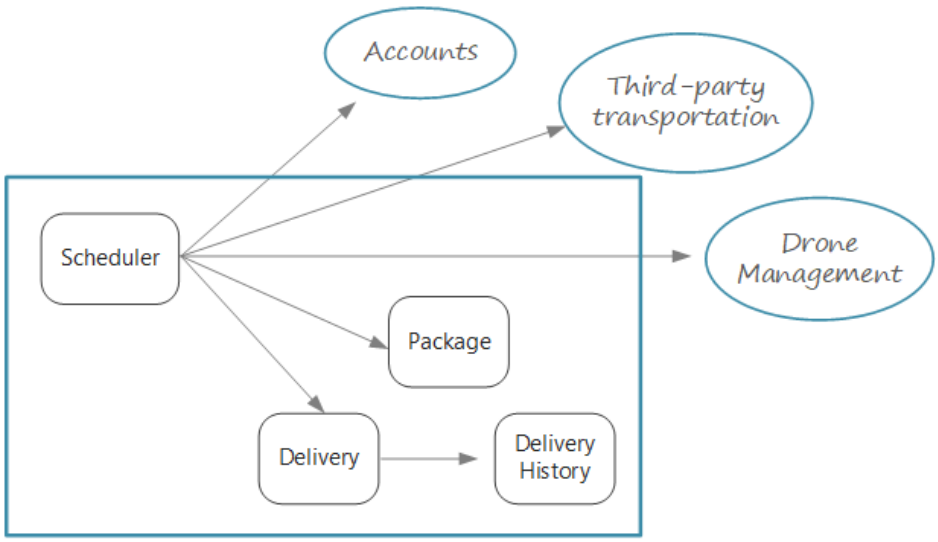
Ce contexte peut être constitué d'un ou plusieurs modèles, de ce fait peu importe le modèle, il est toujours lié à un contexte précis. Cependant dans les grandes applications d'entreprises, plusieurs

modèles entrent en jeu. En combinant les morceaux de codes basés sur les modèles distinct, on obtient une solution buggée, peu fiable, difficile à comprendre et de ce fait difficile à maintenir.

Le travail qui consiste à scinder les modèles en modèles plus petit est d'essayer de regrouper les éléments liés par un concept naturel, la finalité étant d'obtenir un modèle assez petit et suffisamment autonome pour pourvoir être assigné à une seule équipe.

Le contexte borné est le cadre logique à l'intérieur duquel sera amené à évoluer ce type de modèle, Un contexte borné devrait donc permettre :

- De connaitre les limites du modèle
- A une équipe de rester autonome à l'intérieur de ce périmètre
- De faciliter la pureté, la cohérence et l'unicité d'un modèle
- Faciliter le refactoring sans répercussion sur les autres modèles
- D'avoir une visibilité très spécifique sur une partie du domaine
- De limiter les dépendances avec les autres modèles en gardant uniquement les point de liaison explicites
- De faciliter la gestion des impacts
- D'englober un ou plusieurs modules permettant de développer des microservices



Source : https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries

Dans cet exemple d'analyse de domaine pour le scénario de livraison de drone sur la plateforme Microsoft Azure, on met l'accent sur le contexte de livraison afin de montrer les limites du modèle et ce qu'il intègre ainsi qu'une communication directe avec d'autres contextes bornés.

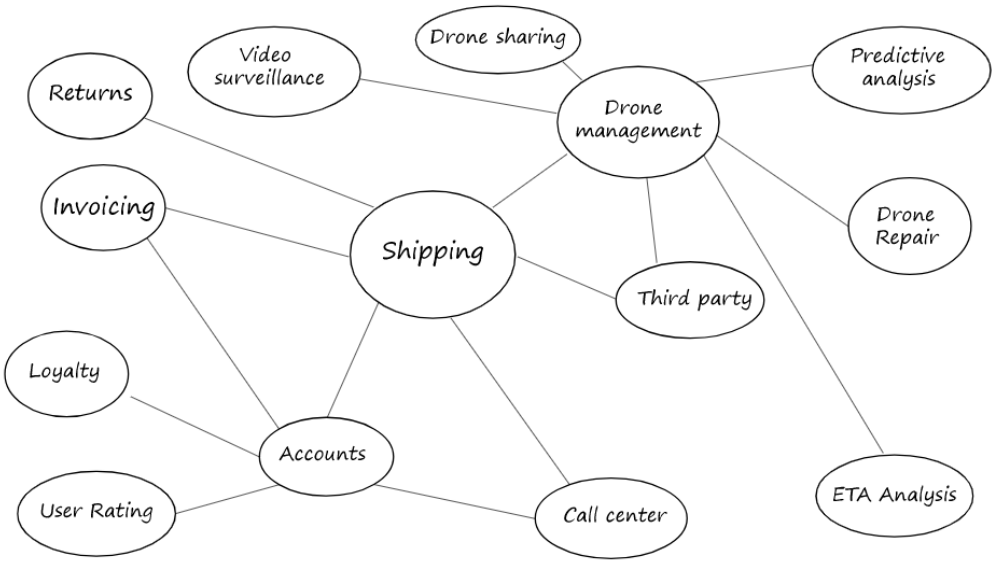
2.2.3. La carte des contextes

Dans une stratégie qui consiste à avoir de multiples modèles ayant chacun un contexte borné, on a une visibilité très spécifique sur certaines parties du domaine, cela permet de mieux encadrer les impacts et gérer les dépendances. Les individus d'une même équipe, communiquent plus facilement et travaillent plus efficacement à intégrer un modèle au sein d'un contexte borné.

Une carte de contexte est un diagramme, un schéma ou n'importe quel document écrit avec un niveau de détail variable, qui permet de mettre en évidence les différents contextes bornés, les langages utilisés et les liens entre eux. Car même si chaque équipe travaille sur son modèle il est important que toutes les équipes aient une vision d'ensemble sur le projet et puisse se situer géographiquement sur l'ensemble de tout le périmètre du projet.

L'importance d'une carte de contexte est de constituer une visibilité globale du domaine et de cartographier les liens entre les contextes bornés et par conséquents les modèles. Cela devrait permettre de mieux penser la stratégie d'organisation du code, des ressources et des équipes de développements afin de prendre des responsabilités sur certains sujets

La carte de contexte est fortement liée à la notion d'intégration du système global. La finalité étant qu'une fois toutes les pièces rassemblées, le système fonctionne correctement. Cela permet aussi de vite constater les systèmes qui se chevauchent ou qui sont susceptible de l'être. Si les liaisons entre les contextes ne sont pas clairement définies et mis en évidence, le risque d'échec au niveau de l'intégration du système est accru.



Source : https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis

Dans cet exemple d'analyse de domaine pour le scénario de livraison de drone sur la plateforme Microsoft Azure, on représente la carte de tous les contextes permettant la mise en place de l'application de gestion de livraison d'un drone, ainsi que tous les liens entre ces derniers.

En somme, mettre en place la DDD dépend du niveau d'isolation et d'encapsulation que l'on souhaite conserver au niveau du modèle. Aussi il faut tenir compte du coût d'entrée qui va augmenter car il faut absolument être formé à la méthode pour intervenir sur un projet désigné.

2.3. Communication entre les microservices

Après un travail de conception sur le découpage fonctionnel qui permet de distinguer les premiers microservices et API, on doit se pencher sur l'importante question de savoir comment on intègre les microservices au sein d'une architecture et comment ces derniers pourraient communiquer entre eux.

Plus qu'une architecture, la MSA est un ensemble de concepts fonctionnels, techniques et organisationnels. Et en ce qui concerne l'intégration il y'a un ensemble de bonnes pratiques dont il faut tenir compte et qui pourraient fortement influencer les choix technologiques conformément à la solution que l'on souhaitera implémenter :

- Garder les APIs « technology-agnostic » : Etant donné que le marché est en constante mutation, les communications entre les services ne doivent pas dépendre d'une stack technologique en particulier mais de normes connues de tous. N'importe quel client, peu importe sa technologie devrait pouvoir accéder à une API et n'importe quel microservice, peu importe sa technologie devrait pouvoir exposer des ressources via une API.

- Rendre le service simple pour les consommateurs : Un service doit être le plus simple et exposé le plus rapidement possible pour laisser le choix à l'utilisateur de l'intégrer à son modèle technique. Cette simplicité peut s'avérer utile quand il s'agit de mettre à jour le service ou d'en recréer un autre sur la base de ce dernier.

- Masquer les détails de l'implémentation : ceci est une recommandation pour éviter un couplage entre un consommateur et la mise en œuvre externe du service afin de ne pas casser la liaison de communication et /ou d'avoir des mises à jour obligatoires et fréquentes du côté du consommateur.

Partant de ces objectifs, certains processus et technologies ont évolué, émergé, gagné en popularité et sont très souvent associés en premier aux architectures de microservices.

2.3.1. Les API REST (REpresentationnal State Transfer)

Emergence

C'est en 2000 que **Roy Fielding**, à l'époque directeur de la fondation Apache, a introduit le REST lors de sa thèse de doctorat « **Architectural Styles and the Design of Network-based Software architectures** » à l'université de Californie à Irvine.

Dans le chapitre 5 de cette thèse, qui y est particulièrement consacré, Fielding défini le REST comme un style architectural basé sur un ensemble de contraintes permettant de créer des services web. Lesquelles sont :

- **Une communication Client-serveur** : le client doit être séparé du serveur
- **Une communication sans état** : Une requête doit contenir toutes les informations nécessaires à son exécution et les perd une fois utilisées.
- **La mise en cache** : une réponse du serveur doit contenir des informations permettant au client de mettre en cache ladite réponse.
- **Des interfaces uniformes** : les interfaces doivent permettre d'identifier les ressources disponibles.
- **Un système hiérarchisé en couche** : un client doit pouvoir se connecter à un serveur final ou à un intermédiaire sans qu'il ne s'en aperçoive.
- **Du code à la demande** : cette contrainte permet d'exécuter des scripts récupérés à partir du serveur.

Avant l'apparition du REST les normes pour concevoir une API étaient approximatives et pas particulièrement formalisées, de plus, leur intégration nécessitait l'utilisation de protocole tels que SOAP qui étaient notoirement complexe à créer, à gérer et à corriger. Bien que le REST impose beaucoup de règles, la plupart d'entre elles sont universelles, la finalité étant d'avoir des APIs plus simples et faciliter considérablement l'intégration.

- Les premières API REST

Une API **RESTFul** est une API donc la conception, l'intégration et l'exploitation respectent l'ensemble des contraintes du REST et les premiers à s'intéresser à ce phénomène ont été des géants du commerce électronique telque **EBay**, suivis **d'Amazon**.

L'accès à l'API REST d'EBay, facile à utiliser et disposant d'une documentation solide, a été offert à une sélection de partenaires. Cela a montré à quel point les APIs nouvellement accessibles pouvaient être lucratives. Par conséquent, sa place de fournisseur sur le marché n'était plus limitée aux seuls visiteurs sur son site Web, mais à tout site Web accédant à son API.

L'avantage était évident : une visibilité accrue de son offre de produits et donc, une augmentation considérable des opportunités de vente ! La simplicité du système a immédiatement séduit d'autres plateformes en ligne qui ont commencé à réfléchir à la valeur de leur code et non plus uniquement à leurs produits grand public

- L'API Economy

En 2004, **Flickr** a lancé sa propre API REST juste à temps pour la montée en puissance des réseaux sociaux et des blogs. Devenant rapidement la plate-forme photo de référence, Flickr a ainsi ouvert la voie au partage social auquel **Facebook** et plus tard **Twitter** ont rapidement adhéré. La demande d'API publique a augmenté, des API REST facilement accessibles permettent désormais à tous d'ajouter une fonctionnalité à un site Web en un temps record.

En 2006, **Amazon** a contribué au lancement du cloud grâce à son API REST. Depuis lors, les APIs REST sont devenues la colonne vertébrale d'Internet et les créateurs d'énormes opportunités commerciales en raison de leur capacité à étendre la portée d'une marque au-delà du public d'un site Web. Au cours des dix dernières années, le nombre d'API disponibles au public a donc été multiplié par 50.

- Les perspectives d'avenir

De nos jours, la construction de logiciels ne nécessite plus une équipe d'ingénieurs ou de serveurs coûteux. Une clé d'API et sa documentation sont majoritairement ce dont on a besoin pour intégrer facilement une fonctionnalité existante et disponible à l'externe.

Sachant que les utilisateurs finaux des APIs sont les développeurs, les APIs ont été de plus en plus simplifiées au fil du temps pour repousser le maximum de limites et de laisser libre cours à la mise en place de solutions de plus en plus ingénieuses. De ce fait le nombre d'outil pour automatiser certains processus lors de la conception des APIs ont été automatisés (Exemple : générer la documentation d'une API)

L'utilisation et l'exploitation des APIs REST est de plus en plus pris en compte dans le modèle économique des entreprises dans les aspects tant B2C que B2B. Les APIs ont accru les affaires de sociétés de logiciels et sont devenues des éléments essentiels du modèle commercial. Les APIs sont non seulement en train de changer la face du Web mais aussi celle du modèle économique de nombreuses entreprises.

REST vs SOAP

REST et SOAP sont des éléments souvent comparés à tort l'un à l'autre dans la conception des applications client-serveur. REST est un style architectural qui définit un ensemble de contraintes à respecter si l'on souhaite fournir des services web dit RESTful, par exemple les transmissions sans état et l'utilisation du HTTP, tandis que SOAP est un protocole dont les spécifications sont des normes Web officielles, maintenues et développées par le World Wide Web Consortium (W3C).

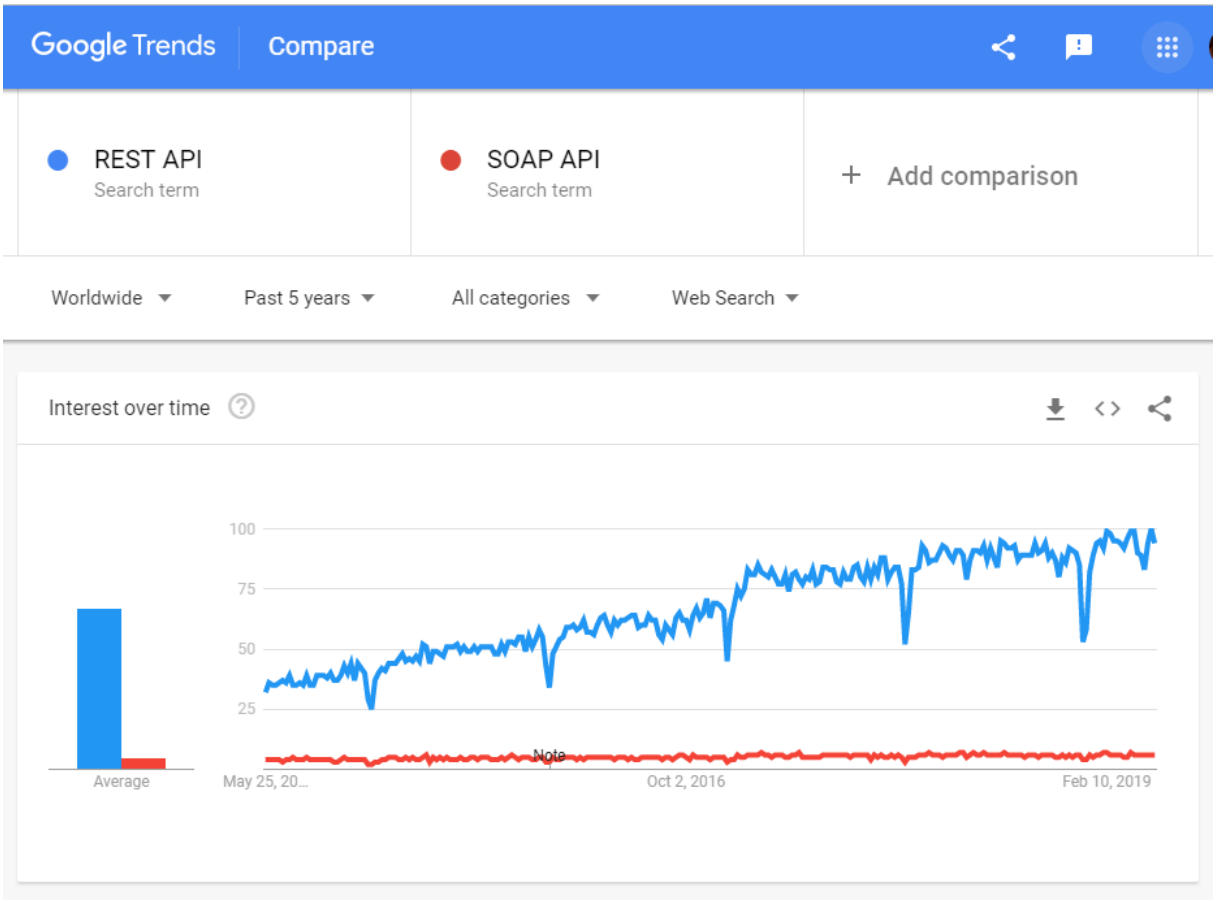
En partant du principe que, dans la conscience collective, SOAP et REST symbolisent deux concepts de création de services web qui abordent la question de transmission de données d'un point de vue différent, les deux styles permettent de créer des APIs. Le tableau comparatif ci-dessous permet de mettre en évidence les caractéristiques de chacun et ainsi aider à la prise de décision pour choisir l'un ou l'autre des styles :

	SOAP	REST
Définition	Simple Object Access Protocol	REpresentational State Transfer
Conception	Protocole standardisé avec des règles prédéfinies à suivre	Style architectural avec des guidelines et recommandations
Approche	Axée fonction : les données sont accessibles comme services, exemple : getUSer	Axée Ressource : les données sont accessibles comme ressources, exemple : User
Gestion des états	Sans état par défaut. Il est à noter qu'il est possible de rendre une API SOAP stateful	Sans état uniquement. Il n'y a pas de gestion des sessions coté serveur
Caching	Les appels d'API ne peuvent pas être mise en cache	Les appels d'API peuvent être mise en cache
Sécurité	Intègre les normes de sécurité du protocole WS-Security, offre	Supporte HTTPS et SSL

	un support SSL et intègre les conformités ACID	
Performance	Requiert plus de bande passante et de puissance machine	Requiert moins de ressources
Format des messages	XML uniquement	Du texte, HMTL, XML, JSON, YMQL, PDF, et autres MEDIA
Protocoles de transfert	HTTP, SMTP, UDP et d'autres	HTTP uniquement
Recommandations d'utilisation	Les applications d'entreprises, les applications qui demande une haute sécurité, les environnements distribués, les services financiers, les passerelles de paiement, les services de télécommunication	Les APIs publiques, les services web, les services mobiles, les réseaux sociaux
Avantages	Une grande sécurité, une grande standardisation	La scalabilité, une grande performance, pour l'amélioration de l'expérience utilisateur (navigateur friendly), la flexibilité
Inconvénients	Moins performant, plus complexe et moins flexibles	Moins sécuritaire

SOAP étant un protocole officiel, il est soumis à des règles strictes et à des fonctionnalités de sécurité avancées telles que la conformité et l'autorisation ACID intégrées. Plus complexe, il nécessite plus de bande passante et de ressource, ce qui peut ralentir les temps de réponses et de chargement des pages. REST a été créé pour résoudre les problèmes de SOAP. Par conséquent, il ne s'agit que de directives générales permettant aux développeurs de mettre en œuvre les recommandations à leur manière. Il autorise différents formats de messagerie tels que le HTML, JSON, XML et du texte brut, tandis que SOAP autorise uniquement le XML.

REST étant également une architecture plus légère, les services web RESTful offrent de meilleures performances. A cause de cela, le REST est devenu particulièrement populaire, surtout à cette ère de la téléphonie mobile où même quelques secondes (notamment en ce qui concerne le temps de chargement des contenus) comptent énormément pour l'amélioration de l'expérience utilisateur.



Source : statistiques Google Trends

Représentations pour les tendances des API REST vs SOAP sur les 5 dernières années dans le monde entier : la tendance est aux REST comme choix de conception des API

Le modèle de maturité de Richardson

> *I am getting frustrated by the number of people calling any HTTP-based interface a REST API. […]. Please try to adhere to them or choose some other buzzword for your API*

Source : https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Article du 20/10/2008 du blog de Roy Fielding, où l'auteur insiste sur la différence entre une API basée sur le HTTP et une API REST

Dans les lignes précédentes, une API RESTful a été désignée comme une API respectant l'ensemble des contraintes du REST cependant dans la pratique, toutes les règles ne sont pas exploitées. Dans l'article

de son blog, **« Untangled *musing of Roy T. Fielding* »**, l'auteur insiste sur le fait que tant qu'une API n'est pas hypertext-driven, autrement dit, quand les données remontées par un appel ne permettent pas d'avoir des liens vers les autres données de l'API, il ne s'agit pas d'une API REST, mais plutôt d'une simple API HTTP.

Cette notion d'HyperText vient de **Léonard Richardson**, un expert de la conception des API RESTful (développeur de la librairie en Python appelée *Beautiful Soup*), qui a créé un modèle qui décompose les principaux éléments d'une approche REST en 4 niveaux (0-3) permettant d'évaluer une API par rapport aux contraintes REST. Ce modèle introduit successivement les ressources, les verbes HTTP et les contrôles hypermédia :

Niveau de maturité :	Caractéristiques :
Nveau 0	Le RPC sur HTTP en Plain Old XML : - Le protocole ne sert qu'à transporter le message - Tout circule via un seul point d'entré - Chaque requête contient l'action à effectuer, l'objet cible sur lequel va porter l'action et tout autre paramètres nécessaires à l'exécution de l'action.
Niveau 1	Introduction des Ressources : - Les ressources sont identifiées via des URI - Pas de sémantique particulière lors des appels - L'identifiant d'une ressource est obligatoirement rangé dans un champ nommé « Id » - Chaque requête contient l'action à effectuer auprès de la ressource ainsi que les paramètres nécessaires à l'exécution de toute action sur ces ressources
Niveau 2	Utilisation des verbes et codes de retours HTTP : - En mode multi ressources - L'utilisation des verbes HTTP (GET, POST, DELETE, PUT) est sémantiquement correcte - L'exploitation des codes de retour HTTP est correcte
Niveau 3	Le contrôle Hypermédia :

- Basée sur le principe d'HATEOAS (HyperText As The Engine Of Application State)
- Les réponses contiennent des liens permettant de parcourir les ressources de l'API
- Les opérations sont indiquées sous formes d'hyperliens

Avec le modèle de maturité de Richardson, les API de type 0 et 1 n'utilisent que des POST, au niveau 2 l'API parait plus formalisé et au niveau 3 on a une API auto descriptive grâce à la contrainte HATEOAS qui permet d'indiquer dans la réponse à une requête GET toutes les autres opérations possibles sur l'API.

L'intérêt du niveau du MMR, n'est pas nécessairement d'implémenter le niveau le plus élevé pour une API, mais juste pour voir où l'on se situe et ce qu'il faut améliorer, force est de constater que **la moyenne des API sur le marché sont de niveau 2.**

Avantages et inconvénients

Les architectures REST, que ce soit dans le cadre des microservices ou pas, ont l'avantage :

- D'être plus facile à mettre en œuvre que les alternatives classiques.
- De ne nécessiter que l'utilisation d'un navigateur pour accéder aux ressources d'un services.
- De permettre une mise en cache des ressources pour accélérer certaines opérations.
- De ne pas consommer excessivement la mémoire.
- De repartir les requêtes sur plusieurs serveurs, notamment grâce à l'absence d'état.
- De permettre l'utilisation d'un vaste panel de format de données (JSON, XML, Atom, etc.).
- De permettre l'échange des requêtes entre diverses application ou média grâces aux URLs.
- De disposer des avantages du HTTP (redirection, cache) qui est supporté par plusieurs plateformes et plusieurs technologies.

Il faut cependant tenir compte que toute cette flexibilité nécessite aussi de garder une certaine vigilance sur certains aspects :

- Ne pas perdre de vue qu'il s'agit d'une architecture orientée ressource et en créant les services, il y a une résistance au changement qui consiste à raisonner en en termes de

fonctions, or ce type de raisonnement amène à créer des fonctionnalités qui multiplie les appels.

- Les données nécessaires à l'utilisation d'un service REST doivent être conservées localement.
- Un modèle orienté ressource nécessite la création d'un modèle de données robuste.
- En matière de sécurité, les architectures REST sont moins sures que celles basées sur le protocole SOAP et nécessite des ajustements via une expertise en sécurité.

En somme, à l'instar de type de communication tels que le SOAP, XML-RPC et certains protocoles tampons, la mise en place du REST dans les architectures de microservices respecte en quelque sorte l'objectif de technologie agnostique.

A surveiller : Le **gRPC** est un système d'appel de procédure à distance open source développé par Google et présenté en ce moment par opposition au REST. Il découle de l'infrastructure **RPC** polyvalente interne de Google, nommée **Stubby**, qui connectait tous les microservices de Google exécutés au sein de ses centres de données et qui a été retravaillé pour étendre son applicabilité au mobile, à l'IoT, au cloud et créer une infrastructure open source normalisée. Le gRPC utilise le format **Protobuf** (.proto) comme format d'échange de données qui est un format JSON (voir point 2.3.2) compacté. Il prend en charge plusieurs langages de programmation tels que Java, C#, C++, Python, Ruby, Javascript et Objective-C. Tandis que le REST utilise le protocole **HTTP 1.1**, le gRPC dépend du protocole **HTTP /2** qui permet une communication bidirectionnelle et multiplexée, ce qui permet une circulation plus rapide et plus continue des flux de données. Le gRPC offrirait une meilleure performance et sécurité que du REST+JSON, il est fortement recommandé par SSL/TLS pour authentifier un serveur et chiffrer toutes les données échangées. Le gRPC devient un framework de plus en populaire pour les microservices, car moins d'un an après son lancement, il a été adopté par CoreOS, Netflix, Square, Cockroach Labs, Cisco, pour n'en nommer que quelques-uns.

Plus d'informations sont disponibles sur www.grpc.io

2.3.2. Le JSON, format d'échange de données préférentiel

L'architecture REST permet aux fournisseurs d'API d'exposer des données dans plusieurs formats, tels que du texte brut, du HTML, du XML, du YAML et du JSON, ce dernier est l'un des plus appréciés.

Grâce à la popularité croissante de REST, le format JSON qui signifie **JavaScript Object Notation**, qui est un format d'échange de données léger, lisible et facilement analysable par l'homme, a également rapidement gagné du terrain, car il convient parfaitement à des échanges rapides et est très intuitif.

Bien que son nom ne le laisse pas sous-entendre, le JSON est totalement indépendant du langage Javascript et peut donc être utilisé avec n'importe quel langage de programmation. Sa syntaxe est un sous-ensemble de la 3eme édition de la **norme ECMA-262**. Les fichiers JSON se composent de collections de paires nom / valeur et de listes de valeurs ordonnées qui sont des structures de données universelles utilisées par la plupart des langages de programmation. Par conséquent, JSON peut être facilement intégré à n'importe quel langage.

Historique

Le JSON est un format d'échange de données né d'une association entre le langage Javascript et le Scripting coté client. Il a été inventé entre 2002 et 2005 par **Douglas Crockford**, architecte logiciel chez PayPal, connu pour ses apports conséquents au développement du langage Javascript, notamment pour ce qui est de la création de l'outil ***JSLint*** qui permet de détecter les erreurs de syntaxes et de mauvaise pratique lors de l'utilisation du Javascript comme langage de programmation.

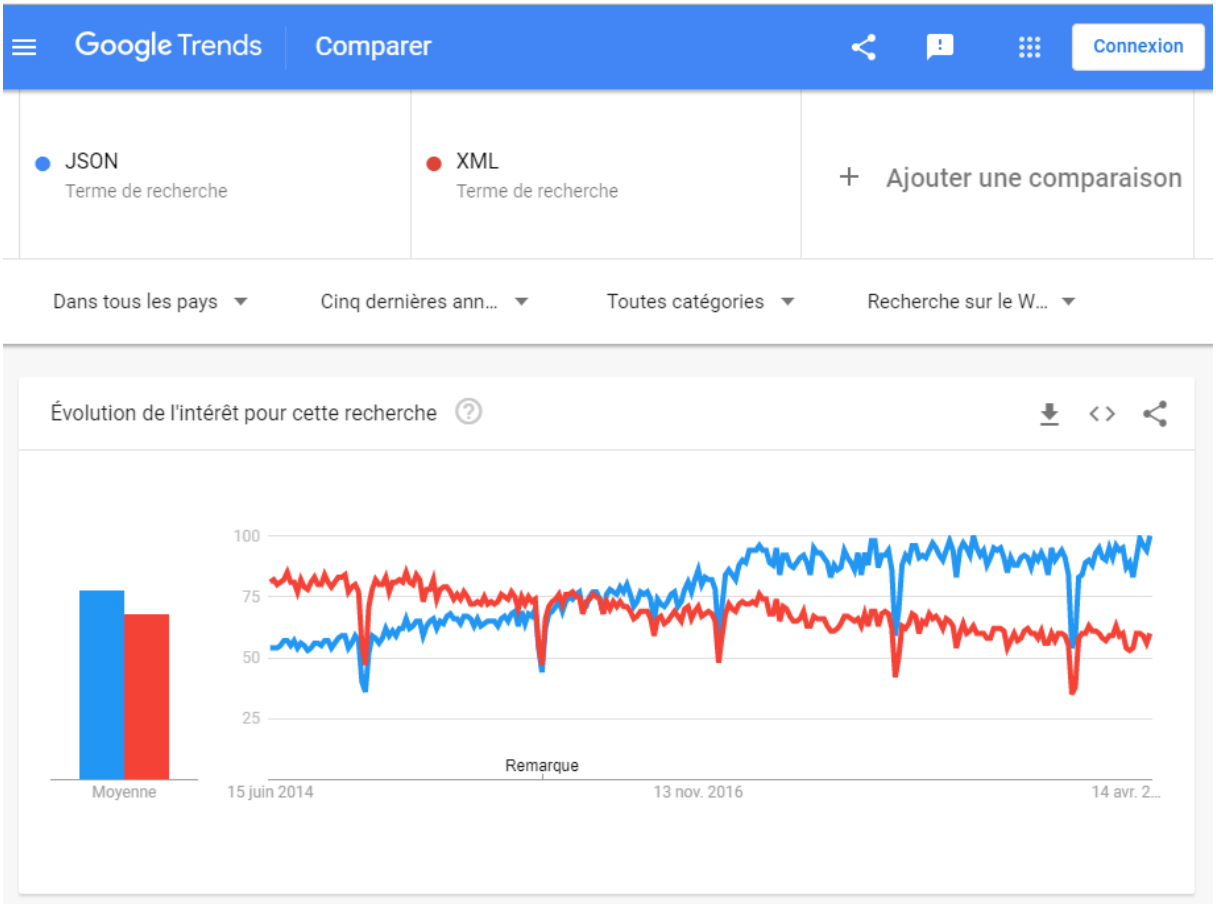
Le JSON est né d'un besoin d'avoir une communication sans état et en temps réel entre le serveur et le navigateur sans l'utilisation de plugins non sécuritaire tels que Flash ou les applets Java, qui dominaient le marché au début des années 2000. L'idée étant de créer un système utilisant les fonctionnalités du navigateur standard et fournissant une couche d'abstraction aux développeurs Web pour la création d'applications Web avec état. Ceci était fait en maintenant une connexion duplex persistante à un serveur Web en maintenant deux connexions HTTP ouvertes et en les recyclant avant les délais d'expiration standard du navigateur si aucune donnée n'était échangée. Crockford a découvert que le langage Javascript pouvait être utilisé comme format de messagerie objet lors de la mise en place d'un tel système.

Le système a été vendu à Sun Microsystems, Amazon et EDS, le site web JSON.org a été lancé en 2002. En 2005, Yahoo a commencé à proposer certains de services Web en JSON, et a été suivi de près en 2006 par Google dans cette démarche.

A l'origine destiné à être un sous-ensemble de Javascript avec la 3ème édition de la norme **ECMA-262** de l'ECMA (European Computer Manufacturers Association), un organisme de standardisation dans le domaine informatique au niveau de l'Europe, on s'est vite rendu compte que le JSON pouvait être intégré à n'importe quel langage de programmation. Ainsi à l'heure actuelle, la syntaxe du JSON est actuellement décrite par deux normes concurrentes : **RFC 8259** de l'IETF (Internet Engineering Task Force) pour les standards internet en Amérique du nord et **ECMA-404** de l'ECMA.

Bien que le XML soit toujours très utilisé pour décrire les données, le JSON a beaucoup gagné en popularité du fait de sa légèreté et de la facilité d'analyse car très intuitif pour l'humain. De plus, selon Douglas Crockford, ce format présente plusieurs avantages par rapport au XML

JSON vs XML



*Source : **Google Trends.***

Analyse comparative de l'utilisation du JSON et du XML sur les 5 dernières années dans le monde

Cette analyse statistique proposée par Google Trends permet de mettre en évidence une certaine popularité montante du JSON. Bien que le XML ait été standardisé en 1998 pour structurer les pages HTML, essayons de comparer les deux standards sur certaines bases de leur utilisation :

Base de comparaison	JSON	XML
Définition	Javascript Object Notation	eXtensible Markup Language
Origine	Dérivée du langage Javascript	Dérivée du SGML
Applicabilité	Transmettre les données de manière analysable via Internet	Pour que les données soient structurées de manière que l'utilisateur puisse utiliser pour annoter les métadonnées, analysez les scripts.

Code de représentation des objets	{
	"Paragraphs": [
	{
	"align": "center",
	"content": [
	"Here ",
	{
	"style": "bold",
	"content": [
	"is"
	]
	},
	"some text"
	]
	}
	]
	}
		<Document>
		<Paragraph Align = "Center">
		Here <Bold> is </Bold> some text
		</Paragraph>
		</Document>

Représentation des éléments en hiérarchie	{
	"firstName": "Mr.",
	"lastName": "A",
	"details": [
	"Height",
	"Weight",
	"Color",
	"Age",
	"Sex",
	"Language"
	]
	}
		<Person>
		<FirstName>Mr</FirstName>
		<LastName>A</LastName>
		<Details>
		<Detail>Height</Detail>
		<Detail>Weight</Detail>
		<Detail>Color</Detail>
		<Detail>Age</Detail>
		<Detail>Sex</Detail>
		<Detail>Language</Detail>
		</Details>
		</Person>

Raison de la popularité	Moins verbeux et rapide	Très verbeux (plus que nécessaire). Travail d'analyse fastidieux. Couts en termes de consommation de mémoire
Structure de donné	Sous forme de carte. La carte est similaire aux paires clé / valeur et est utile lorsque l'interprétation et la prévisibilité sont nécessaires.	Sous forme d'arbre. Il s'agit d'une représentation arborescente des données. Rend l'analyse fastidieuse.
Information de données	A préférer pour la transmission de données entre serveurs et navigateurs	A préférer pour le stockage des informations coté serveur
Communication navigateur-serveur	OK	OK
Marquage des métadonnées	Fastidieux : Il faut transformer une entité en un objet ensuite l'attribut doit être ajouté en tant que membre de l'objet.	Simple : utilisation des balises
Contenu mixte (les chaines contenant des balises structurées)	Fastidieux : Il faut transformer une entité en un objet ensuite l'attribut doit être ajouté en tant que membre de l'objet.	Simple et efficace : placer le texte baliser dans une balise enfant du parent auquel il appartient
Gestion des namespaces	Aucun support	Supporté
Gestion des listes	Supporté	Non supporté
Sécurité	Moins sécurisé	Plus sécurisé
Gestion des commentaires	Non supporté	Supporté
Encodage	UTF-8 seulement	Varié

Au vu de ces éléments, on peut conclure que le JSON et le XML sont tous les deux un moyen d'organiser les données dans un format compréhensible pour de nombreux langages de programmation et les APIs. Ces deux types, sont pour la plupart des cas, utilisés au sein de même applications. Ce qui est certain c'est que le XML est définitivement plus ancien et largement répandu dans les applications

d'entreprises. Récemment le JSON a gagné en popularité auprès de sa communauté très active d'utilisateurs (en grande partie, les développeurs) en raison de l'essor du JavaScript.

Aucune de ces solutions n'est véritablement supérieure à l'autre. Cependant, elles s'accordent chacune à des cas d'utilisation bien précis :

- Le JSON est plus simple pour extraire des données sur un serveur et les manipuler. Il faut maitriser la structure des données pour l'utiliser (Être propriétaire des données). Il est léger et économise les ressources. Il est particulièrement prisé par les utilisateurs du JavaScript et peut être pluggé à plusieurs langages de programmations, ce qui peut particulièrement faciliter les développements.
- Le XML, bien que verbeux, convient mieux quand il s'agit de représenter les données, il peut être utilisé en provenance de sources externes et créer des bases de données. Il existe une multitude d'outils d'aide pour traiter le XML et c'est le format de traitement de documents. Il reste aujourd'hui encore le langage de nombreuse interfaces graphiques.

Avantages et inconvénients

Le principal avantage du JSON et à qui il doit sa popularité est sa complétude et sa simplicité de mise en œuvre dans le cadre d'un développement :

- Peu verbeux, il est facilement lisible par l'être humain et interprétable par la machine
- La montée en compétence sur la prise en main du langage est très rapide (Exemple : en 1h sur Udemy), car la syntaxe bien que limitée est réduite et non extensible
- Les données sont faciles à décrire
- Presque tous les langages de programmations ont un plugin pour décrire les données au format JSON
- Dans le cadre des micro-services il est très utile pour la communication des applications dans un milieu hétérogène.

Cependant dans certains cas d'usage il faut tenir compte des inconvénients du langage :

- Le langage est limité à quelles types généraux sans possibilité de les étendre (Exemple : les dates, les couleurs, etc.)
- Le typage faible est la principale faille de sécurité et de fiabilité du langage

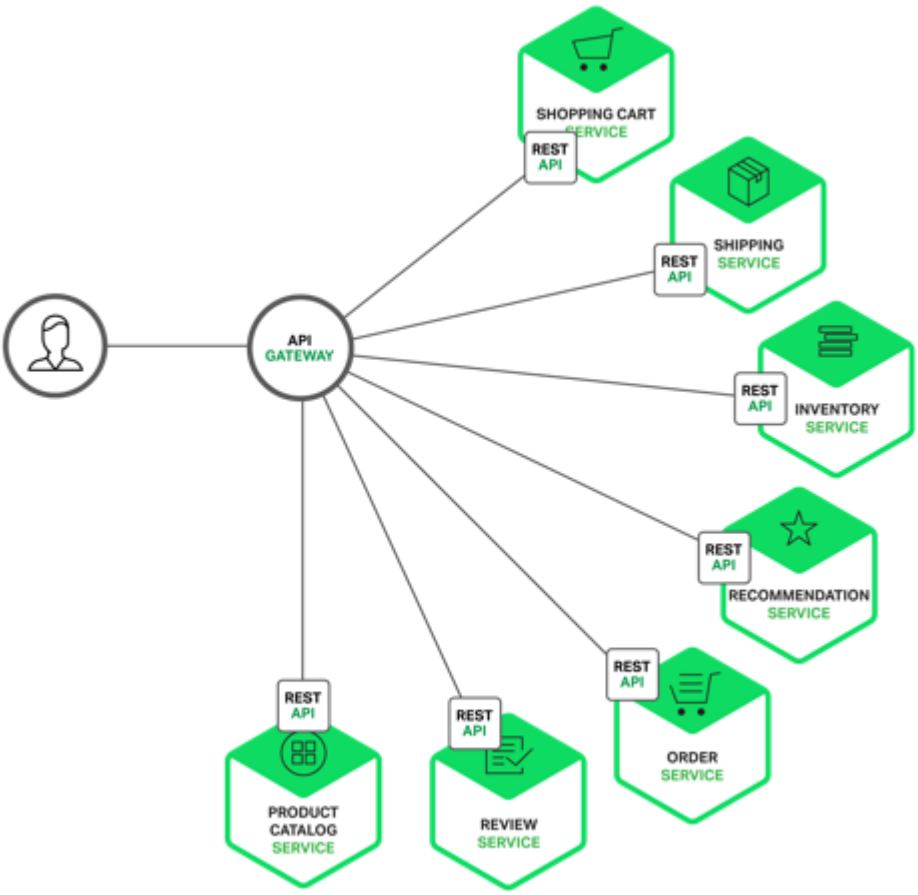
- L'impossibilité d'ajouter des commentaires peuvent nuire à la compréhension d'un grand flux de données tels que les configurations et les métriques complexes
- Il n'est toujours compact dans tous les cas
- Ne disposant pas d'un système permettant de valider la structure, les fichiers massifs peuvent contenir des erreurs (nom mal orthographié, attribut manquant mais nécessaire, etc.)
- Il n'est pas adapté à la rédaction de documents
- Il ne dispose pas d'un système de transformation de données afin de les exporter sous différents formats

Tous ces éléments sont à prendre en compte par rapport aux types de projets que l'on souhaite implémenter, le JSON répond à une stratégie de performances au niveau des ressources machines et de rapidité d'implémentation plus qu'à des traitements plus sécuritaires ainsi qu'à un modèle de donnés complexe. Sachant que dans le cadre des microservices où on souhaite réduire la base de code, la limitation du modèle de données et être dans une stratégie de communication multi-plateforme - multi-technologie, ce langage sied bien à l'architecture.

2.3.3. L'API Gateway

Lorsqu'on implémente une architecture de microservices, il est nécessaire de décider comment les applications clients (mobile, web) généralement appelées Frontend vont interagir avec les microservices (Backend).

Dans une application monolithique, il est d'usage d'avoir et d'entretenir un ensemble de plusieurs points d'entrée (endpoints), qui seront répliqués via un load-balanceur qui sera utilisé pour distribuer la charge entre ces derniers. Cependant dans une architecture de microservices, chaque microservices expose via une API un ensemble d'endpoints généralement fins. De ce fait pour les communications entre Frontend et Backend, une approche consiste à utiliser une passerelle d'API (API Gateway) pour faire le lien entre les 2 systèmes de l'application.



*Source : « **Building microservices using an api gateway** » nginx.com*

Dans cet exemple, NGINX permet d'observer qu'une API Gateway est un serveur qui constitue le point d'entrée unique dans un système, généralement interne comme le Backend. Elle encapsule le système interne et fournit une API adaptée à chaque client.

Le rôle majeur d'une API Gateway est d'intercepter toutes les requêtes en provenance des systèmes externes et de gérer le routage de ces derniers vers les microservices appropriés. Elle traite souvent une demande en appelant plusieurs microservices afin d'agréger les résultats. Elle peut faire la traduction entre des protocoles web comme le HTTP, les websockets et les protocoles utilisés à l'interne (AMQP)

Une API Gateway pourrait également avoir d'autres responsabilités telles que :

- L'authentification
- La surveillance de l'équilibrage de la charge
- La mise en cache

- Le management des requêtes
- Le traitement des réponses statiques

Il est tout à fait possible de mettre en place une solution d'API Gateway maison, cependant il existe plusieurs solutions sur le marché qui ont permis de résoudre un ensemble de problème de conception qu'on devrait prendre en compte dans le cadre de l'intégration des microservices :

La performance et la scalabilité

Pour la plupart des applications la performance et la scalabilité de l'API Gateway est cruciale. Il est donc important de construire la passerelle sur une plateforme prenant en charge les **E/S asynchrones et non bloquantes**. Sur la JVM, on pourrait implémenter des frameworks tel que **Netty**, **Vertx** ou encore **Spring Reactor**. Notez que la solution non-JVM la plus populaire est **Node.js** construite sur le moteur javascript de Chrome.

L'utilisation d'un modèle de programmation réactive

Une API Gateway gère certaines requêtes en les acheminant simplement vers le service backend approprié. Cependant le traitement de certaines requêtes nécessite l'appel et l'agrégation des réponses en provenance de plusieurs services. Afin de minimiser le temps de réponse, l'API Gateway doit effectuer les requêtes de façon indépendantes simultanément. Parfois il peut exister des dépendances entre les APIs, il faut donc savoir orchestrer les appels.

La création de l'algorithme de composition des appels en utilisant les callbacks asynchrones traditionnels conduisent rapidement à un code enchevêtré, difficile à maintenir et sujet aux erreurs. Une meilleure approche consiste à écrire le code dans un style déclaratif en utilisant une approche réactive. L'approche réactive permet d'écrire le code de l'API Gateway de façon simple mais efficace. On peut penser à des solutions telles **ReactiveX** pour le .NET, **RxJava** pour la JVM, **RxJS** pour le Javascript.

L'invocation de service

Une application basée sur les microservices est un système distribué et doit utiliser un mécanisme de communication interservices. Comme nous le verrons plus en détail dans le point 2.3.4, il existe deux modes de communications pouvant être implémenter entre les services. Les communications asynchrones basées sur la messagerie, qui implémentent des messages-broker tel que **JMS** ou **AMQP**.

Les communications synchrones tel que HTTP ou **Thrift**. Généralement, un système utilisera les 2 styles, par conséquent, l'API Gateway devra prendre en charge les deux mécanismes de communication.

Le service Discovery

L'API Gateway doit connaître l'emplacement (adresse IP et port) de chaque microservice avec lequel il communique. Dans une application traditionnelle, on pourrait probablement se connecter directement aux emplacements, mais dans une application moderne de microservices basée sur le cloud, trouver les emplacements est un problème non trivial.

Certes, les services d'infrastructure, tels qu'un bus de messages, auront généralement un emplacement statique, qui peut être spécifié via des variables d'environnements du système d'exploitation. Cependant, déterminer l'emplacement d'un service d'application n'est pas si facile. En effet, ces derniers ont des emplacements attribués de manière dynamique. En outre, l'ensemble des instances d'un service change dynamiquement en raison de la mise à l'échelle automatique. Par conséquent, l'API Gateway, comme tout autre client de service du système, doit utiliser le mécanisme de service Discovery (côté serveur ou côté client). Le point 2.3.6 décrit plus en détail la notion de service Discovery. Pour le moment, il est intéressant de noter que si le système utilise le Discovery côté client, la passerelle API doit pouvoir interroger le **service de registre**, qui est une base de données de toutes les instances de chaque microservice et de leurs emplacements.

Le traitement des échecs partiels

Un échec partiel est un problème qui se pose dans tous les systèmes distribués lorsqu'un service appelle un autre et que ce dernier répond lentement ou est tout simplement indisponible. L'API Gateway ne devrait jamais rester bloquée en attente indéfiniment. Toutefois le traitement d'un échec dépend de la nature de ce dernier.

S'ils existent des ressources plus ou moins statique, l'API Gateway devrait être en mesure de les récupérer depuis son cache (ou un cache externe comme **Redis**), si le service chargé de les envoyer est indisponible.

Ce type de scénario, s'ils sont identifiés et gérés par l'API Gateway, alors cela pourrait garantir que les défaillances du système auront un impact minimal sur l'expérience utilisateur. La bibliothèque **Hystrix** (de Netflix) peut s'avérer être une solution particulièrement intéressante pour écrire du code permettant de gérer un cas d'échec partiel sur un service indisponible :

- On écrit un code effectuant des appels distants
- Hystrix met un time-out sur les appels de services
- Si le seuil est dépassé, Hystrix implémente le pattern de circuit-breaker qui empêche le client d'attendre indéfiniment et définit une action de secours en cas d'échec de la requête vers un service donné.
- Tout appel vers le service aura pour réponse l'action de secours d'Hystrix pendant une période bien définit (qui devrait correspondre au temps qu'il faudrait au service pour être à nouveau opérationnel)

Avantages et inconvénients

L'utilisation d'une API Gateway présente à la fois des avantages et des inconvénients.

Un important avantage à utiliser une API Gateway est qu'elle encapsule la structure interne de l'application. Plutôt que de devoir faire appel à des services spécifiques, les clients communiquent uniquement avec l'API Gateway. Celle-ci fournit à chaque type de client une API spécifique. Cela réduit le nombre d'allers-retours entre le client et l'application et simplifie également le code coté client.

L'API Gateway présente également certains inconvénients. C'est encore un autre composant hautement disponible qui doit être développé, déployé et géré. Il existe également un risque qu'elle devienne un goulot d'étranglement pour le développement. Les développeurs doivent mettre à jour L'API Gateway afin d'exposer les endpoints des microservices implémentés. Il est important que le processus de mise à jour soit le plus léger possible.

Malgré ces inconvénients, cependant, pour la plupart des applications actuelles, utiliser une API Gateway dans une architecture de microservices fait partie des bonnes pratiques.

2.3.4. Le synchronisme vs l'asynchronisme

L'une des décisions les plus importante à prendre avant de faire des choix technologiques bien spécifiques est de savoir si la communication entre les services doit être synchrone ou asynchrone.

En effet ce choix fondamental peut être structurant pour la mise en œuvre de l'architecture, car la mise en place de l'un et/ou l'autre des deux modes de communication permettent deux styles de collaboration idiomatiques différents : la requête/réponse et les évènements.

Avant de sélectionner un mécanisme de communication, il serait intéressant d'analyser la question en 2 dimensions. Premièrement :

- **One-to-one** : Chaque requête client est opérée par une unique instance de service
- **One-to-many** : Chaque requête est opérée par plusieurs instances de service

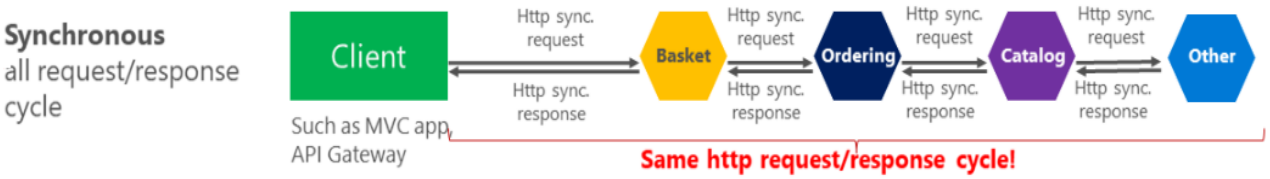
Ensuite, au niveau de l'interaction, déterminer si elle est asynchrone ou synchrone, après quoi on peut déduire quel type d'implémentation on devrait mettre en place :

	ONE-TO-ONE	ONE-TO-MANY
SYNCHRONE	Requête/Réponse	-
ASYNCHRONE	Notification	Publish/subscribe
	Request/ async response	Publish/ async responses

Maintenant explorons plus concrètement les 2 aspects.

Appels synchrones et collaboration requête/réponse

Les appels synchrones correspondent à une **collaboration *request/response*.** Ceci implique que le client doit s'adapter au temps de traitement du microservice et doit en tenir compte dans son implémentation. L'importance de la prise en compte du temps de traitement du service permet par exemple d'anticiper des timeouts dans le cas où le temps de réponse serait trop long.



Source : https://docs.microsoft.com/fr-fr/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

« Communication dans une architecture de microservices », cas d'une communication synchrone.

Dans ce cas de figure, quand le client décide d'effectuer un appel HTTP, un ensemble de services traite la requête et met en attente le client jusqu'à obtention de la réponse. Le temps de réponse est donc dépendant du nombre de services désignés dans le traitement de la requête.

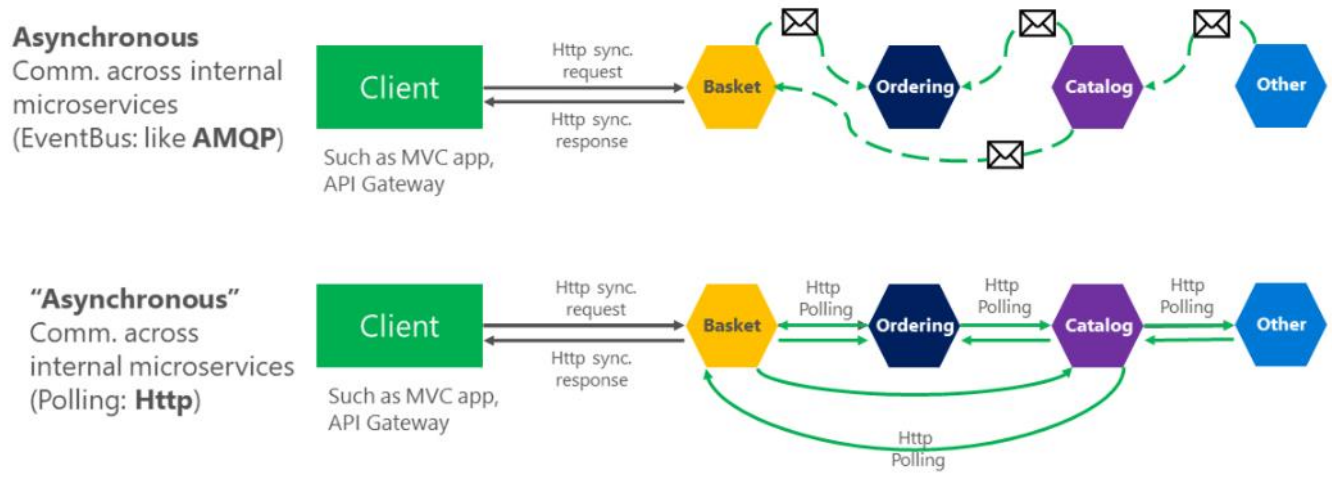
Le choix technologique pour gérer ce scénario devrait porter sur l'implémentation du **REST** et pour aller plus loin, il serait intéressant de considérer une alternative telle qu'**Apache Thrift** qui est un framework multi-langage permettant de faire du RPC. Ceci permet de réduire de manière significative la surcharge de programmation manuelle, et fournir des mécanismes efficaces de sérialisation et de transport à travers toutes sortes de langages de programmation et de plates-formes.

L'avantage d'une communication synchrone, est la certitude d'avoir une réponse sur le statut de la requête. Mais le fait qu'il faille qu'un émetteur doive s'adapter au temps de traitement de la requête est un type de couplage qui risque de se fortifier sur le long terme.

Appels asynchrones et collaboration basée sur les évènements

Lors d'une communication asynchrone, l'appelant n'attend pas que l'opération soit terminée avant de poursuivre son processus. Dans ce cas de figure, un client envoie une requête au microservice mais n'attend pas pendant le traitement, il peut être notifié à la fin du traitement directement par ce service ou s'il est abonné à des événements déclenchés par le service de façon à recevoir les notifications.

Les appels asynchrones correspondent à une **collaboration basée sur les évènements**, car le client s'abonne à des évènements émis par le service en fonction de quoi il souhaite être notifié. Le service n'a pas de connaissance des clients qui sont abonnés à ses évènements. Contrairement à une communication synchrone, il n'est pas nécessaire que le client s'adapte en fonction du traitement de la requête et doit être écrit en conséquence.



Source : https://docs.microsoft.com/fr-fr/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

« Communication dans une architecture de microservices », cas de communications asynchrones.

Les systèmes de messageries supportent des protocoles standard tel que AMQP et STOMP. Il existe sur le marché un large éventail de solution open-source permettant d'implémenter ce mode de communication, tel que RabbitMQ, Apache Kafka, Apache ActiveMQ pour ne citer que ceux-là.

L'avantage d'une communication asynchrone vient du fait qu'il n'est pas nécessaire pour un client de s'adapter en fonction des traitements des requêtes, cela peut s'avérer utile pour les travaux de longue durée, où il est peu pratique de maintenir une connexion ouverte pendant une longue période entre le client et serveur. De plus, ce mode peut s'avérer particulièrement judicieux face à des problèmes de latence, en effet, les blocages d'appels en attente de résultats peuvent significativement ralentir le trafic.

En conclusion : une communication synchrone semble facile à raisonner, car les informations sur la fin des traitements ainsi que les statuts de succès ou d'échec sont connus. Mais en raison de la nature des réseaux et des appareils mobiles, le fait de lancer des demandes et de supposer que le traitement de ces dernières est un succès permet de garantir la réactivité de l'interface utilisateur, même si le réseau est très lent, c'est ce que permet une communication asynchrone, bien qu'elle soit technologiquement plus complexe à implémenter.

Les communications basées sur les évènements qui sont de nature asynchrone, sont privilégiées dans les architectures de microservices car elles permettent de respecter les principes de découplage et de tolérance aux échecs partiels prônés par l'architecture. Le choix technologique afin de gérer ce cas de figure pourrait se porter sur la mise en place d'un répartiteur de message léger (qui ne se limite qu'à la répartition des messages).

Une règle importante à garder à l'esprit consiste à utiliser la communication asynchrone entres les services internes et à utiliser uniquement la communication synchrone au premier niveau des microservices au plus près des applications Frontend.

2.3.5. L'orchestration vs chorégraphie

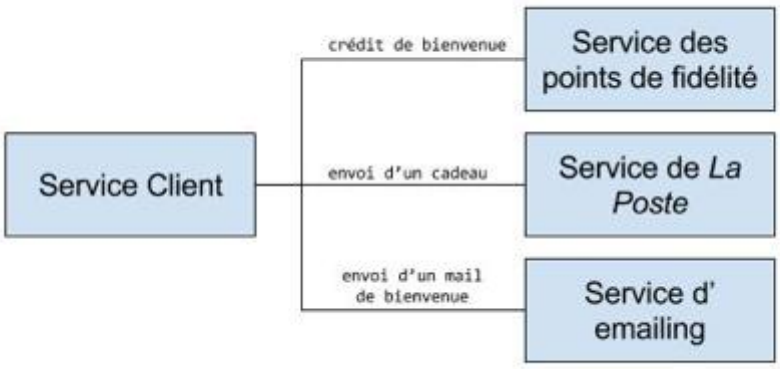
Faisant suite au point précédent qui a permis une bonne compréhension entre les appels synchrones, les appels asynchrones et les avantages à employer l'une ou l'autre des pratiques dans le cadre des microservices, il est important de comprendre comment au sein de cette architecture, il est possible d'enchainer les appels. En effet, certain processus, nécessite plusieurs enchainements d'appels de services pour être implémentés. Voici ce qu'il est possible de faire :

Appels en orchestration :

Pour effectuer des appels en orchestration dans une architecture de microservices, il faut mettre en place un service qui va implémenter la logique du workflow. Ce service appelant que l'on pourrait appeler le service Master, sait exactement quels sont les services à appeler et l'ordre d'appel. Le fait que le service master soit considéré comme un chef d'orchestre permet de distinguer la notion d'orchestration des appels de services.

L'ordonnancement entre les microservices est plus facile à implémenter et se fait directement dans le service master. Cette disposition permet d'effectuer des appels synchrones aux services, ce qui fait que le workflow peut être stoppé si l'un des appels est un échec.

En plus des inconvénients remontés comme on l'a vu avec des appels synchrones classiques, l'inconvénient majeur de cette approche est que la connaissance des services à appeler par le master augmente le couplage entre tous ces services, car si le contrat d'un des services appelés venait à être modifié, cela impacterait automatiquement le master.



*Source : « **Building microservices** », Samuel Newman.*

Le service client qui est le service Master est responsable des enchainements des appels vers les autres services qui lui permettrai de construire sa propre implémentation. Il s'agit essentiellement d'appels synchrones et les services appelés sont connus.

Appels en chorégraphie :

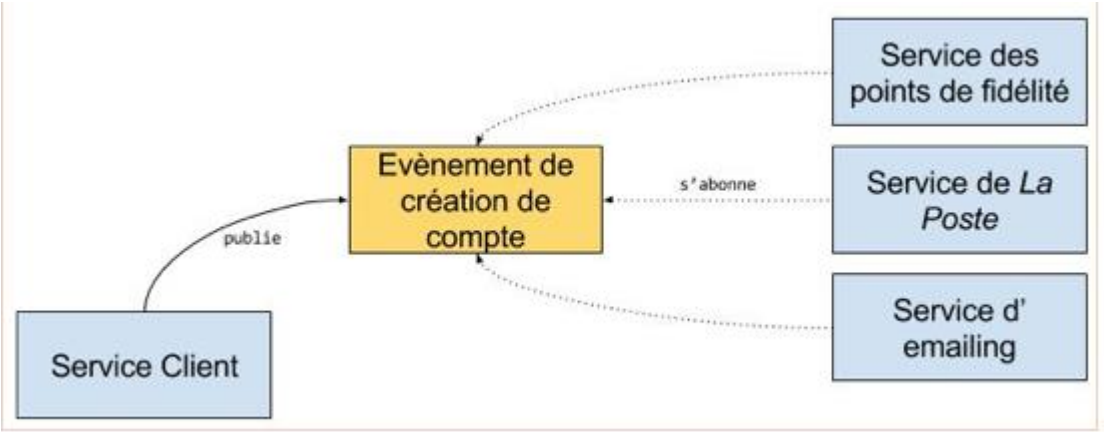
Dans le cadre d'appels en chorégraphie, il est aussi nécessaire d'avoir un service master dans lequel décrire les enchainements. Mais dans ce cas de figure, le master n'a pas la connaissance des services qui dépendent de lui. En effet, les services souscrivent auprès du master pour être notifié qu'un

évènement spécifique est survenu, à la suite de quoi ces derniers effectuent des traitements. Chaque service renvoi un résultat au master.

La notion de chorégraphie vient du fait que les services sont abonnés au master et décident eux-mêmes d'effectuer le traitement lors qu'on leur donne le signal via un évènement.

Des communications basées sur les évènements étant de nature asynchrone, on retrouve ici, la même complexité d'implémentation, à laquelle il faut ajouter la difficulté à interrompre un workflow en cas d'erreur et l'ordonnancement des services en fonctions des évènements déclenchés.

L'intérêt de ce mécanisme est avant tout le découplage, le service master ne connait pas les services appelés et ignore l'ordre dans lequel ces derniers effectuent leur traitement. Il se contente juste de déclencher les évènements et les services savent eux même s'ils doivent s'exécuter ou non.



*Source : « **Building microservices** », Samuel Newman.*

Le service client est le service master, il publie un ensemble d'évènement dont il a besoin pour implémenter son processus, et les services abonnés vont s'exécuter et renvoyer leur résultat. Il s'agit essentiellement d'appels asynchrones et les services intervenant ne sont pas connus du master.

En Conclusion : Un enchainement d'appels en orchestration semble facile à raisonner et est peut-être plus simple à mettre en place, cependant ils entrainent un couplage fort dans l'architecture. Il vaut mieux dans ce cadre des architectures de microservices, privilégier le plus possibles des enchainements d'appels en chorégraphie, pour rester dans un modèle asynchrone et garantir la disponibilité des applications.

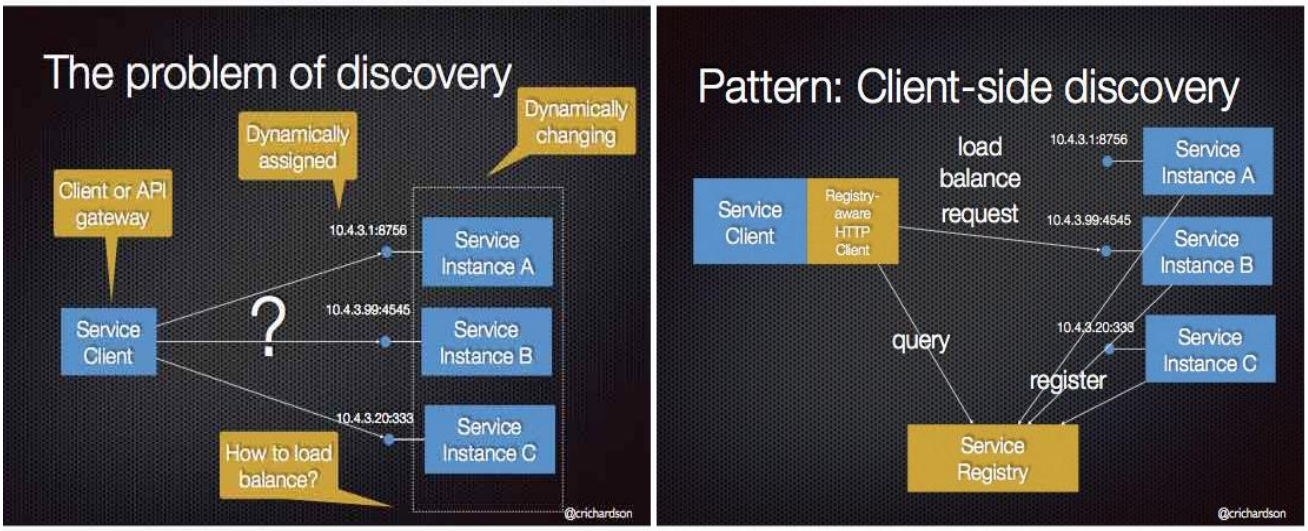
2.3.6. Le service Discovery

Dans le cadre de l'intégration, imaginons que nous ayons du code qui appelle une instance de service dotée d'une API REST. Afin d'envoyer une requête, ce code doit connaitre l'emplacement réseau de cette instance, c'est-à-dire son adresse IP et son port. Dans une application traditionnelle fonctionnant sur du matériel physique, les emplacements réseau sont relativement statiques. Par exemple, les emplacements réseaux sont consignés dans un fichier de configuration qui est mis à jour de temps de temps. Cependant, dans une application moderne de microservices basée sur le cloud il s'agit d'un problème difficile à résoudre car les instances de services ont des emplacements réseau attribués de manière dynamique. De plus, ces emplacements changent en raison de la mise à l'échelle automatique et des échecs. Par conséquent il est d'usage d'utiliser un service élaboré appelé le service Discovery qui est capable de « découvrir » les adresses des instances de services avec lesquelles on souhaite communiquer dans une architecture de microservices hautement distribuée.

Il existe 2 principaux patterns du service Discovery, celui coté client et celui coté serveur. Commençons tout d'abord par le premier

Le pattern Discovery coté client

Quand ce modèle est utilisé, c'est le client (service appelant) qui est responsable de la détermination des emplacements réseaux des instances disponibles et qui effectue les requêtes d'équilibrage de charge entre ces instances. Le client interroge un registre de service, qui est une base de données regroupant les instances disponibles. L'emplacement d'une instance est automatiquement enregistré dans le registre des services, mis à jour périodiquement avec un mécanisme de pulsation (une requête GET/healthcheck vers l'instance dont le statut HTTP(OK/KO) permet maintenir l'instance dans le registre) et supprimée quand elle se termine. Ce client devra par la suite, utiliser un algorithme d'équilibrage de charge pour sélectionner l'une des instances.



Source : https://microservices.io/patterns/client-side-discovery.html

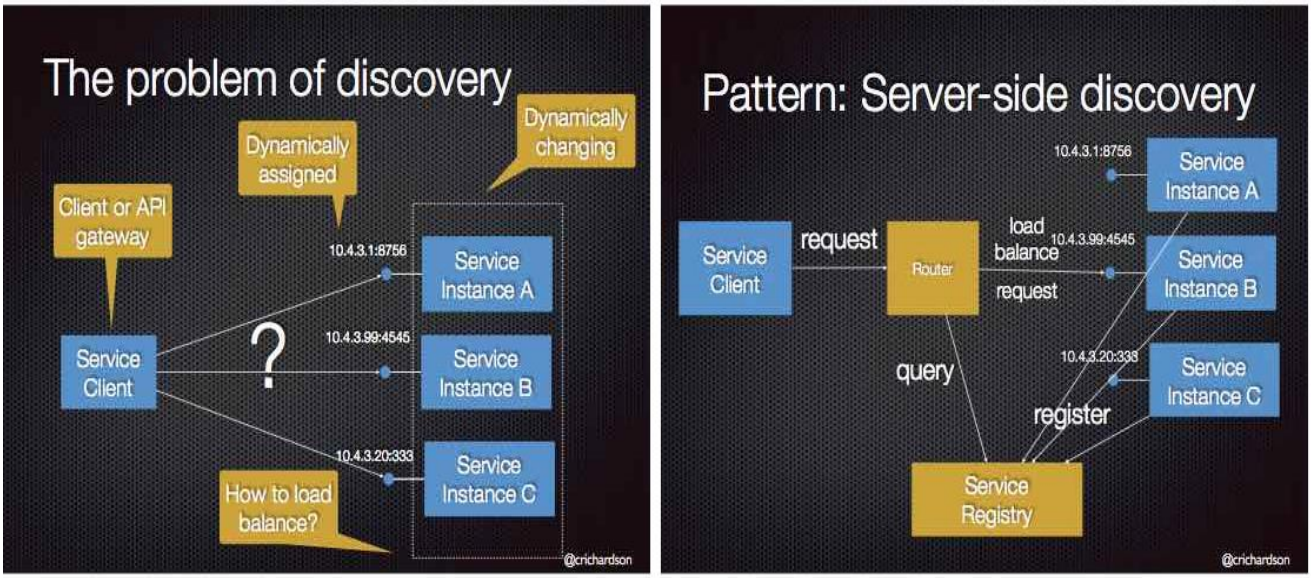
Démonstration de ce que le pattern « Client-side discovery » mis en situation, apporte comme solution.

Ce modèle présente quelques avantages et inconvénients. Il est relativement simple à implémenter, mis à part le registre de services à mettre en place, les mécanismes restants sont faits automatiquement. De plus, la connaissance des instances de services disponibles par le client lui permet de prendre des décisions intelligentes d'équilibrage de charges en fonction des besoins de l'application. Un inconvénient majeur de ce modèle est qu'il couple le client au registre de services et qu'il faille implémenter cette logique de service discovery pour chaque langage de programmation d'un client.

Netflix OSS est un bon exemple de modèle de découverte coté client, il propose des services tout prêt (on parle de Edge microservices) tels que Netflix Eureka qui est un registre de services qui fournit une API REST permettant de gérer l'enregistrement des instances et d'interroger les instances tel que défini ci-dessus. Netflix Ribbon qui est un client qui fonctionne avec Eureka pour repartir les requêtes sur les instances de service disponibles

Le pattern Discovery coté-serveur

Lors de l'utilisation de ce modèle, le client effectue une requête à une instance de service via un load-balanceur. Ce dernier interroge le registre de services et par la suite route la requête vers l'instance de service disponible. Comme avec le pattern Discovery coté client, les instances de services sont enregistrées et désenregistrées dans le registre de service



Source : https://microservices.io/patterns/server-side-discovery.html

Démonstration de ce que le pattern « Server-side discovery » mis en situation, apporte comme solution.

L'un des plus grands avantages de ce modèle est que les détails de découverte des instances sont abstraits pour un client. Les clients devront simplement interroger l'équilibreur de charge. Ceci élimine le besoin d'implémenter la logique de Discovery pour chaque langage de programmation et framework utilisé par les clients de service. Aussi, bien que nous soyons dans la section liée à l'intégration, il est intéressant de dire que certains environnements de déploiement fournissent cette fonctionnalité. De ce fait, cette information doit être prise en compte dans le choix technologique de la solution de déploiement.

Cependant, cette solution présente un léger inconvénient, le load-balanceur compte parmi les composants du système qui devra être hautement disponible et qu'il va falloir gérer.

Une solution intéressante à explorer serait un serveur HTTP et load-balanceur telle que NGINX afin d'implémenter le pattern Discovery coté serveur.

Les Edge microservices

Il s'agit de microservices génériques que l'on retrouve sur le marché qui assurent le fonctionnement et la cohésion globale du système. On pourrait citer entre autres :

- Hystrix : rend une application résiliente aux pannes de ses dépendances externes en arrêtant momentanément de les invoquer le temps de leur disponibilité.
- Ribbon : Permet les appels de procédure distante avec des équilibreurs de charge logicielle intégrés.
- Zuul : Joue un rôle de reverse proxy, c'est un routeur qui va dispatcher les utilisateurs sur les différents services en fonction des URLs appelés.
- Archaius : Propose un ensemble fonctionnalités pour la gestion des configurations des APIs.
- Zipkin : Système de traçage distribué qui aide à rassembler les données de synchronisation nécessaires pour résoudre les problèmes de latence dans les architectures de microservices.
- Zookeeper : Logiciel de gestion de configuration pour système distribué qui supporte une haute disponibilité grâce à des services redondants
- Redis : Système de gestion de base de données clef-valeur scalable. Il est adapté aux bases de données non-relationnelle de type NoSQL et vise à fournir les performances les plus élevées possible.
- Feign : Client de service web déclaratif qui facilite l'écriture de clients de services Web.

3. API management

A ce stade de l'étude, nous avons observé comment les microservices sont élaborés et regroupés dans plusieurs APIs, comment on les concevait et comment ils s'intègrent entre eux au sein d'une application. L'idée d'introduire à présent la notion de gestion d'API ou encore API management revient à voir un aspect lié à la gouvernance des microservices via leur API. En effet les APIs sont conçues dans la stratégie d'une entreprise afin de permettre à cette dernière de lui générer du profit.

En effet, l'API management est un processus de publication et de supervision des APIs dans un environnement sécurisé et évolutif, dont l'objectif est de permettre aux entreprises qui les publient ou les utilisent, d'en surveiller le cycle de vie et de s'assurer que les besoins des développeurs et des applications utilisant les APIs soient satisfaits.

Les besoins en termes d'API management varient d'une organisation à l'autre, mais la notion d'API management en elle-même englobe certaines fonctionnalités de base, notamment en ce qui concerne:

- L'exposition et la documentation des services
- Gestion des authentification et habilitations
- La sécurité globale
- La supervision des services exposés au travers de métriques prédéfinies
- La gestion du trafic
- Le contrôle de version
- La transformation des données

Mais bien au-delà de l'aspect purement métier d'une organisation, l'API elle-même devient un véritable enjeu business. En effet, beaucoup d'entreprises souhaitent exposer leurs ressources et services sous forme d'API afin d'en monétiser les accès et/ou les commercialiser entièrement. Les organisations, quel que soit leur taille, cherche par ce biais un différenciateur concurrentiel très fort qui a mené à ce qu'on appelle aujourd'hui l'**API Economy**. En 2015, l'étude Tech Trends publiée par le cabinet Deloitte identifie l'API Economy comme une tendance majeure qui couvre les secteurs suivants :

- Les réseaux sociaux
- Le e-commerce via le Big Data
- Les télécommunications
- Les objets connectés (IoT)
- Les services de géolocalisation
- Les Apis bancaires

C’est notamment pour prendre en compte cet aspect business que l’API management est devenue de plus en plus importante. Il y a une dépendance croissante des entreprises vis-à-vis des APIs, une augmentation considérable du nombre d'API dont elles dépendent et de la complexité de ces dernières. Les exigences et processus de création et de gestion des APIs diffèrent de la plupart des autres applications. Pour être utilisées correctement, les APIs nécessitent une documentation solide, des niveaux de sécurité accrus, des tests complets, une gestion des versions et une fiabilité élevée. Étant donné que ces exigences vont souvent au-delà de la portée des projets logiciels, les organisations sont ouvertes à l’idée d’utiliser des solutions d’API management.

3.1. Les API publiques et privées

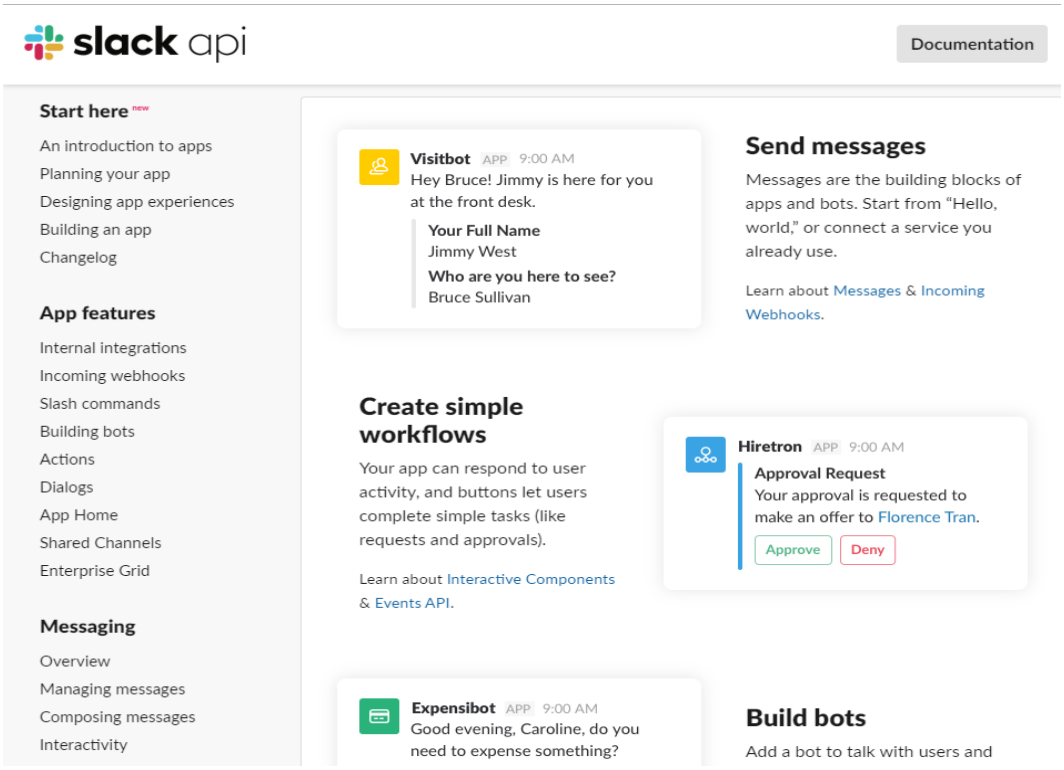
Dans la mise en place d’une brique tel qu’un outil d’API management à la fondation de son architecture de microservices, une organisation doit en premier lieu définir ses enjeux stratégiques et économiques afin de décider si elle doit ou non ouvrir son API. Voyons maintenant quels sont les utilités d’ouvrir ou non un système au grand public ou à des partenaires au travers des définitions d’API publiques et privées.

Les APIs publiques

Comme le nom l’indique les API publiques sont ouvertes à tous. Elles sont conçues pour être accessible par une communauté de développeurs plus large (d’un point de vue mondial) pour la création d’applications mobiles et Web. Elles peuvent être observées comme des API utilitaires pour implémenter une solution interne en permettant aux organisations d’ajouter de la valeur à leur cœur de métier sans fournir un certain effort de réalisation. Les API publiques permettent essentiellement aux développeurs d’utiliser leur créativité et aident à réduire des couts en termes de temps de développement. Elles peuvent aussi aider à générer de nouvelles idées commerciales.

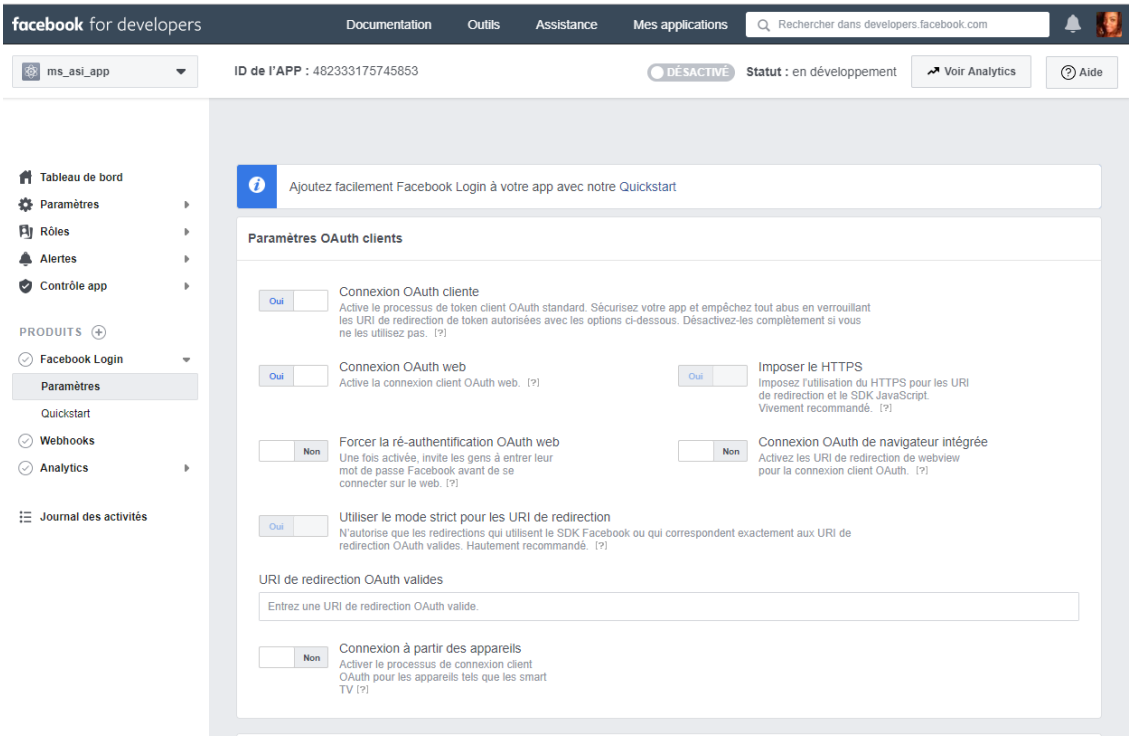
Lorsqu’une entreprise décide d’ouvrir ses APIs, cela est fait dans le but d’accéder et de capter un marché d’utilisateurs.

Par exemple : en ouvrant ses APIs, une entreprise comme **Slack** (logiciel de messagerie) a réussi à créer un cercle vertueux en ayant une base large d’utilisateurs qui a attiré des développeurs qui ont intégré des services tiers. Ces services ont par la suite attiré d’avantage d’utilisateurs.



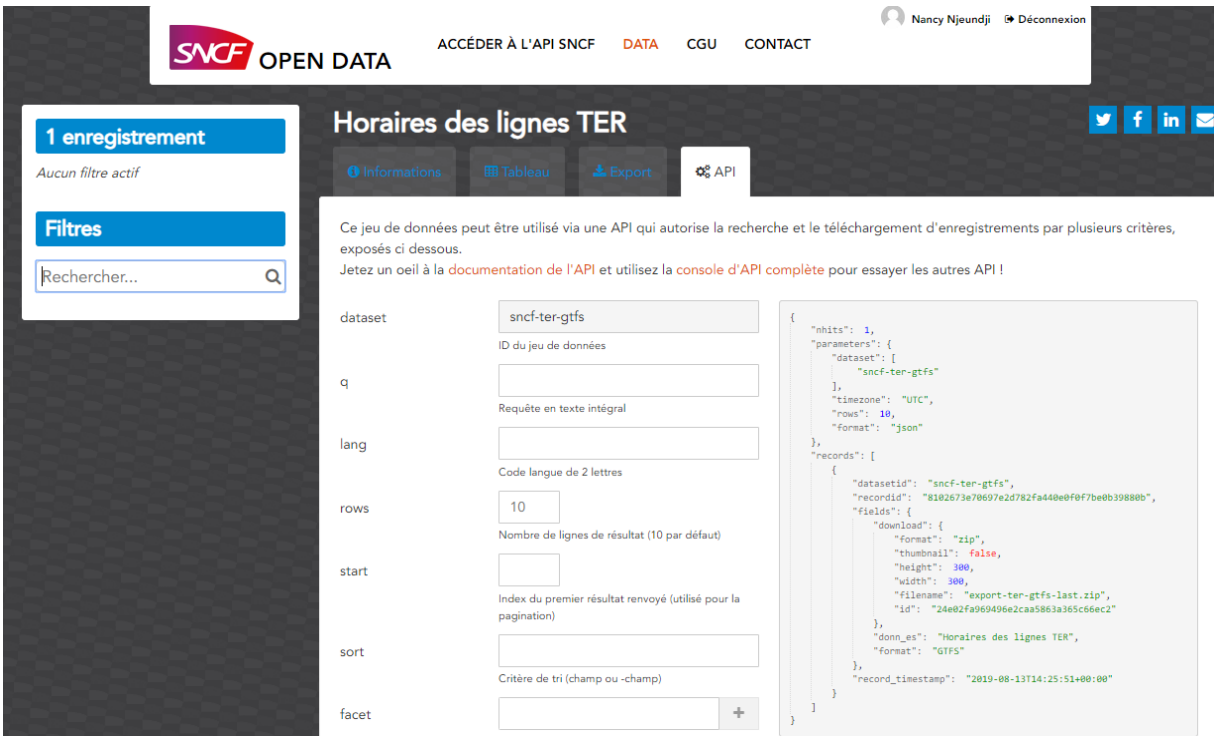
Capture : Exploration de l'API de Slack

Nous pouvons aussi citer le **Facebook Connect** qui permet à des millions d'utilisateurs de se connecter à des applications tierces en quelques clics. En ouvrant cette API, Facebook s'est rapidement imposé comme une brique technologique pour un écosystème technique.



Capture : Exploration de le l'API de Facebook

Le groupe **SNCF** est aussi un acteur majeur de l'open Data en France, grâce à la collecte et au partage de nombreuse données remontées par le transport quotidien de 10 millions de voyageurs. L'Open Data et les API proposés par la SNCF constituent un vecteur d'innovation sur les nouveaux challenges liés à la mobilité (cheminement, optimisation et valorisation du temps de voyage, adaptations aux besoins des voyageurs, etc.)



Capture : Exploration de l'API du groupe SNCF

Partant toujours du principe que les utilisateurs cible des API sont ceux qui l'exploitent, c'est-à-dire les développeurs, le succès d'une API publique dépend de sa capacité à attirer ces derniers et à les aider à créer de véritables applications.

Les APIs privées

Il s'agit essentiellement d'API hébergées sur un réseau privé et qui sont exposées et utilisées à l'interne dans une organisation. Généralement, elles sont sécurisées avec des restrictions d'accès et des limites d'utilisation par un groupe de développeurs et de partenaires connus. Les ressources ne sont par conséquent pas accessibles, ni aux usagers, ni au grand public

Ce type d'API, permettent à des entreprises de centraliser les données nécessaires à leur bon fonctionnement. Elles conservent la main sur leurs données et informations partagées et cela pourrait avoir un impact positif sur la productivité de celles-ci.

Elles sont généralement destinées aux intégrations de partenaires B2B et à un usage interne. Ainsi on peut scinder des API privées en deux groupes :

- API partenaires : elles contribuent à une stratégie commerciale et économique pour fournir des services à des partenaires de business
- API internes : Elles contribuent à une stratégie de croissance et de gouvernance pour développer le cœur de métier d'une organisation et son savoir-faire. (Applications web ou mobiles utilisées strictement en interne)

L'intérêt d'utiliser des API fermées se porte sur la nature des données échangées, à l'exemple de tout type de données sensibles telles que des données personnelles, bancaires, des secrets industriels…Avoir un environnement clos pour ce type de données permet d'identifier et de traiter en amont des problèmes potentiels de sécurité.

Attention : Une pratique consiste à héberger des APIs sur un réseau public sans toutefois les exposer, c'est-à-dire sans partager leur existence et leur documentation. Ces APIs sont théoriquement privées parce qu'elles ne sont pas connues de la communauté des développeurs. Mais techniquement, les développeurs, dans leur phase d'expérimentation, de recherche et de tests peuvent les trouver et les utiliser accidentellement. Ce n'est donc pas la bonne approche !

Le site www.programmableweb.com répertorie l'ensemble des APIs du globe. On peut avoir un aperçue rapide de l'utilité de certaines APIs en fonction d'un besoin et d'un métier particulier :

Organisation	Adresse	Catégorie	Synopsis
Facebook	https://developers.facebook.com	• Social • Marketing • Publicité • Paiement • Réalité augmentée	Pour publier des activités sur les pages d'actualisation et de profil de Facebook, sous réserve des paramètres de confidentialité de chaque utilisateur

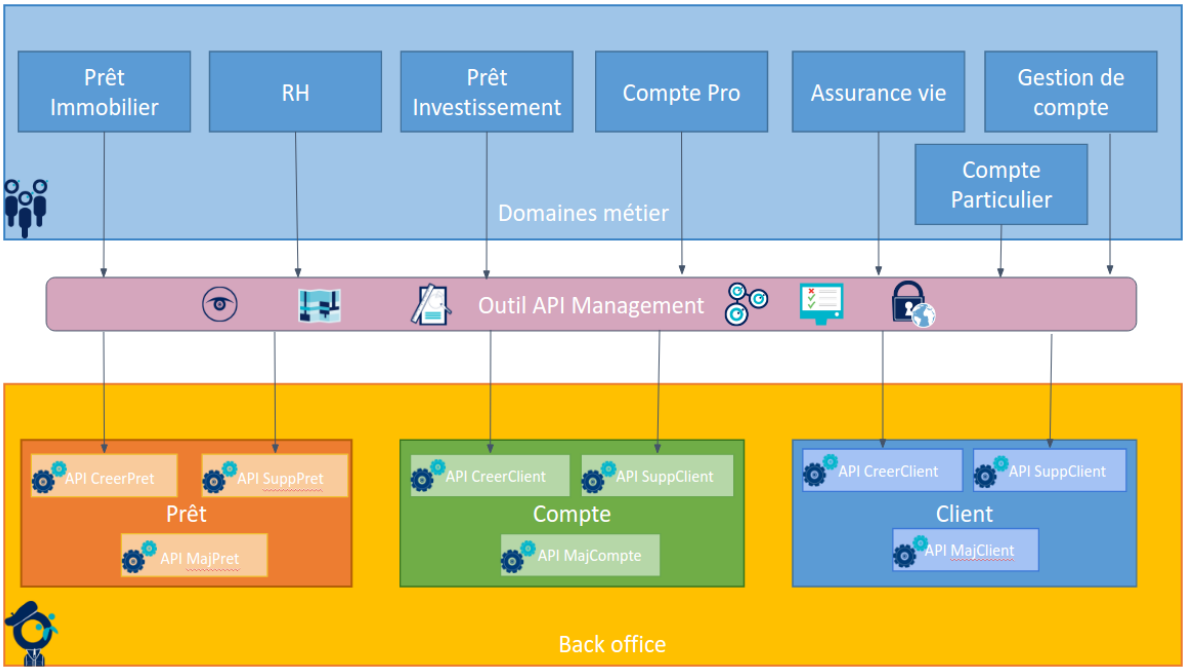
Google	https://developers-google-com.res.banq.qc.ca	• Cloud • Cartographie • Outils • eCommerce • Entreprise • Office • Photographie • Stockage de donnée • Etc.	Pour la communication avec les services Google, tels que la recherche, la traduction, Gmail, les cartes géographiques, les vidéos, les réseaux sociaux et la publicité
Instagram	https://www.instagram.com/developer/	-photographie - social	Pour obtenir des photos à partir d'Instagram et de les afficher sur votre propre site Web ou application
Amazon	https://developer.amazon.com/	• Base de données • Email • Paas • Gestion de la relation client • eCommerce	Pour les achats intégrés, les annonces mobiles, les notifications push, l'expérience utilisateur très poussée
AT & T	https://developer.att.com/	• Messagerie • Relation client • Urgences • Sécurité • Management	pour créer des applications pouvant envoyer des messages (SMS / MMS), localiser des utilisateurs, convertir du texte en parole et de la parole à texte, monétiser des applications via des publicités incorporées, utiliser les fonctionnalités M2X, etc.

Une fois les API crées, elles peuvent être gérées à l'aide d'une plateforme de gestion d'API management. Cette dernière aide les organisations à publier des API (à l'interne et à l'externe), afin de fournir des services liés à leur expertise.

Pour les développeurs, l'API management permet d'analyser, sécuriser et documenter les APIs. En ce qui concerne les entreprises, elle permet d'accélérer leur déploiement sur les canaux numériques, à monétiser le savoir-faire, à optimiser les investissements de transformation numériques et améliorer la collaboration avec de nouveaux partenaires.

3.2. Les services principaux

Dans une architecture cible d'une application, un outil d'API management est positionné stratégiquement de telle sorte qu'il puisse permettre l'intégration entre les systèmes externes (Frontend et systèmes tiers) et internes (Backend). Ainsi il pourra réaliser les taches aidant à l'interception, au routage et au contrôle des données.



Source : blog.octo.com

Mise en situation d'un outil d'Api management au sein d'une architecture

L'API Gateway

La notion de passerelle d'API ou encore API Gateway constitue le cœur de toute solution d'API management, car elle permet une communication flexible, sécurisée et fiable entre les services principaux et les applications numériques. Elle permet d'exposer, de sécuriser et de gérer les données et services principaux en tant qu'API RESTful.

Tel un proxy très poussé, elle crée une façade en amont des services Backend qui intercepte les requêtes vers l'API afin de :

- Appliquer la sécurité
- Valider les données
- Transformer les messages
- Limiter le trafic
- Rediriger les réponses vers les différents services
- Mettre les données statiques en cache pour améliorer la performance
- Se connecter aux bases de données

(Pour plus de détail cf. le point 2.3.3)

Le service de routage

Bien qu'il soit possible de mettre en place un service uniquement dédié à gérer les redirections vers d'autres fonctionnalités Backend, il est possible de déléguer cette tâche à un outil d'API management. Elle doit pouvoir identifier et acheminer les demandes vers les bonnes instances, notamment via :

- Le mappage d'URL : le chemin de l'URL entrante peut être différent de celui du service principal. Une fonctionnalité de mappage d'URL permet à la plateforme de modifier le chemin d'accès dans l'URL entrante à celui du service principal. Ce mappage d'URL a lieu au moment de l'exécution, de sorte que le consommateur récupère la ressource demandée via la répartition des services.

- La distribution de service : cette fonctionnalité permet à la plate-forme de sélectionner et d'appeler le service principal approprié. Dans certains cas, il peut être nécessaire d'appeler plusieurs services pour effectuer une sorte d'orchestration et renvoyer une réponse agrégée au consommateur.

- Le regroupement de connexions : la plate-forme d'API management doit pouvoir conserver un pool de connexions au service principal. Le regroupement de connexions améliore les performances globales.

- Le load-balancing : cela consiste à sélectionner un algorithme qui achemine les demandes vers les ressources appropriées afin de distribuer le trafic API vers les services principaux. Divers algorithmes de load-balancing peuvent être pris en charge. Ceci devrait améliorer les performances globales d'une API.

- Service d'orchestration : La fonctionnalité d'orchestration de service permet de créer un service grossier en combinant les résultats de plusieurs appels de services principaux dans une séquence particulière ou en parallèle. Cette fonctionnalité cible également l'amélioration de la performance globale en réduisant la latence que pourraient induire plusieurs appels d'API. L'API Gateway dans une solution d'API management peut jouer un rôle d'orchestrateur (plutôt léger) notamment en ce qui concerne les requêtes sans état et non transactionnelle.

Le contrôle et la gestion de version

En règle générale, le contrôle de version des APIs n'a qu'un objectif : celui d'éviter au maximum de passer d'une version à une autre. En effet, la publication d'une nouvelle version d'API contraint les développeurs d'applications de lancer un nouveau cycle de mise en production avec tout ce que cela comporte :

- Analyse d'impact sur les applications
- Mise à niveau/jour du code
- Débogage
- Déploiements sur différents environnements
- Les tests d'assurance de qualité….

De ce fait, le passage d'une version à l'autre est tout ce qui va **inévitablement** rompre cette communication et rien autre. Ainsi le contrôle de version des APIs doit respecter quelques principes :

Ne pas bloquer le client : Une nouvelle version d'API ne doit bloquer aucun client existant depuis les versions antérieures. En effet, une fois publiée, il faut partir du principe que l'API est utilisée par des développeurs d'applications et changer fréquemment de version augmente le risque de régression et

par conséquent de briser les applications clientes. Ceci peut nuire à la popularité d'une API auprès des développeurs. Il vaut mieux ajouter des modifications compatibles avec les anciennes versions plutôt que d'en ajouter systématiquement de nouvelles. Il s'agit plus d'entretenir cette idée pour faciliter les prises de décisions.

Faire de la rétrocompatibilité : La rétrocompatibilité consiste à apporter des modifications qui n'impacte pas le client. Par exemple :

- Exposer des nouveaux services qui ne sont pas nécessairement utilisés par le client
- Changer la structure d'une réponse en gardant ce qui existait dans les versions ultérieures
- Rendre les nouveaux paramètres d'entrée optionnels/facultatifs…

En principe, ce type de modifications n'impactent pas le client et ne nécessite pas un changement de version.

Oublier le logiciel : Le logiciel évolue très rapidement. Chaque version majeure, les améliorations et les corrections de bogues donnent lieu à une nouvelle version du logiciel. Lier la version du logiciel à celle de l'API entrainerait des changements trop réguliers et qui seraient ingérables par les fournisseurs des API et par conséquent très frustrant par ceux qui les consommes. Cette option ne devrait pas être envisagée.

Le caching

La mise en cache est un mécanisme permettant d'optimiser les performances en répondant aux requêtes avec des réponses statiques stockées en mémoire. L'outil d'API management positionné tel un proxy (API Gateway) dans le plan d'architecture, peut stocker les réponses des requêtes fréquemment appelées et les retourner au lieu d'interroger systématiquement les serveurs principaux en Backend. Cela permet d'améliorer les performances des APIs grâce à une réduction de la latence et du trafic réseau.

Tous les points évoqués ci-dessous forment un ensemble de bonnes pratiques à implémenter. Une solution d'API management adéquate devrait permettre au fournisseur d'APIs de maintenir plusieurs versions d'APIs suffisamment longtemps pour permettre une transition en douceur.

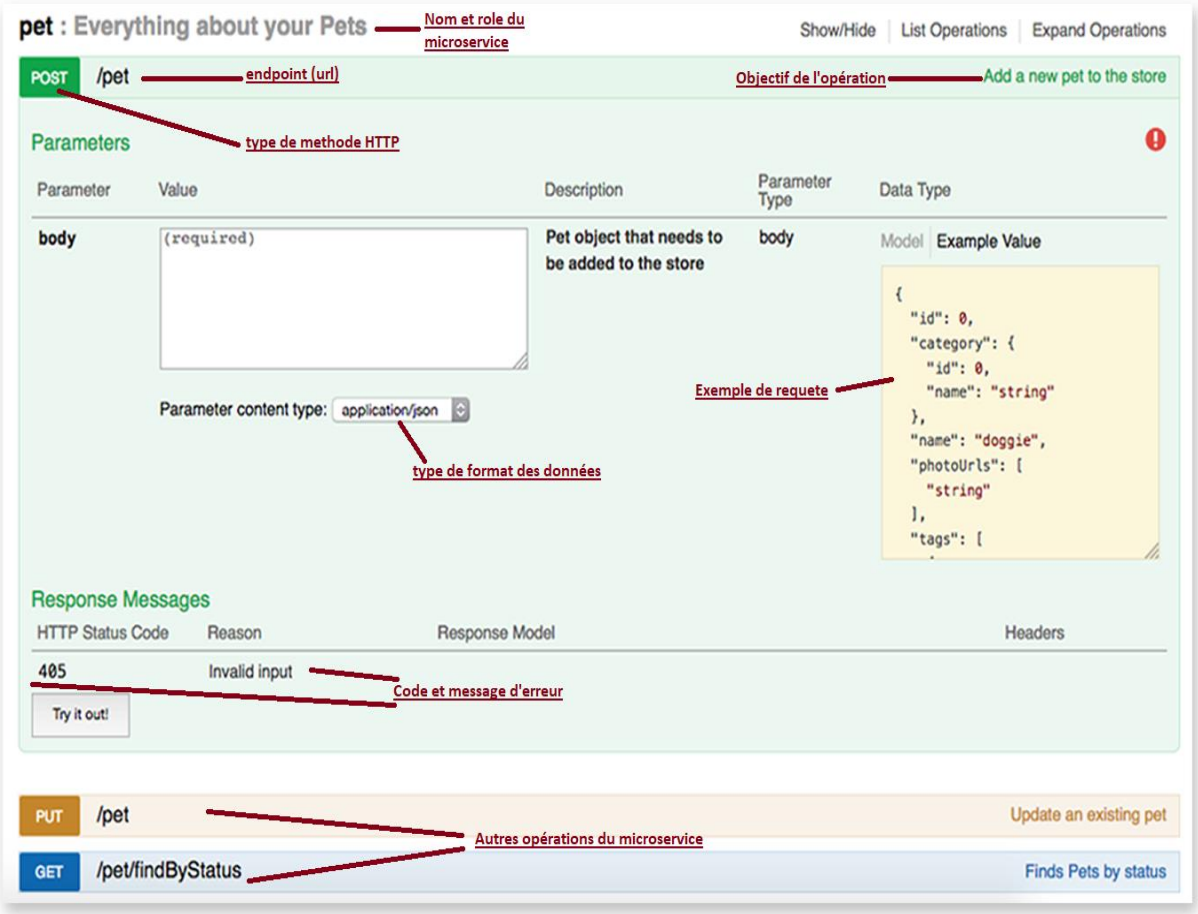
3.3. La documentation

Il a été évoqué à plusieurs reprise que les principaux consommateurs des APIs étaient les développeurs. Et pour ces derniers, la documentation est certainement l'élément essentiel qui pourrait retenir leur attention et les fidéliser car elle communique une grande quantité d'informations utiles et pertinentes sur l'API tel un manuel d'utilisation très détaillé. La documentation d'une API doit présenter certains aspects afin de permettre aux développeurs de se familiariser avec les fonctionnalités de façon intuitive, de commencer à les utiliser facilement et de les adopter rapidement.

Le modèle de documentation

Pour que la documentation de l'API soit efficace et permette aux développeurs d'accélérer leur montée en compétence, elle doit inclure les aspects suivants relatifs à l'API :

- Le nom d'API
- Le nom du microservice et son rôle
- Les endpoints : URLs que les développeurs vont utiliser pour appeler les services présentés par l'API
- Les méthodes : principalement les verbes HTTP (GET, POST, PUT, DELETE)
- Les paramètres dans les URLs
- Les paramètres dans le corps de la requête (payload)
- Les paramètres d'en-tête (Headers)
- Les codes de réponses et d'erreurs
- Un exemple de requête et de réponse avec la structure des données échangées.
- Des tutoriels et visites guidées
- Les SLA : les contrats de niveau de services qui définissent les exigences non fonctionnelles



Capture : Exploration de l'interface de documentation Swagger UI

Les APIs de documentations

Une tache recommandée est d'intégrer un outil qui va générer la documentation de l'API au fur et à mesure que l'API évolue. En effet, le risque principal que l'on court à essayer de gérer manuellement la documentation est de perdre la synchronisation de phase avec l'implémentation réelle. Cela peut arriver en cas d'oubli, de manque de rigueur ou tout simplement faute de temps car documenter une API est une tâche qui a un coût.

Il existe de nombreux outils sur le marché pour documenter une API qui, en important leurs packages et en les incluant dans les configurations des projets, vont être en mesure de générer la documentation en même temps que le déploiement de nouvelles versions de ces applications.

Bien qu'il existe de nombreuses solutions concurrentes, les 3 solutions les plus courantes sont à l'heure actuelle **Swagger**, **RAML** et **Blueprint**

Toutes ces solutions sont disponibles en téléchargement sur GitHub

Critères	Swagger	RAML	API Blueprint
Entreprise	Reverb	MuleSoft	Apiary
Sortie	07/2011	09/2013	04/2013
Communauté	Oui	Oui	Non
Approche design	Top-down et bottom-up	Top-down	Top-down
Outil de creation	Swagger.io	API Designer	Apiary.io
Ad-hoc testing	Swagger UI	API console	Apiary.io
Documentation	Supportée	Supportée	Supportée
Format	JSON	YAML	Markdown
Ressources	Oui	Oui	Oui
Methodes/Actions	Operations	Méthodes	Actions
Paramètres(requetes, URL et Headers)	Oui	Oui	Oui
Code des status	Oui	Oui	Oui
Documentation	Oui	Oui	Oui
Nested resources	Non	OUi	Oui
Exemples d'utilisation	Oui	Oui	Oui
Versionning de l'API	Non	Oui	Oui
Parsing	Java, Js	Java	C++, C# et nodejs
Recherche google par nom	+ 6M	+70K	+ 807K

3.4. La gestion du trafic

Le trafic offre une très grande valeur commerciale à une API quand il s'agit de monétiser cette dernière. En effet, le fournisseur d'API peut proposer à ses utilisateurs un certain nombre d'appels d'API par jour / semaine / mois / illimités et des accès différents en fonction du budget (ou aussi gratuitement, par exemple aux développeurs) et de la période (heures creuses ou pleines). Une plateforme d'API management doit fournir un ensemble de fonctionnalités pour assurer la gestion du trafic.

Fixer le quota de consommation

Le quota de consommation définit le nombre d'appels d'API qu'une application est autorisée à effectuer sur un intervalle donné. Cette limitation est couramment appelée **Throttling**. Les appels dépassant la limite de quota peuvent être encore plus limités ou quasiment interrompus. Le quota autorisé pour une application dépend uniquement de la stratégie commerciale et du modèle de monétisation de l'API. Un objectif commun pour un quota est de diviser les développeurs en catégories, chacune ayant un quota différent et donc une relation différente avec l'API.

Exemple : les développeurs freelances qui s'inscrivent peuvent être autorisés à effectuer un petit nombre d'appels. Mais les développeurs payants (après leur identification) pourraient être autorisés à faire un nombre d'appels plus élevé.

Amorcer un arrêt rapide

L'arrêt rapide est très souvent une mesure prise lorsque la plateforme d'API management identifie une augmentation inattendue du trafic de l'API. Comme nous allons le voir dans le point sur la sécurité, cette mesure aide à protéger les systèmes contre les attaques de déni de service (DDOS) mais pas seulement. Elle aide aussi à protéger les systèmes qui ne sont pas à la base conçus pour gérer une charge élevée

Exemple : Une petite entreprise commerciale durant la période de solde.

Limiter les utilisations

La limitation de l'utilisation fournit un mécanisme permettant de ralentir les appels vers une API. C'est une mesure qui aide à améliorer la performance globale d'un système et à réduire les impacts pendant les heures de pointe. Cela permet de contrecarrer certains problèmes de latence en s'assurant que l'infrastructure de l'API n'est pas ralentie par les volumes importants de requêtes provenant d'un certain groupe de clients ou d'applications.

Une sorte de limitation des utilisations consiste aussi à hiérarchiser les appels. En effet les clients prioritaires sont traités en premier. Cependant toutes les plates-formes d'API management ne prennent pas en charge cette fonctionnalité. Par conséquent, une autre approche ou conception peut être nécessaire pour mettre en œuvre la hiérarchisation du trafic.

3.5. Les métriques

Le trafic API passant par une plate-forme d'API management peut fournir aux entreprises de nombreuses informations utiles, qui peuvent aider à gérer efficacement un programme d'API d'entreprise. En analysant régulièrement ces données, une entreprise peut apporter des améliorations à son API et capitaliser ses investissements. Notamment en termes de popularité, de prévisionnel, d'amélioration, de chiffrage, d'aide à la prise de bonnes décisions

Les informations analysées

Le tableau de bord des métriques d'une solution d'API management permet d'obtenir des réponses selon les informations suivantes :

- Utilisation de l'API au fil du temps
- Pics et creux du trafic
- Les temps de réponses pour jauger la performance
- Les informations sur les utilisateurs principaux (développeurs et applications) et finaux (consommateurs du produit final)
- Le/les microservices de l'API qui génère le trafic maximal
- Comment l'API est utilisée suivant la zone géographique.
- Comment les investissements dans l'API sont rentabilisés.

Les métriques de trafic

Il s'agit de l'ensemble de données opérationnelles et métier collectées à partir du trafic API. Les mesures clés à analyser en provenance du trafic peuvent être classées comme suit :

Trafic au niveau d'une l'API surveillée	- Le total des APIs interagissant avec celle que l'on surveille et qui sont sur le même réseau - La répartition du trafic passant par le proxy - Le trafic par métier ou par actif technique - Les microservices les plus appelés - Les méthodes les plus appelées.
Le temps de réponse	- Le temps de réponse moyen de l'API - Le temps de réponse par service cible

	- La latence de traitement des requêtes et des réponses sur l'API Gateway
Le taux d'erreur	- La répartition des erreurs sur une durée - La répartition des erreurs par API - Le taux d'erreur par service cible - La répartition par code d'erreur (500, 502, 404, etc.)

Les métriques de popularité

Il s'agit de l'ensemble de données dont l'analyse permettrait de jauger la popularité de l'API auprès des développeurs. Cette démarche induit implicitement à une amélioration continue (maintenance) de la plateforme de développement et des bonnes pratiques de l'API du fait du succès ou non de cette dernière.

Les métriques suivantes sont particulièrement à surveiller pour améliorer la popularité :

L'implication des développeurs	- Le total des développeurs enregistrés avec le programme API - Le total des développeurs ayant des applications - Le total des développeurs actifs - Le trafic généré par développeur - Le trafic généré à partir d'applications de développeurs
Les classements dans le trafic	- Le trafic des meilleures applications - Le trafic des meilleurs développeurs - Le trafic des meilleurs produits
L'implication des utilisateurs finaux	- La répartition géographique du trafic API - Répartition par canal (mobile, desktop, tablette etc. - Répartition par systèmes d'exploitation - Répartition par navigateur

Dans certain cas, les données d'analyse par défaut capturées par la plate-forme d'API management à partir du trafic d'API peuvent ne pas être suffisantes pour fournir toutes les informations métier. On devrait alors capturer certaines données personnalisées de la charge utile du message et en tirer des informations d'analyse utiles. De nombreuses plates-formes de gestion d'API permettent d'extraire des données personnalisées à partir de messages et de les enregistrer dans une base de données d'analyse. Ces données peuvent être extraites des en-têtes de message, des paramètres de requête ou des données utiles, et utilisées pour créer des rapports d'analyse personnalisés pertinents.

Exemple : Dans une API de réservation hôtelière, une entreprise peut être intéressée à connaître la répartition des réservations par ville ou hôtel. Ces informations peuvent aider les entreprises à prendre des mesures pour améliorer la satisfaction de leurs clients et développer leurs activités dans les différentes villes selon les conditions externes imposées (bord de mer, montagnes, camping, hiver, été, etc.)

Les analyses API fournissent des informations sur les mesures qui doivent être contrôlées régulièrement pour garantir le succès d'un programme API. Une baisse du trafic API peut signifier que l'intérêt de l'utilisateur s'éloigne de l'API, les raisons pouvant en être nombreuses. Cela pourrait être dû à la faible performance de l'API ou aux déplacements de clients vers des services fournis par d'autres concurrents sur le marché. Les propriétaires d'entreprise devraient examiner de manière critique les rapports d'analyse des API et réfléchir à la manière dont ils devraient alimenter et modifier leurs APIs pour les rendre plus compétitives et plus populaires sur le marché. La mise en œuvre appropriée des analyses d'APIs est la clé du succès du programme API d'une entreprise. La gestion des API est incomplète sans les informations appropriées fournies par les analyses API.

3.6. Le portail de développeur

La majorité des fournisseurs de solutions d'API management met en avant la qualité de leur portail développeur comme argument de vente. En effet, le succès d'une API se mesure à son niveau d'adoption par une communauté de développeurs. Les développeurs aimeraient connaitre l'aptitude d'une API à implémenter leurs applications. Ainsi, en plus de parcourir la documentation de cette dernière, ils ont recours aux blogs et forums pour recueillir les avis honnêtes et chercher des éléments de réponses à leur question. En tant que fournisseur d'une API il est important de mettre à disposition un ensemble de fonctionnalités qui favoriserait une adoption plus rapide

L'enregistrement et la connexion du développeur utilisateur

La plateforme devrait permettre un processus d'enrôlement :

- **Simple** : un grand nombre d'informations peut empêcher le développeur de s'inscrire.
- **Rapide** : Il faut tenir compte de l'impatience que pourrait avoir un développeur à essayer les microservices qu'exposent les API, et par conséquent éviter un excès d'étapes intermédiaires entre l'inscription à la plateforme et son accès.
- **Sécurisé** : les données personnelles des développeurs sont des données personnelles standard et doivent être protégées comme telles. De plus, il faudrait aussi la possibilité d'avoir le droit à l'anonymat.
- **Administré** : en parallèle du portail du développeur, il devrait y avoir un portail administrateur afin de gérer les utilisateurs, notamment en ce qui concerne l'attribution des rôles pour contrôler les privilèges et les droits d'accès, la validation des inscriptions, les blocages de comptes.

Une console pour tester l'API

La mise à disposition d'une console pour tester une Api est un impératif du portail de développeur. Elle devrait permettre d'explorer les microservices et de soumettre du code pour les tester (notion de bac à sable). Elle devrait mettre à disposition la documentation, des exemples réels d'utilisation pour comprendre à facilement utiliser l'API. La console peut également inclure un guide de référence expliquant le vocabulaire commun, les formats de données, les meilleures pratiques, les codes de réponses HTTP courants et les messages d'erreur.

Cette partie à elle seule, peut être très déterminante pour assurer l'adoption des développeurs.

Enregistrement d'application et gestion des clés

Le portail devrait permettre d'enregistrer une application qui serait créé à partir de l'API ou qu'on souhaitera intégrer à l'API. L'approbation peut être automatique ou manuelle mais doit produire une clé d'API pour l'application enregistrée. Le cas d'une approbation manuelle implique qu'un administrateur a examiné et approuvé l'application et par la suite a généré la clé d'API. Un administrateur peut également révoquer des clés ou en régénérer de nouvelles.

Autres fonctionnalités intéressantes

L'enrôlement d'un utilisateur, la console de test et l'enregistrement d'une application constituent les prérequis d'un portail développeur. Cependant certaines fonctionnalités peuvent améliorer l'attractivité de l'utilisation de l'API :

- Les forums et blogs : Dans une stratégie marketing, il serait intéressant d'offrir à une communauté qui adopte une API, la possibilité de produire du contenu pour décrire leurs expériences. Une bonne rétroaction dans les forums et les blogs favorise une adoption plus rapide de l'API.
- Les tableaux de bords : Les développeurs souhaitent connaitre et publier des informations statistiques sur leur applications. Un portail de développeur qui met à disposition un tableau de bord pour obtenir des métriques telles que le nombre d'utilisateurs consommant leurs applications, le nombre d'appels effectués par leurs applications vers l'API, les méthodes les plus utilisées et plus encore, est une bonne valeur ajoutée.
- Un dispositif de support technique : la mise en place d'une équipe ou des informations de support peut être très appréciée. Pour que cela soit fait efficacement, il faudrait mettre à disposition des contacts de développeurs en support (numéro de téléphone, adresse mail) à qui on pourrait poser des questions ou des problèmes liés à l'utilisation de l'API et qui sont particulièrement disponible pour cette tâche. En ce qui concerne les informations de support, la mise en place d'une page regroupant des FAQ, des avis ou des informations à venir devrait parfaitement convenir et devrait être régulièrement mise à jour.

3.7. Quelques solutions sur le marché

Le marché des solutions d'API management est en voie de consolidation, si l'on considère des rachats significatifs qui ont eu lieu ces dernières années : Google ayant racheté Apigee, Tibco ayant racheté Mashery, CA Technology s'étant offert Layer7 et Axway ayant rajouté Vordel à son catalogue.

Hormis les différents mouvements d'acquisition, la couverture fonctionnelle de ces solutions tend à s'homogénéiser à tel point qu'on pourrait parler d'uniformisation de l'offre des solutions d'API. Elles proposent toutes les solutions qui constituent la base permettant de piloter et gérer des APIs. De ce fait

le choix de l'une d'entre elles est relativement difficile dans la mesure où les services proposés et les messages véhiculés sont à peu près similaires.

Le cabinet de conseil ***IT Central Station***, qui effectue des sondages auprès de professionnel de l'IT, afin d'aider à la prise de décision concernant différentes acquisitions digitales, a mené une enquête auprès de 365 192 professionnels et a pu établir le guide des revues des acheteurs de solutions d'API management en septembre 2019.

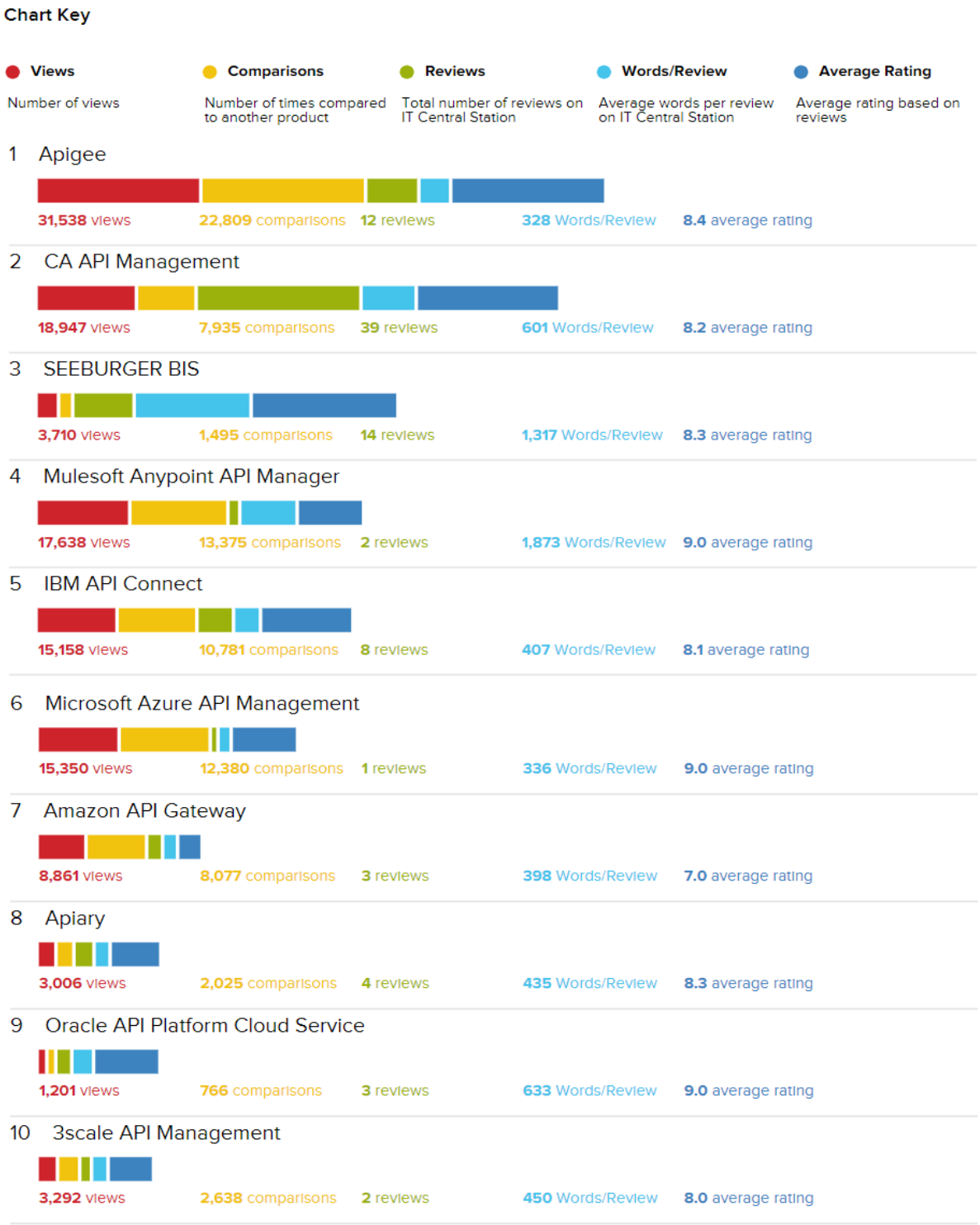
Ci-dessous le répertoire des différents fournisseurs du marché :

Amazon	Amazon API Gateway
Axway	Axway API Management Plus
CA Technologies	CA API Management
Cloud Elements	Cloud Elements
Google	Apigee
IBM	IBM API Connect
Kong	Kong Enterprise
Microshare.io	Microshare.io
Microsoft	Microsoft Azure API Management
MuleSoft	Mulesoft Anypoint API Manager
OpenLegacy	OpenLegacy
Oracle	Oracle API Management
Oracle	Apiary
Oracle	Oracle API Platform Cloud Service
Oracle	Oracle Communications Services Gatekeeper
PokitDok	PokitDok
Red Hat	3scale API Management
Rogue Wave	Akana API Management
Seeburger	SEEBURGER BIS
Sikka Software	Sikka Software
SmartBear	SwaggerHub
Software AG	webMethods API Management
Software AG	CentraSite
Software AG	Software AG webMethods Integration Cloud
Talena	Talena
TIBCO	TIBCO Mashery API Management
Torry Harris Business Solutions	DigitMarket
Tyk	Tyk
WSO2	WSO2 API Manager

Source : API Management Buyer's Guide and Reviews September 2019

Document commandé à ***IT Central Station****, cabinet de sondages de solutions IT*

Suites aux revues et comparaisons, des notes ont été déduites et le classement suivant des 10 premières solutions a été établit :



Source : API Management Buyer's Guide and Reviews September 2019

*Document commandé à **IT Central Station**, cabinet de sondages de solutions IT*

Suite à ces statistiques, il est intéressant de remarquer, que les 5 solutions suivantes ont été fortement comparées aux autres produits, suscitant ainsi l'intérêt de la majorité :

1 – Apigee

2 - MuleSoft AnyPoint API Manager

3 - Microsoft Azure API management

4 - IBM API Connect

5 – Amazon API Gateway

Intéressons-nous de plus près aux 3 premières solutions :

Apigee

Google Apigee, est une plate-forme de gestion des APIs qui permet aux entreprises de sécuriser, de faire évoluer, de gérer et d'analyser leurs activités numériques et de développer les programmes d'API pour répondre à la demande croissante. Elle permet également aux entreprises de concevoir et de construire des API qui partagent en toute sécurité leurs services et leurs données.

Les fonctionnalités clés de la plate-forme de gestion des APIs Apigee incluent :

- La Sécurité : protégez les APIs, les messages et les backends avec des stratégies configurables telles que OAuth2, la vérification des clés d'API, la protection contre les menaces XML / JSON, le contrôle d'accès.
- L'API Gateway : sécurise et gère le trafic entre les clients et les applications principales, ainsi qu'entre les APIs d'une société et les développeurs, les clients, les partenaires et les employés qui utilisent les APIs.
- Un Portail des développeurs : attire et engage les développeurs d'applications pour leur permettre de découvrir, d'explorer, d'acheter (ou d'en tirer profit), de tester des API et de s'inscrire pour accéder aux API et les utiliser.
- Gestion du cycle de vie des API : gère le processus de conception, de développement, de publication, de déploiement et de version des API.

Quelques clients de la plateforme : Adobe, AT&T, Citrix, Dell, Equifax, HCSC, Orange, Pearson, etc.

Les axes d'amélioration selon l'avis des clients :

- Mauvais support des webhooks (fonctions de rappel http définies par l'utilisateur, qui récupère et stockent les données issues d'un évènement, en général externe à l'application).
- La configuration initiale de gestion de version par URI n'est pas triviale.
- Le modèle de support technique pour l'OPDK (kit de déploiement sur site) n'est pas assez mature.

MuleSoft AnyPoint API Manager

API Manager est un composant de la plate-forme Anypoint de MuleSoft pour la conception, la construction, la gestion et la publication d'API. Anypoint Platform utilise Mule comme moteur d'exécution principal. On peut utiliser API Manager sur un cloud public, tel que CloudHub, un cloud privé ou un système hybride. Le déploiement Ahybrid est une API déployée sur un serveur privé mais dont les métadonnées sont traitées dans le cloud public.

Les fonctionnalités de la plateforme Mulesoft AnyPoint incluent :

- La gestion et adaptation centralisées des programmes d'APIs (sécurisation des APIs et microservices, portails personnalisables, gestion des accès de façon native ou intégrée) l'intégration, le B2B/EDI sur une seule plateforme.
- La construction rapide des projets en réutilisant les ressources de sa propre entreprise et de l'écosystème MuleSoft.
- La connexion aux pipelines CI/CD grâce à la prise en charge de Maven et Jenkins.
- Le développement sur des plateformes sécurisées conformes aux normes ISO 27001, SOC 1, SOC 2, PCI DSS et RGPD.
- La tokenisation des données sensibles.
- La disponibilité des applications à 99,99%, les métriques, les alertes et tableaux de bords, les tests.
- Le passage du cloud au multi-cloud.

Quelques clients de la plateforme : Coca-Cola, Splunk, Citrix, National Post, Justice System, etc.

Les axes d'amélioration selon l'avis des clients :

- Aucune information n'est diffusée en ce qui concerne la fusion avec Salesforce, les professionnels se posent des questions.

- Ne dispose pas de produit BPM/Workflow, BAM ou de gestion d'identités.
- Les interfaces utilisateurs ne sont pas user-friendly par rapport à la concurrence.
- Le cadre de gestion des transactions n'est pas directement fourni.
- Le cout de la licence qui se trouve entre 300 000$-400 000$.

Microsoft Azure API Management

Microsoft Azure API Management est une solution permettant à une organisation de gérer son programme d'API. Avec la gestion des APIs, les organisations peuvent publier des services Web sous forme d'API fiables, sécurisées et à grande échelle, utiliser la gestion des APIs pour gérer la consommation de partenaires internes, de partenaires et de développeurs, tout en tirant parti des informations commerciales et opérationnelles du portail.

Les principales fonctionnalités proposées par Azure API Management sont :

- Création d'API Gateway et d'un portail de développement
- Gestion des APIs dans un seul emplacement
- Sécurisation et optimisation des APIs, identité
- Documentation
- Connections aux services principaux en tout lieu
- Tableau de bord personnalisé
- Outils de développements, mise en réseau, stockage
- Métriques, analyses, AI et Machine Learning
- Intégration et migration

Quelques clients de la plateforme : adnymics GmbH, LG CNS, AccueWeather, Toyota, Medplast…

Les axes d'amélioration selon l'avis des clients :

- Console de développement pas assez intuitive.
- La console de développement a besoin d'être améliorée pour les tests.
- La gestion de la sécurité des API n'est pas assez fine.

Comme montré dans le répertoire des solutions d'API management, il existe un large panel de solution. La finalité de ces solutions est d'« APIser » un système, mais afin de faciliter la prise de décision, il peut s'avérer important de bien étudier ce dernier au préalable :

- Designer les APIs avec des documentations claires
- Faire des services modulaires
- Définir sa politique de gestion de version des services
- Savoir exactement quels traitements exposer ou pas
- Avoir des métriques

Avoir une idée de ses éléments devrait non seulement aider à la prise de décision mais aussi faire en sorte que l'implémentation de cette nouvelle brique au sein de l'architecture se passe sans difficultés bloquante avec une bonne gestion des couts d'utilisations.

4. La Sécurité

De base, les systèmes à grande échelle qui exposent les données sont sujet aux failles de sécurité. Dans une ère comme celle -ci, où les clients et les entreprises ont conscience de la valeur économique des données, il est indispensable de ne pas négliger le volet de la sécurité lors de la conception et l'implémentation des systèmes. Il est nécessaire de réfléchir à une stratégie de protection des données lorsqu'elles transitent sur un réseau, ainsi que de la protection de ce réseau lui-même.

Le défi que représente les microservices en termes de sécurité vient de leur surface d'attaque qui est bien plus grande que celle des applications monolithiques. En effet, dans une architecture de microservices, il ne s'agit plus d'assurer la protection d'une ou deux applications monolithiques, mais plutôt de celles de plusieurs dizaines de petits services qui interagissent les uns avec les autres de plusieurs manières. Sécuriser des microservices revient à les protéger des agressions extérieures mais aussi de mauvais usages internes :

- Les accès aux microservices doivent être authentifiés et autorisés
- Les communications entre microservices doivent être sécurisées
- Les restrictions doivent être mise en place entre les services et les utilisateurs
- Les opérations doivent être enregistrée pour assurer la traçabilité

Ces actions correctement suivies devraient permettre de constituer un bon niveau de sécurité. Voyons un peu plus en détails.

4.1. L'authentification et l'autorisation

L'authentification et l'autorisation sont des concepts fondamentaux pour les personnes et les éléments qui interagissent avec un système.

L'authentification est un processus par lequel on confirme l'identité d'une tierce partie (login/mot de passe, clé d'accès, jeton de sécurité, empreinte digitale, etc.) pour lui permettre d'accéder à une information.

L'autorisation est un mécanisme de mappage de l'authentifié à l'ensemble des actions qu'il est autorisé à effectuer sur les informations (lecture, écriture, etc.) en fonction de ses informations remontée par l'authentification.

Une approche commune en matière d'authentification et d'autorisation consiste à utiliser une **solution d'authentification unique** (SSO) auprès d'un **fournisseur d'identité** qui valide les informations d'accès aux données et l'accès aux ressources. Le **SAML** est l'implémentation en vigueur dans l'espace d'entreprise et **OpenID Connect** dans le domaine public, mais ce dernier gagne en popularité notamment grâce à des travaux importants sur le protocole de sécurité OAuth.

Un fournisseur d'identité peut être un système hébergé en externe ou au sein de l'entreprise. Par exemple, Google et Facebook sont des fournisseurs d'identité OpenID Connect et pour les entreprises il est courant d'avoir une solution telle que **Keycloak** qui est liée à un service d'annuaire comme le protocole LDAP ou Active Directory. Ces systèmes permettent de stocker des informations sur les utilisateurs et leurs rôles dans l'organisation.

SAML est un standard basé sur SOAP et est connu pour être assez complexe à utiliser malgré les bibliothèques et les outils disponibles pour le prendre en charge. OpenID Connect est un standard qui a émergé comme une implémentation spécifique d'**OAuth 2.0**, basé sur la façon dont Google et d'autres géants du Web traitent l'authentification unique. Ce dernier utilise les appels REST plébiscité par les architectures de microservices et est de plus en plus populaire au sein des entreprises en raison de sa facilité d'utilisation qui a été continuellement améliorée.

- Focus sur OpenID Connect (OIDC) :

Bien qu'une entreprise souhaitant garder davantage de contrôle et de visibilité sur le mode et l'emplacement de ses données partirait sur un fournisseur d'identité interne, il est t'important de

maintenir une veille de sur l'avancée des technologies des fournisseurs externes tels que Google qui implémente de l'OpenID Connect.

OICD est une couche d'authentification basée sur le protocole OAuth 2.0, qui autorise les clients à vérifier l'identité d'une partie tierce en se basant sur l'autorisation fournie par un serveur d'autorisation en suivant le processus d'obtention d'informations du profil de l'utilisateur final. Il spécifie une interface HTTP RESTful en utilisant le JSON comme format de données, ce qui lui permet notamment d'échanger des informations avec un vaste panel de clients web, mobiles et utilisant des frameworks Javascript.

OAuth2 (voir : http://tools.ietf.org/html/rfc6749) est un protocole qui permet à des applications tierces d'obtenir un accès limité à un service via HTTP à partir d'une autorisation du détenteur des ressources. Entre d'autres termes, si un client (applications, site internet…) demande accès à des ressources, il doit obtenir des autorisations ou s'identifier à la solution qui héberge les ressources.

OAuth2 définit 4 rôles principaux :

- Le détenteur des données
- Le serveur de ressources
- Le client
- Le serveur d'Autorisation

On distingue 2 types de tokens :

- L'Access Token : il donne directement accès aux ressources
- Le Refresh Token : permet de lancer la demande d'un Access Token quand ce dernier est expiré.

Un Token peut être envoyé de plusieurs façon au serveur de ressource :

- Directement dans une requête GET ou POST (non recommandé)
- En entête des requêtes (Headers- Authorization)

Selon l'emplacement et la nature des données, peut choisir parmi 4 types d'autorisations :

- L'autorisation via un code : permet d'obtenir les Access et Refresh Tokens pour accéder directement à la ressource. Le client est un serveur web.
- L'autorisation implicite : Permet d'obtenir un Access Token mais pas le Refresh Token. Le client est une application web

- **L'autorisation via un mot de passe** : les identifiants sont envoyés au client et au serveur d'autorisation
- **L'autorisation serveur à serveur** : Quand le client est lui-même le détenteur des ressources il se flague pour ne pas s'empêcher lui -même d'accéder aux données.

Il est à noter que OAuth2 correspond à **99%** des exigences et typologies du client car il permet, contrairement à OAuth1, de gérer l'authentification et l'habilitation des ressources par tout type d'application (native mobile, native tablette, application javascript, application de type serveur, batch/back-office, …). De ce fait, il est le standard par excellence de sécurisation des API.

NB : Il faut systématiquement utiliser le protocole **HTTPS** pour l'utiliser.

4.2. La sécurité entre les services

La plupart des organisations assurent la sécurité sur le périmètre de leur réseau en contrôlant les accès aux ressources par les utilisateurs et supposent par conséquent qu'il n'y a pas de nécessité à implémenter de la sécurité entre les services quand ces derniers communiquent entre eux à l'intérieur d'un réseau. C'est sans compter sur le fait que les pirates informatiques opèrent généralement en interceptant, lisant et modifiant des données et en s'insérant dans un réseau pour commettre des impairs. Selon le niveau de sensibilité des données, il est nécessaire de pousser plus loin le modèle de confiance au-delà du réseau en implémentant des modèles de sécurité à l'échelle du service.

- L'authentification HTTPS basique

Via HTTP, les données de connexion ne sont pas envoyées de manière sécurisée vers les serveurs d'authentification et transitent par toutes les parties intermédiaires qui peuvent consulter ces informations dans les entêtes des requêtes.

L'authentification HTTPS qui est la combinaison du HTTP avec un couche de chiffrement telle que **SSL** (Secure Socket Layer) ou **TLS** (Transport Layer Security) devient un niveau de sécurité standard permettant à un client d'obtenir la garantie que le serveur avec lequel il communique est bien celui avec qui il est supposé établir cette communication. Les services intermédiaires ne pourraient, à priori, pas lire les données d'authentification en entête et juste router les requêtes vers leur destinataire final.

Cependant le serveur doit gérer ses propres certificats SSL, ce qui a tendance à présenter un fardeau administratif et opérationnel supplémentaire avec la génération et le partage des certificats.

- Les certificats clients

Une approche pour confirmer l'identité d'un client consiste à utiliser les fonctionnalités de TLS (qui succède à SSL) sous forme de certificats de client. Ainsi chaque client dispose d'un certificat (X.509) permettant d'établir un lien avec le serveur. Ce dernier vérifiera l'authenticité du certificat client en fournissant des garanties solides que ce client est valide.

Le défi de ce processus repose en partie sur la gestion des certificats qui est plus laborieuse que leur utilisation proprement dite. La création d'un certificat est très procédurale, sujette à des erreurs qui entrainent des révocations et réémission. Cette technique est à envisager si la nature des données est très sensible ou si la maitrise des données qui transitent sur le réseau est faible. Il est fréquent de rencontrer ce processus répandu dans des institutions bancaires.

- Le code d'authentification d'une empreinte cryptographique de message avec clé

En termes de sécurité, une meilleure alternative au HTTP serait le HTTPS, mais la gestion des certificats et du trafic HTTPS impose une charge supplémentaire aux serveurs. Une autre approche, largement utilisée par les APIs S3 d'Amazon pour AWS et dans certaines parties de la spécification OAuth, consiste à utiliser un code d'authentification d'une empreinte cryptographique de message avec clé, communément appelé HMAC, calculé en utilisant une fonction de hachage cryptographie combinée à une clé privée, pour signer une requête.

Avec le HMAC, le corps d'une requête ainsi qu'une clé privée sont hachées et le résultat du hachage est envoyée dans le corps de la requête. Le serveur utilise ensuite sa propre copie de clé privée et le corps de la requête pour recréer le hachage, si les deux HMACs correspondent, la requête est acceptée et traitée. L'avantage de ce processus c'est qu'il réduit considérablement les tentatives d'interception des données qui auront alors tendance à changer la signature des requêtes et le fait que la clé privée n'étant jamais envoyée dans la requête, elle n'est pas comprise en cours de route.

L'inconvénient de cette approche réside dans le partage de cette clé privée. Elle peut soit être mise en dure, soit gérée et envoyée par une tierce partie qui se doit d'être particulièrement sécurisé. De plus, la notion de HMAC n'est pas normée, donc tout le monde est libre d'implémenter les fonctions de hachage comme il le souhaite, cependant il vaut mieux partir sur une fonction sensible avec une clé suffisamment longue de type SHA-256. Cette approche n'est garantie que si la clé privée reste privée !

4.3. La sécurité entre les APIs

En raison du fait que la plupart des APIs sur les réseaux publics sont accessibles à tous, elles sont vulnérables à diverses attaques pirates. La question de la sécurité est très cruciale dans le choix d'une solution d'API management. S'agissant d'un enjeu majeur et critique, il convient donc de s'assurer qu'un ensemble de mesure de sécurité soit mis en place afin de protéger les données.

L'authentification d'une API via une clé

Une API est identifiée par son nom et par un UUID unique appelé clé d'application ou clé d'API ou encore ID client. Elle est émise par l'entreprise qui expose l'API ainsi que d'autre informations concernant les accès. Une plateforme d'API management devrait être en mesure de créer, d'émettre, gérer, suivre et révoquer l'UUID. Il devrait donc être impossible de franchir le premier pallier de sécurité s'il manque dans les appels de services, l'UUID de l'API à laquelle on souhaite accéder.

Certains services d'authentification peuvent également nécessiter une intégration avec des systèmes de gestion des identités qui contrôlent l'accès des utilisateurs aux API. Il est possible de générer manuellement des UUID, directement à l'accueil du site https://www.uuidgenerator.net/. Le site propose aussi plusieurs services au travers d'une API permettant de générer de UUID de diverse catégorie : https://www.uuidgenerator.net/api.

L'autorisation à l'aide d'un token

L'autorisation contrôle le niveau d'accès fourni à une application effectuant un appel API. Il contrôle les ressources et les méthodes de l'API qu'une application peut appeler. Lorsqu'une application effectue un appel d'API, elle transmet normalement un jeton d'accès OAuth dans les en-têtes HTTP. Ce jeton de sécurité permet de définir la portée d'accessibilité des APIs (limite à une ou plusieurs APIs ou services) durant une période bien définie pouvant aller de plusieurs secondes, minutes, jours ou mois.

Si le jeton a expiré, un nouveau jeton d'accès doit être généré. Une solution d'API management peut le faire automatiquement en présentant un jeton d'actualisation (le '*refresh Token*'). Le jeton d'actualisation devra être échangé pour obtenir un nouveau jeton d'accès avec une nouvelle période de validité.

La médiation d'identité

Les APIs utilisent généralement les protocoles OAuth (plus spécifiquement OAuth2) pour implémenter la sécurité. Toutefois, les services principaux peuvent être sécurisés à l'aide de SAML ou de tout autre entête WS-Security. Par conséquent, la plate-forme de gestion d'API doit pouvoir s'intégrer aux plates-formes IDM backend et faire la médiation d'identité. *'OAuth to SAML'* est une exigence de médiation d'identité très courante.

4.4. Quelques menaces à considérer

Toutes les failles de sécurité représentent un risque qui peut être, ou non, suffisamment sérieux pour mériter une attention particulière. Cependant, certaines d'entre elles peuvent être sans conséquence tandis que d'autre peuvent provoquer la mise en faillite de toute une organisation.

Les menaces les plus fréquentes à considérer, autant sur des applications basées sur des architectures de microservices ou autre, sont :

L'injection : Les failles d'injection, telles que l'injection SQL, NoSQL, OS et LDAP, se produisent lorsque des données non fiables sont envoyées à un interpréteur dans le cadre d'une commande ou d'une requête. Les données hostiles de l'attaquant peuvent amener l'interprète à exécuter des commandes involontaires ou à accéder à des données sans autorisation préalable.

L'authentification brisée : Les fonctions d'application liées à l'authentification et à la gestion de session sont souvent implémentées de manière incorrecte, ce qui permet aux attaquants de compromettre les mots de passe, les clés ou les jetons de session, ou d'exploiter d'autres failles d'implémentation afin d'assimiler, de façon temporaire ou permanente, l'identité des autres utilisateurs.

L'exposition des données sensibles : De nombreuses applications (Mobile/Web) et API ne protègent pas correctement les données sensibles, telles que les données financières, les soins de santé et les données personnelles. Les attaquants peuvent voler ou modifier ces données faiblement protégées pour commettre une fraude sur carte de crédit, un vol d'identité ou d'autres infractions. Les données sensibles peuvent être compromises sans protection supplémentaire, telle que le cryptage à l'arrêt ou en transit, et nécessitent des précautions particulières lorsqu'elles sont échangées avec le navigateur.

Le déni de service (DDoS) : Les attaques par déni de service consistent à faire tomber des systèmes Backend en augmentant significativement le trafic via les APIs. Par conséquent la solution d'API management devrait être capable de détecter cette anomalie et prendre les mesures nécessaires.

Le manque de contrôle d'accès : Les restrictions sur ce que les utilisateurs authentifiés sont autorisés à faire ne sont souvent pas correctement appliquées. Les pirates peuvent exploiter ces vulnérabilités pour accéder à des fonctionnalités et / ou à des données non autorisées, telles qu'accéder aux comptes d'autres utilisateurs, afficher des fichiers sensibles, modifier les données d'autres utilisateurs, modifier les droits d'accès, etc.

Les mauvaises configurations : La mauvaise configuration de la sécurité est le problème le plus fréquemment rencontré. Cela est généralement dû à des configurations par défaut non sécurisées, à des configurations incomplètes ou ad hoc, à des en-têtes HTTP mal configurés et à des messages d'erreur détaillés contenant des informations sensibles. Tous les systèmes d'exploitation, infrastructures, bibliothèques et applications doivent non seulement être configurés de manière sécurisée, mais ils doivent également être corrigés et mis à niveau rapidement.

Le cross-Site Scripting (XSS) : Des failles XSS se produisent chaque fois qu'une application inclut des données non fiables dans une nouvelle page Web sans validation ou échappement correct, ou met à jour une page Web existante avec des données fournies par l'utilisateur à l'aide d'une API de navigateur pouvant créer du HTML ou du JavaScript. Le XSS permet aux attaquants d'exécuter des scripts dans le navigateur de la victime, ce qui peut détourner les sessions utilisateurs, altérer les sites Web ou rediriger l'utilisateur vers des sites malveillants.

La désérialisation non sécurisée : Cela conduit souvent à l'exécution de code à distance. Même si les failles de désérialisation n'entraînent pas l'exécution de code à distance, elles peuvent être utilisées pour effectuer des attaques, notamment des attaques par rejeu, des attaques par injection et des attaques par élévation de privilèges.

L'utilisation de composants externes non sécurisé : Les composants, tels que les bibliothèques, les frameworks et d'autres modules logiciels, s'exécutent avec les mêmes privilèges que l'application. Si un composant vulnérable est exploité, une telle attaque peut faciliter la perte de données ou la prise de contrôle de serveur. Les applications et les APIs utilisant des composants avec des vulnérabilités connues peuvent faire tomber les défenses des applications et permettre diverses attaques et impacts.

Le manque de logging et de monitoring : Une journalisation et une surveillance insuffisantes, associées à une intégration manquante ou inefficace avec la réponse à un incident, permettent aux pirates d'attaquer les systèmes, de maintenir la persistance, de faire pivoter un plus grand nombre de systèmes et de falsifier, extraire ou détruire des données. La plupart des études sur les violations indiquent que le temps nécessaire pour détecter une violation est supérieur à 200 jours, généralement détecté par des parties externes plutôt que par des processus internes ou une surveillance.

Bien que certains risques soient faciles à détecter, il en restera d'autres bien plus difficile à déterminer. Il est important de prendre connaissances des risques les plus fréquents afin d'augmenter la difficulté à détecter les failles de votre système par les pirates.

NOTA BENE : la fondation OWASP, met régulièrement à jour les informations sur le top 10 des risques de sécurité les plus fréquents et les plus critiques constatés sur les applications web (www.owasp.org)

5. Etude du cas CoMove

Présentation

CoMove est une idée d'application de covoiturage urbain dont l'idée est la suivante :

Lors d'un déplacement urbain ou péri-urbain l'automobiliste ouvert au covoiturage rentre dans l'application sa destination. Il dispose alors des fonctions classiques d'un GPS pour l'y guider.

- Le piéton qui souhaite « covoiturer » descend dans la rue et rentre dans l'application sa destination.
- Un système informatique va en permanence comparer les trajets prévus par les automobilistes et les piétons. Il va repérer si un automobiliste pourrait amener un piéton à proximité de sa destination, moyennant un détour limité pour aller le chercher. Si c'est le cas, il va proposer à l'automobiliste de prendre en charge le piéton en indiquant la nature du détour (temps et distance estimés).
- L'automobiliste peut accepter la prise en charge et se faire guider vers le piéton. Ce dernier recevra alors confirmation qu'un automobiliste vient à sa rencontre avec une description de son véhicule.
- Le système détecte que le piéton a été pris en charge en comparant les positions GPS des deux covoitureurs et garde trace du parcours effectué en commun afin de calculer le coût du partage (sur la base des forfaits fiscaux kilométriques)
- En fin de mois, le système organise le transfert d'argent entre les deux covoitureurs en prélevant une commission.

Principes du logiciel :

Application Conducteur :

L'application Conducteur fonctionnera sur le principe suivant :

- Le conducteur crée un compte avec ses informations personnelles, la description de son véhicule et ses informations bancaires de manière à être rémunéré.
- Lorsqu'il démarre un trajet sur lequel il accepte de prendre des piétons, il renseigne sa destination dans l'application et le nombre de places disponibles. Via une fonction GPS, celle-ci lui indique le trajet préconisé et les temps estimé d'arrivée à destination.
- Lorsque le système estime que l'automobiliste peut prendre un piéton, il alerte le conducteur en indiquant le profil du piéton et la nature du détour (en temps et distance).
- Le conducteur dispose alors de 30 secondes pour accepter de prendre en charge le piéton en cliquant sur l'application. Au-delà de ce délai, la proposition disparait (et le système part à la recherche d'un autre conducteur).

- Si l'automobiliste accepte la prise en charge, l'application le guide alors vers le piéton.
- Lorsque l'application constate que le mobile du piéton se déplace de manière identique au véhicule il en déduit que la course a commencé. A contrario, quand l'application constate que les deux mobiles ne se déplacent plus de la même manière il en déduit que la course est terminée (un mode manuel permet aussi de signaler la fin d'une course).
- A la fin de la course, le conducteur se voit créditer une rémunération correspondant aux frais de partage de la course. Il peut également évaluer le piéton.

Application Piéton :

L'application Piéton fonctionnera sur le principe suivant :
- Dans la rue, le piéton démarre l'application et renseigne sa destination.
- Lorsqu'un automobiliste accepte de prendre en charge le piéton ce dernier reçoit une confirmation sur son mobile que quelqu'un va venir le chercher. Cette confirmation s'accompagne d'une description du profil du conducteur et de son véhicule.
- Après être monté dans le véhicule, en comparant le déplacement des deux mobiles, le système déclare que la course a commencé.
- Lorsque le système constate que les déplacements des mobiles sont différents, il en déduit que la course est terminée.
- Il débite alors le compte du piéton et crédite celui de l'automobiliste, sur la base d'un partage des frais kilométriques (en suivant le barème fiscal en vigueur), moyennant une commission pour la société CoMove.
- Le piéton peut aussi évaluer son conducteur.

Fonctions principales du système :

Le système doit :
- Suivre en permanence les positions des automobilistes et des piétons déclarés.
- Identifier l'automobiliste le plus pertinent pour chaque piéton. Un automobiliste sera jugé pertinent si :
    - o le temps de prise en charge du piéton est faible,
    - o le détour nécessaire pour prendre en charge le piéton est limité,
    - o le trajet de l'automobiliste permet de déposer le piéton à une distance raisonnable de sa destination
- Identifier automatiquement les trajets effectués en co-voiturage, calculer leur distance et établir une redevance que le piéton devra verser à l'automobiliste, sur la base d'un partage de frais.
- Gérer de manière automatique les transferts financiers entre utilisateurs à chaque fin de mois

Le système doit aussi pouvoir générer des rapports individuels (pour les automobilistes et les piétons) et des rapports généraux sur l'usage du covoiturage (analyse de distances, temps passé, temps d'attente, taux d'occupation, etc. par heure de la journée, jour de la semaine, quartiers d'origine et destination, etc.) permettant à la société CoMove de mesurer sa contribution écologique.

Exigences non fonctionnelles :

Le système doit fonctionner H24 7/7 avec une très forte disponibilité.
Il doit être réactif en suivant le déplacement des véhicules toutes les 5 secondes et en proposant un « match » entre conducteur et piéton en 5 secondes (s'il existe un « match » évidemment).
Il doit supporter la charge suivante :
- Utilisation de l'application Conducteur sur 10% des déplacements urbains
- Utilisation de l'application Piéton par un nombre double de celui des conducteurs

Cette application devra s'intégrer à celles du groupe Ovalie qui gère des réseaux de transport (bus, métro, trams) dans de nombreuses agglomérations.
Le groupe Ovalie est dans une démarche de rénovation digitale.
Le système sera déployé sur l'aire urbaine de Toulouse et, si le système fonctionne, il sera étendu ensuite aux grandes villes françaises, puis mondiales.

Pourquoi faire des microservices ?

Bien qu'il s'agisse d'un cas d'utilisation permettant de démontrer par l'exemple la mise en place d'une architecture de microservices, il est important de faire remonter les points permettant de justifier qu'on aurait intérêt à se tourner vers cette architecture.

A la lecture des exigences de ce projet nous pouvons remonter quelques éléments et poser des réflexions :

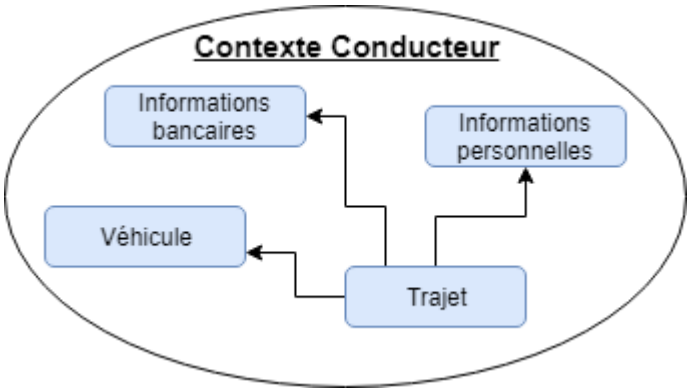
Exigences	Réflexions
Le système devra être porté par 2 applications mobiles, celle du conducteur et du piéton	- Approche Mobile-First - Api Gateway - Sécurité
Le système doit être hautement disponible à raison de 7j/7, 24H/24	- Tolérance aux pannes - Disponibilité
Le système doit être capable de supporter la montée en charge	- Scalabilité horizontale - Infrastructure cloud pour serveurs à la demande
Le système nécessite l'utilisation de fonctionnalités externes telles que le GPS, la banque et une fonctionnalité de notation	- Recherche d'API publiques/ privées pour utilisation de services tiers
Le système doit remonter un ensemble de métriques (analyse de distances, temps passé, temps d'attente, taux d'occupation etc. par heure/jour/semaines/quartiers d'origine et destination, etc.) permettant de mesurer la contribution écologique de la société CoMove	- Mise en place d'une solution de traçage de données directement dans le code - Utilisation de service de métrique (possibilité de mettre en place une solution d'API management)
Le système doit être à jour dans ses technologies car il sera intégré au système Ovalie qui est en pleine rénovation digitale	- Microservices, petits et modulaires facilitant l'ajout de nouvelles fonctionnalités - APIs facilitant l'intégration à d'autres systèmes
Le système doit pouvoir exposer des services car il sera intégré à une nouvelle offre de service avec Ovalie	- APIs facilitant l'intégration à d'autres systèmes - Outil d'API management pour gérer l'intégration avec d'autres APIs
Il faudra prévoir que l'application soit hautement distribuée au-delà de l'agglomération de Toulouse	- Nécessité de concevoir des services petit, modulaires, portés par le cloud avec de faciliter la distribution dès le départ.

Conception

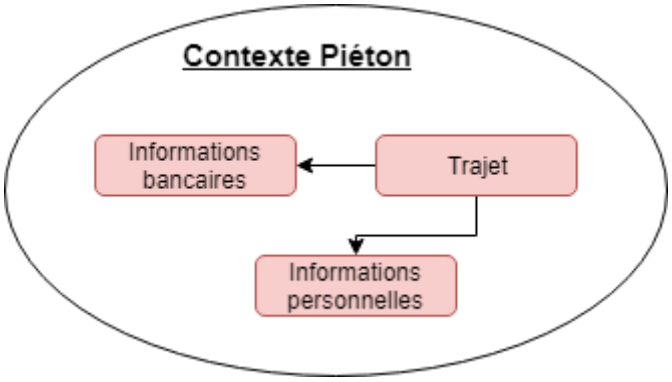
Comme présenté dans la phase de conception de ce document nous allons implémenter des techniques de Domain-Driven-Design.

Découpons l'application en rassemblant logiquement des informations liées à des domaines définis, nous devrions à la fin de cet exercice avoir une liste de tous les contextes bornés du système et un aperçu des premières APIs et équipes de développement

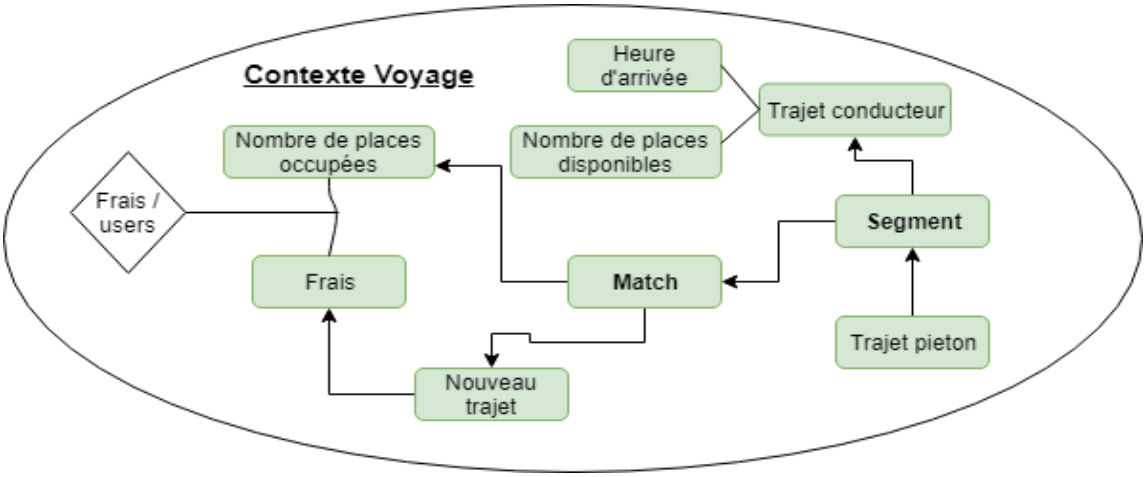
Contexte Conducteur : Ce contexte prend en charge des données liées à un conducteur, son véhicule ses informations personnelles et son trajet



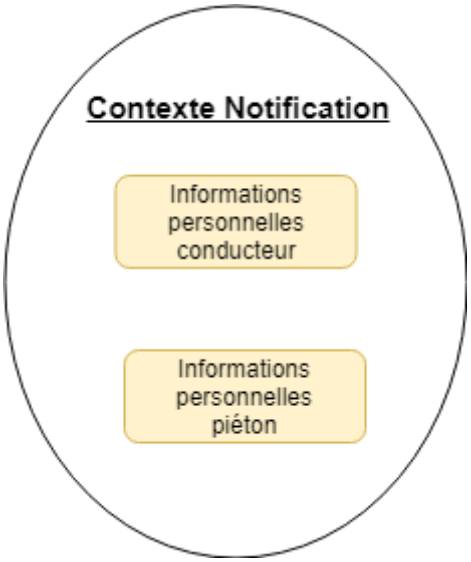
Contexte Piéton : Ce contexte prend en charge les données liées à un piéton ses informations personnelles et des informations liées à son trajet



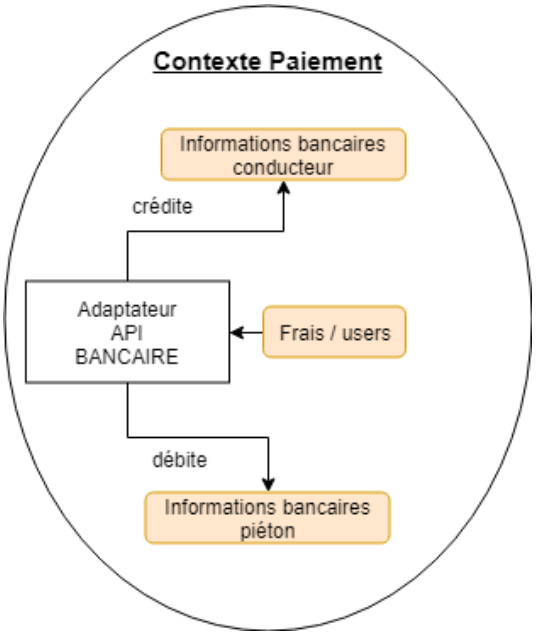
Contexte Voyage : ce contexte englobe le traitement principal de calcul du voyage en fonction des trajets du conducteur et du piéton. Il permet d'implémenter les algorithmes de Segment pour définir la probabilité de prise en charge et l'algorithme de match qui permet de valider la prise en charge et définir le nouvel itinéraire du conducteur. Il permet également de déduire les frais à partager quant au nouveau trajet.



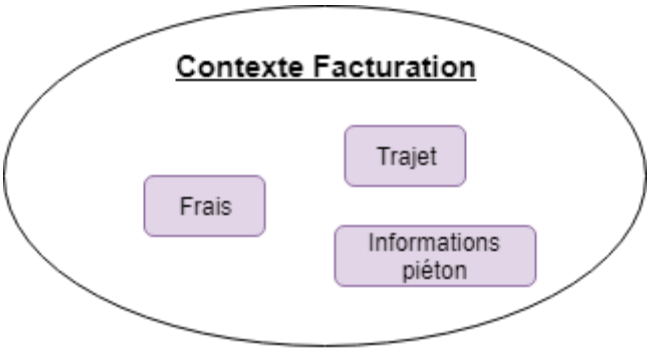
Contexte de Notification : Ce contexte se base sur les informations personnelles du piéton et du conducteur afin d'effectuer l'envoie des différents types de notifications nécessaires pour une bonne communication entre les deux utilisateurs. Exemple : confirmation de prise en charge, demande de regroupement au point de rendez-vous, connaissance du conducteur et informations sur le véhicule etc.



Contexte Paiement : le contexte de paiement permet en fonction des frais généré par le nouveau trajet et des informations de bancaires du piéton et du conducteur de réaliser les différentes transactions de débit/crédit de l'un vers l'autre.

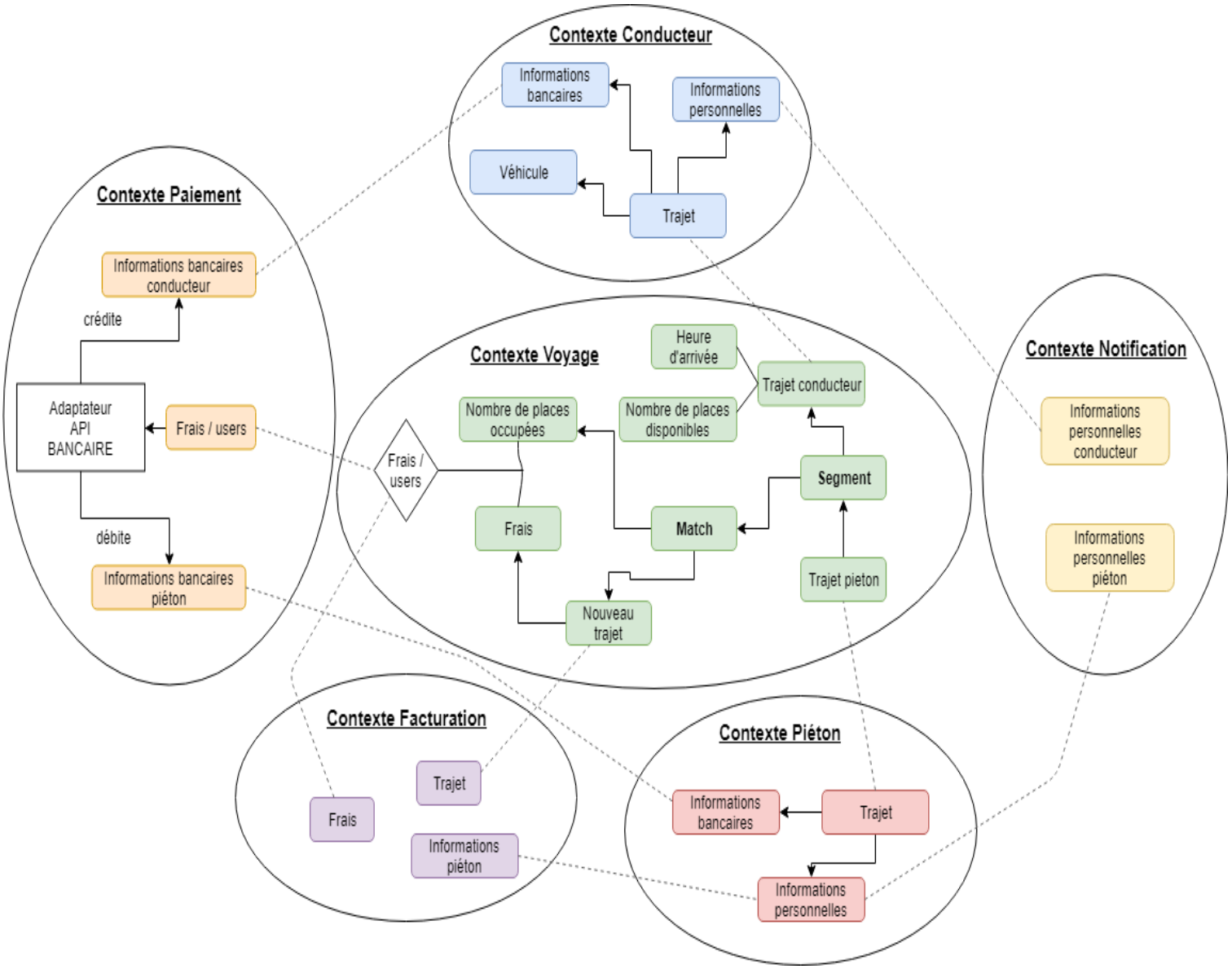


Contexte Facturation : Ce contexte permet de gérer un ensemble de données permettant de produire la facture à envoyer par mail au piéton

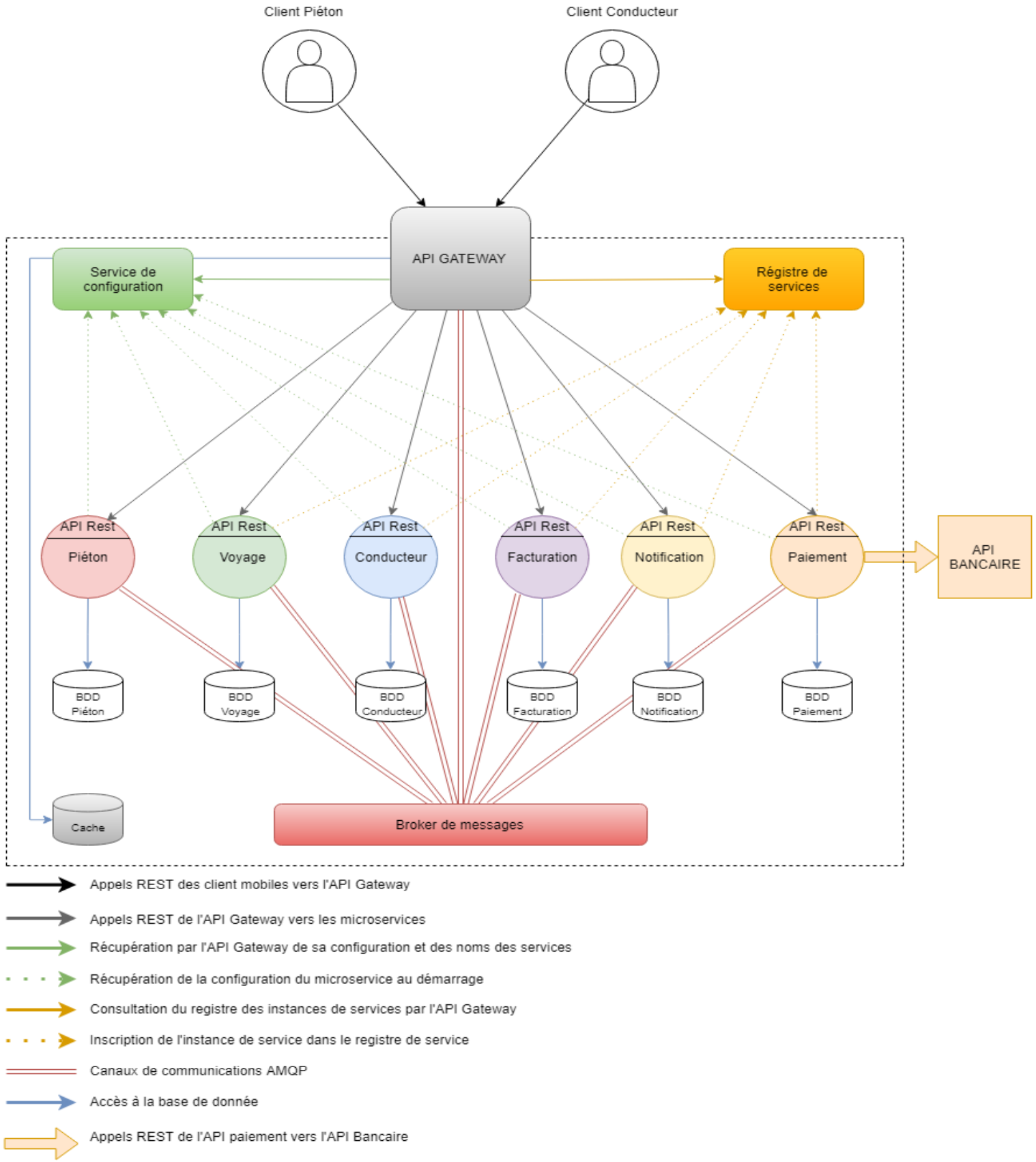


La carte de contextes

Maintenant que nous avons défini l'ensemble des contextes bornés de l'application, il est primordial de faire une cartographie de ces contextes pour avoir une vision d'ensemble des communications entre ces derniers et entres les différentes équipes qui en ont la prise en charge.

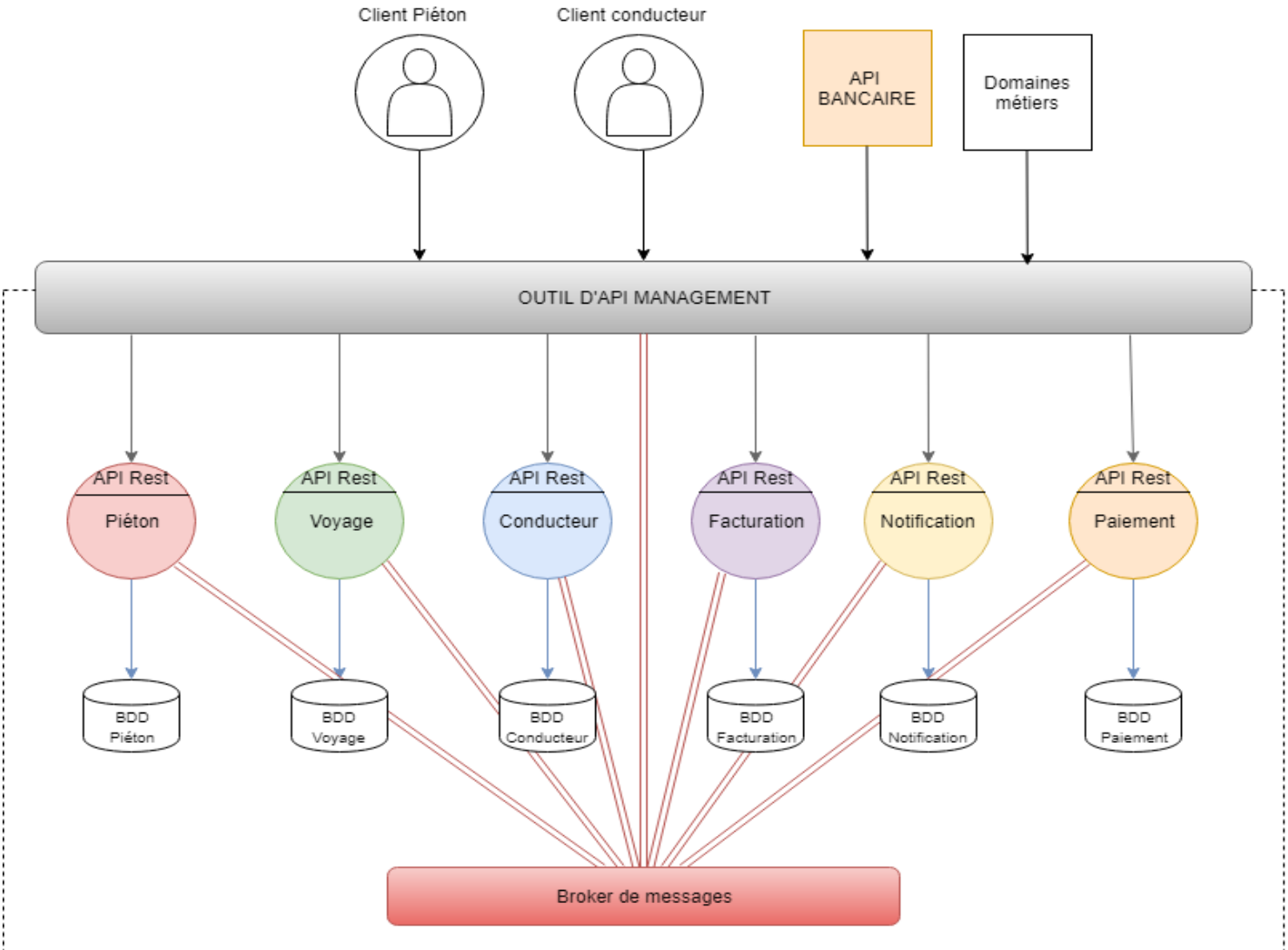


Architecture cible d'intégration :



Proposition d'une stack technologique	
Applications mobiles	IOS, Android
Applications web	Angular 7+
Microservices	Spring Boot 2, .NET
Service de Configuration	Spring cloud – config server
Registre de service	Cloud Discovery – Eureka server
API Gateway	NGINX plus
Cache	Redis
Broker de message	Rabbit MQ
Base de données	MongoDB, Cassandra

Architecture centrée autour d'un outil d'API management :

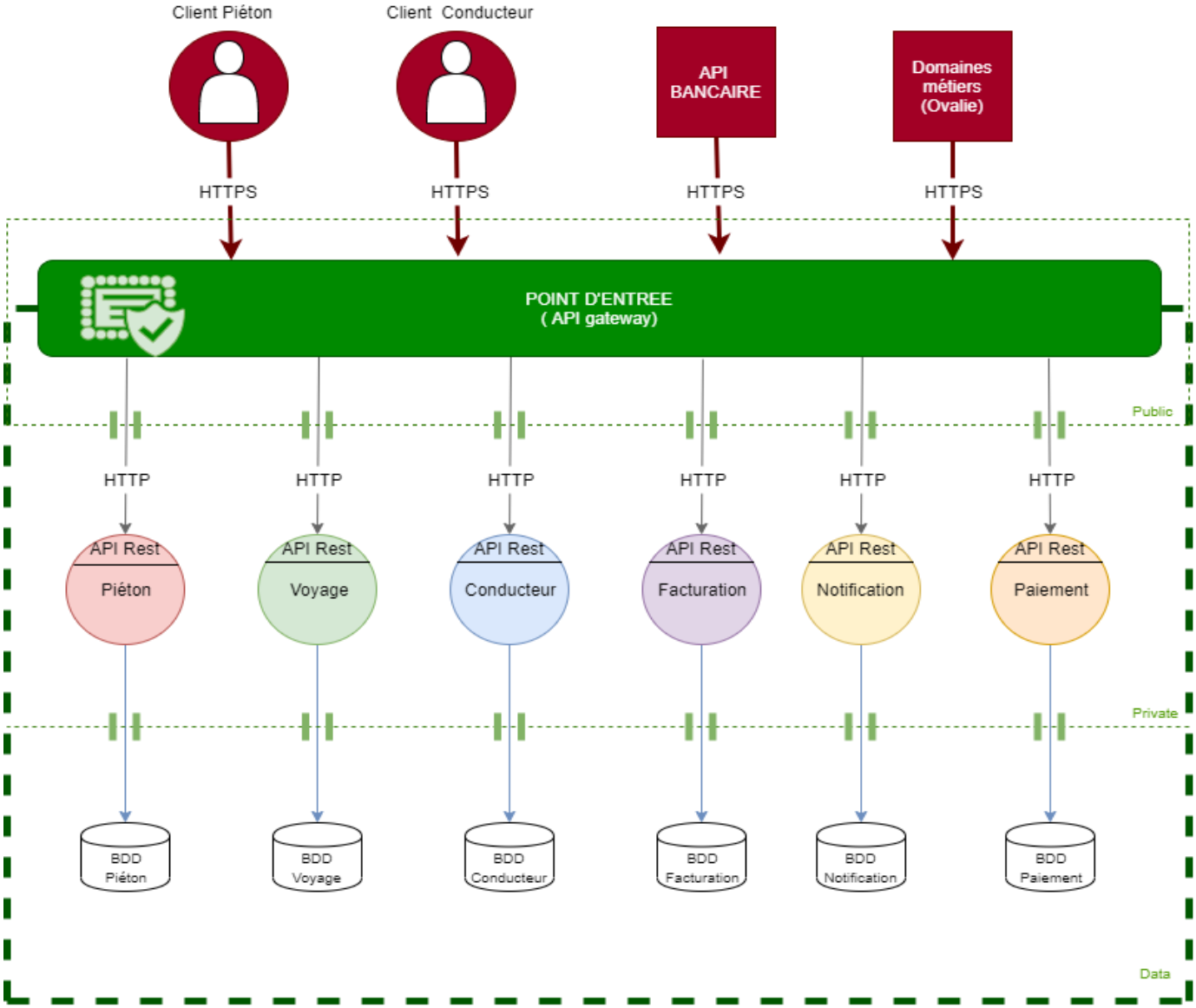


Benchmark de solutions :

Solution d'API Management	Intéressant pour	Taille de l'entreprise	Livraison	Plan de facturation par version	Portail de développeur
Apigee	Des outils de monétisation	PME, PMI	Proxy, Agent, Hybride	Evaluation : gratuite, Equipe : 500€/mois, Entreprise : 2500€/mois	Oui
3Scale	Portail des développeurs	Startups, PME, PMI	Proxy, Agent, Hybride	Pro : 750€/mois, Entreprise : Devis sur mesure	Oui
IBM API Management	User-friendly	Entreprise (Par taille)	Proxy, Agent	Lite : gratuite, Entreprise : 100€/ 100k d'appels API, Autres plans en fonctions de la taille de l'entreprise	Oui
Akana	Les outils de gestion du cycle de vie	Startup, Entreprise	Proxy, Agent, Hybride	Essai : gratuit, Startup : 4000€/mois, Entreprise : Devis sur mesure	Oui
Kong	API Gateway open source	Startup, PME, PMI, Entreprise	Proxy	Gratuite	Non

Implémentation de la sécurité :

CoMove est une application mobile basée sur une architecture de microservices, elle requiert de ce fait l'implémentation de certaines règles de sécurité nécessaires dans ce contexte, afin d'éviter les risques pouvant compromettre les utilisateurs de l'application et l'application elle-même :

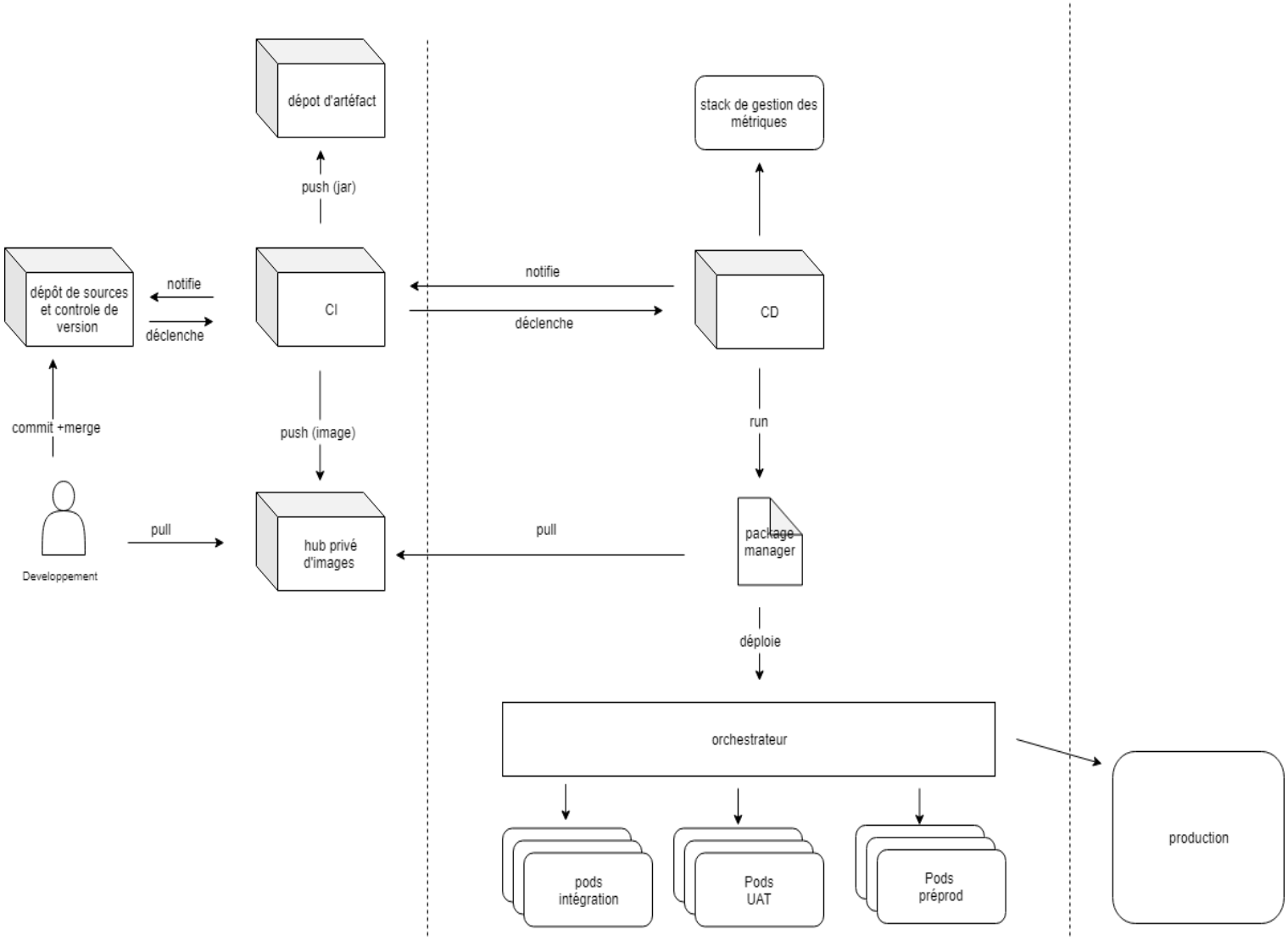


Public	Sous réseau accessible depuis l'extérieur
Private	Sous réseau qui communique avec Public et qui accède aux données
Data	Sous réseau dédié aux données
	Tunnels de communication entre les réseaux
	Terminaison TLS et solution permettant la rotation des certificats

Périmètre à sécuriser	Actions à implémenter
Infrastructure	- Sous réseau publique pour la gestion des échanges en l'extérieur et l'intérieur via l'API Gateway - Ajout d'une solution de gestion de rotation des certificats TLS - Sous Réseau privé dédiée aux microservices qui communiquer avec le réseau public et l'accès aux données - Sous réseau privé dédié aux bases de données pour la protection des données. - Terminaison SSL/TLS pour que tous les accès externes soient en HTTPS et interne en HTTP
La session des utilisateurs	- Encryptage des informations de l'utilisateur avec un UUID qui sera mappé à ses informations réelles dans l'API Gateway - Les données utilisateur restent dans le contexte global et ne transitent jamais à l'extérieur.
Les Microservices	- L'isolation des données : Chaque microservice ne peut accéder aux données gérées par un autre microservices qu'en passant par un appel API ou de façon asynchrone. - Les identifiants de base de données doivent tous être différents et d'un niveau de complexité élevé. - Le contrôle de propriété : vérifier que les données envoyées appartiennent au bon utilisateur (notion de confidentialité/rôles) - Tagger les ressources pour les tracker quand elles sont distribuées - Logger toutes les requêtes dans un système de log central (implémentation de la suite ELK) - Aucune information liée aux ressources ne doive être affichées dans les logs - Monitorer les services pour détecter les incidents et améliorer la performance
API Gateway	- Utiliser un corrélation-ID pour suivre les requêtes

	- Identifier les utilisateurs qui génèrent le plus de requêtes (prévention DDOS, Brute force login)
API externe Bancaire	- Implémenter l'authentification et l'autorisation JWT
API externe Ovalie	- Implémenter l'authentification et l'autorisation JWT
Le client Conducteur	- Implémenter l'authentification et l'autorisation JWT
Code source	- Respect des normes de développement et de quelques standards de sécurité liées aux développements - Bien faire ses configurations

Architecture cible de déploiement :



Proposition d'une stack technologique	
Dépôt de sources et contrôleur de version	Bitbucket
CI / CD	Jenkins
Hub privé d'image Docker	Nexus Sonatype
Dépôt d'artéfact	JFrog Artifactory
Package manager	HLEM
Orchestrateur d'images	Kubernetes
Stack de gestion des métriques	ELK (Logstack, Kibana,Elastique search)

6. Déploiement

Après un travail de conception fonctionnel à l'aide des techniques de conception DDD et l'identification des processus d'intercommunications permettant d'intégrer les services les uns aux autres et après avoir pris connaissance d'un ensemble de mesures de sécurité, il est donc nécessaire de définir une stratégie de déploiement.

En ce qui concerne ce dernier, l'intérêt des microservices se porte sur la capacité à déployer les services individuellement et fréquemment en minimisant l'interruption complète d'une l'application. L'automatisation du processus de déploiement est essentielle et vivement recommandée dès le début, même si l'architecture est composée de très peu de services.

Afin de conserver l'indépendance de chaque équipe, il est nécessaire que ces dernières aient leur propre chaine de production. Cela passe par la mise en place d'une chaine d'intégration continue et des mécanismes de livraison continue vers des environnements dédiés afin de gérer les livraisons fréquentes.

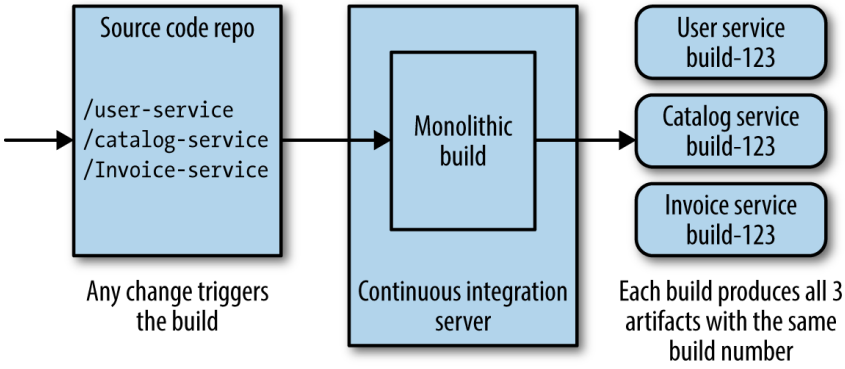
Le besoin d'automatisation du déploiement remonte à l'ère des applications monolithique, ce qui fait qu'à ce jour, le marché regorge de multiples solutions permettant d'implémenter ce processus. En admettant que la plupart des tests aient été couverts, voyons plus en détail l'intérêt qu'il y aurait à faire du CI/CD dans une application de microservices ainsi que les stratégies de déploiement possible.

6.1. L'intégration continue

Couramment, dans une application en monolithe, l'intégration continue (**CI : Continuous Integration**) consiste à exécuter tous les tests de l'application globale. Bien qu'on serait tenté de procéder de même pour une application découpée en microservices au tout début de son implémentation (car très peu de services), il faudrait cependant tenir compte du fait qu'il serait plus intéressant de tirer parti du découpage en microservices.

Analysons l'intégration continue dans les situations de build unique et de build par microservices :

- Build unique de l'application en microservices



*Source : « **Building Microservices** » Sam Newman, Figure 6.1*

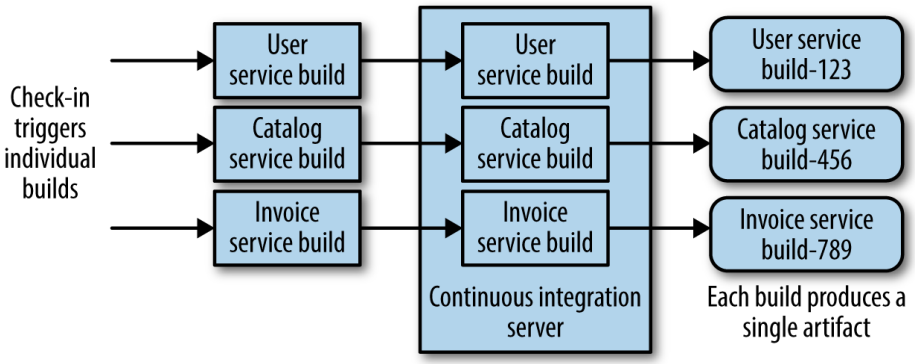
Présentation de l'utilisation d'un référentiel de code source unique et d'une génération de CI pour tous le microservices.

Cette approche consiste à builder l'application entière et de générer les différents artéfacts des microservices après chaque commit Il faut noter que chaque artéfact aura un même numéro de version. Cette information est particulièrement biaisée, car le numéro de version sur un artéfact n'indique pas forcement qu'il y'ait eu des modifications sur ce dernier.

L'avantage de cette approche est sa facilité de mise en œuvre dans la mesure où il ne suffira que d'installer une usine de build qui prendra en charge tous les services.

Cependant, si le nombre de services et de développeurs venait à s'accroitre, on pourrait être confronté à des problèmes de gestion et de latence de l'usine suite à un nombre fréquents de commit.

- Build par microservices



*Source : « **Building Microservices** » Sam Newman, Figure 6.3*

Présentation de l'utilisation d'un référentiel de code source et de la création d'un CI par microservice

Ici comme on peut le comprendre il s'agit d'implémenter **une usine de build par microservice** et non pour toute l'application. Cela rendrait les microservices indépendant les uns des autres en ce qui concerne leur livraison.

Cette approche est légèrement complexe à mettre en œuvre cependant elle présente certains avantages non négligeables en termes de finesse :

- Les tests sont exécutés de façon plus ciblée.
- Le fait de déployer un service à la fois et indépendamment des autres, qui ici est une caractéristique essentielle de cette architecture est ce qui permet de gagner du temps et de la disponibilité.
- Sachant qu'un microservice correspond à une équipe, cela responsabilise d'avantage cette dernière.

Cette approche est efficace quand les contextes fonctionnels sont clairement définis et stable. Bien qu'elle nécessite plus de travail fonctionnels et technique, elle répond bien à une stratégie basée sur la **performance** et le **time-to-market** et par-dessus tout à l'intérêt de mettre en œuvre cette architecture

Recommandations :

En effet, des microservices bien définis fonctionnellement apportent une certaine flexibilité en matière d'intégration continue. Il est tout à fait possible et il y a tout intérêt à créer une chaine d'intégration par microservice, afin d'effectuer et de valider rapidement un changement. Chaque microservice devrait avoir son propre référentiel de code source et déployer un seul artéfact.

L'alignement sur la propriété de l'équipe est également plus clair, si cette dernière prend en charge un microservice, elle devrait posséder un référentiel de construction, être capable d'effectuer des modifications sur ces référentiels.

Si tous les microservices sont définis, même s'ils ne sont pas encore réalisés, il faudrait planifier toutes les chaines de build pour chacun d'entre eux, au risque de se retrouver avec un re factoring plus couteux quand l'application prendra de l'ampleur.

6.2. La livraison continue

La livraison continue (**CD : Continuous Delivery**) est l'approche permettant d'obtenir un retour constant sur l'état de production de chaque enregistrement comme candidat à la publication sur un environnement donné. Elle s'appuie sur le concept de mise en place des **pipelines** qui sont des découpages de build en plusieurs étapes et regroupant des exécutions de tests sur certains critères (rapides, de petite tailles, lent, de grande tailles). Ce système va du constat qu'un cas très commun comme celui des tests peut ralentir un déploiement s'ils sont tous exécutés en même temps.

La procédure consiste à modéliser tous les processus nécessaires pour faire passer les sources logicielles de l'enregistrement à la production et savoir où une version donnée du logiciel est en cours de validation. En livraison continue cela se fait en implémentant des pipelines de build à plusieurs étapes de façon très automatisée.

Focus sur les pipelines

Un pipeline de déploiement en livraison continue est un circuit configurable que parcourra les changements apportés au code à partir du commit jusqu'à la mise en production.

L'objectif de cette méthode est de gagner en performance sur la livraison des services en **ciblant les tests qui vont être (ou pas) exécutés de façon continue**. En effet, certains tests qui impliquent plusieurs services ou qui sont basés sur des **workflows complexes** peuvent être long à exécuter. Afin qu'un build ne soit pas pénalisé par ces derniers, il faudrait construire des pipelines de déploiement entre les tests s'exécutant rapidement pour un feedback rapide et ceux qui sont long et spécifiques.

Sur une application donnée, on peut configurer au minimum les 2 types de pipelines suivants :

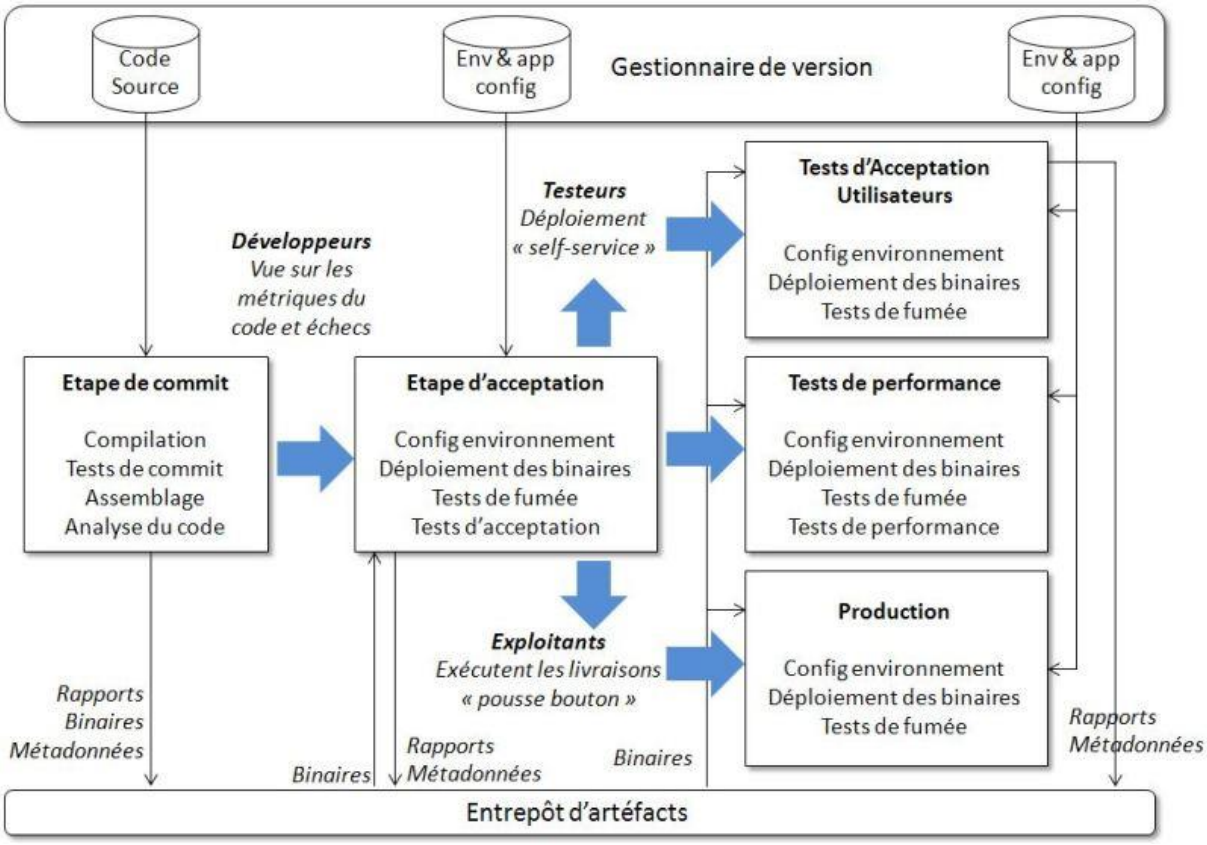
Pipeline 1 : Build de tests rapides

- Les tests sont exécutés à chaque commit
- Le feedback d'exécution des tests sont considérés et prise en charge à l'instant

Pipeline 2 : Build de tests longs

- Les tests ne sont pas exécutés après chaque commit
- Les tests sont exécutés à un moment différé

On pourrait aussi envisager de créer une tache dédiée à leur exécution et la prise en charge de leur feedback.



Source : « Continuous Delivery » de Jez Humble et David Farley

Dans cette présentation Jez Humble et David Farley qui ont publié d'importants travaux sur la culture DevOps et l'intégration continue/livraison continue proposent un modèle pour montrer comment placer stratégiquement les pipelines dans la chaine afin d'obtenir des informations sur la visibilité, avoir du feedback et où intervenir en cas d'erreur.

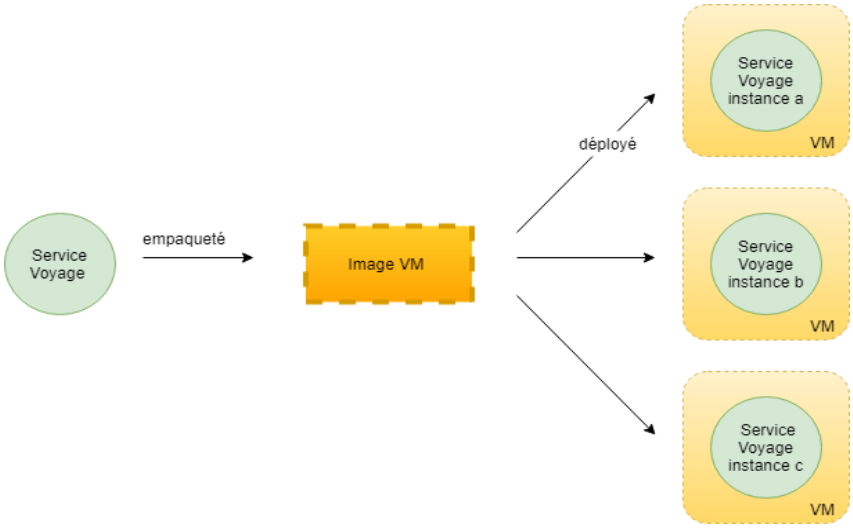
6.3. Les stratégies de déploiement

Il existe sur le marché, plusieurs outils permettant de mettre en place une stratégie de déploiement efficace. Pour une architecture de microservices, la stratégie pour gérer un grand nombre d'hôtes consistera à trouver des moyens de diviser des machines physiques en machines plus petites. En allant du principe que nous allons déployer une instance de service par hôte, on peut déjà dire que la virtualisation traditionnelle telle que VMWare ou virtualisation moderne telle que celle proposée par

AWS a apporté d'énormes avantages en réduisant les couts de gestion de hôtes. Cependant de nouveaux progrès dans cet espace, tel que les conteneurs Linux méritent d'être explorés, car ils ouvrent des possibilités encore plus intéressantes pour traiter les architectures de microservices.

La virtualisation

Dans une stratégie consistant à déployer un microservice à l'aide de la virtualisation, chaque service devra être empaqueté en tant qu'**image VM** et chaque instance de ce service sera exécuté dans une machine virtuelle lancée à l'aide de cette image VM :



Ce procédé a été largement popularisé par AWS avec la solution **Amazon Elastic Compute Cloud (EC2),** qui permet de créer une VM image appelée EC2 AMI et la plateforme AWS créera des instances EC2 sur le cloud conformément aux besoins de l'application. Plusieurs compagnies telles que **Boxfuse** ou encore **CloudNative** proposent des solutions permettant de créer des image EC2 et des solutions de packaging de services ou autres projets vers EC2 telles que **Aminator** et **Packer** sont de plus en plus répandues.

L'avantage du déploiement par VM est que chaque instance est encapsulée (peu importe la technologie) et exécutée de manière isolée (les valeurs de CPU et de mémoire sont fixes et l'instance de service ne peut pas voler des ressources aux autres). Mais un autre avantage à apprécier c'est la capacité à exploiter des infrastructures cloud très mature comme AWS, qui fournit un large éventail de services prêt à l'emploi tel que le load-balancing ou encore l'autoscaling, le déploiement est simple et fiable.

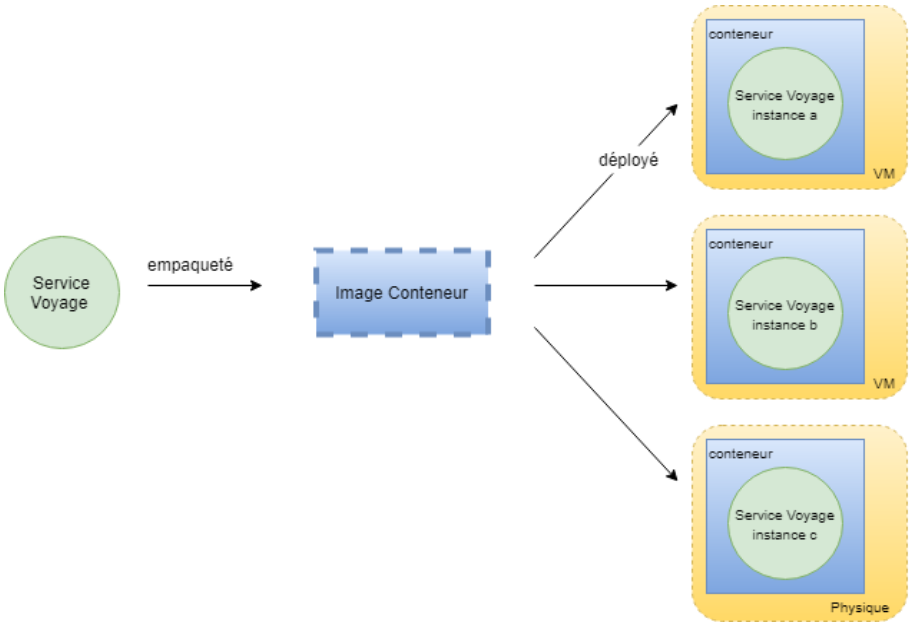
Toutefois, l'inconvénient de cette approche est que l'utilisation des ressources n'est pas tout à fait optimisée, en effet chaque instance de service est surchargée par le système d'exploitation de la VM,

ce qui pourrait impacter son démarrage. De plus, les IaaS facturent les VM de tailles fixes, qu'elles soient occupées ou non, ainsi il est plutôt fréquent d'avoir des VM sous-utilisées engendrant des couts de déploiement élevés. Autre constat, les déploiements des nouvelles versions de services sont lents du fait de la lourdeur des images VM et du démarrage de leur système d'exploitation. Mais il y'a une forte tendance dans la recherche pour alléger ce processus.

La conteneurisation

Les conteneurs sont le plus souvent présentés par opposition aux systèmes de virtualisation standards afin de démontrer tous les bénéfices que ceux-ci peuvent apporter en termes de rapidité et de limitation des couts. Car contrairement à une VM qui exécute une copie complète d'un système d'exploitation et une copie virtuelle de tout le hardware dont le système a besoin pour fonctionner (ce qui augmente le nombre de cycles de RAM et CPU qui entraine les problèmes de latence évoqués dans le point précédent), un conteneur utilise un système d'exploitation, des programmes, des bibliothèques compatibles et des ressources système pour virtualiser des composants.

Dans cette stratégie de déploiement, chaque service devrait être empaqueté en tant qu'image de conteneur avec les applications et fichiers nécessaires pour exécuter le service. Habituellement un conteneur utilise le minimum pour exécuter un service, la taille d'une image de conteneur est beaucoup plus limitée que celle d'une image de VM, il devient alors possible de créer plusieurs conteneurs en provenance de cette image sur un hôte (physique ou virtuel) :



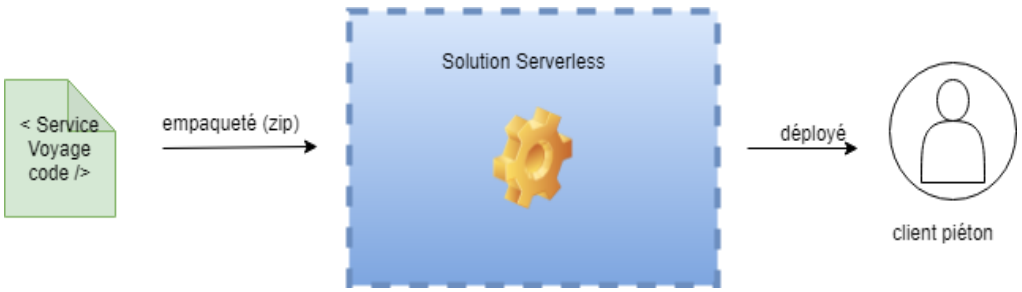
La solution de conteneurisation la plus populaire reste **Docker** suivi de près par **Solaris**. Habituellement plusieurs conteneurs peuvent être déployés sur un hôte, il faudrait donc utiliser un gestionnaire de cluster tel que **Kubernetes** ou **Marathon** pour gérer les conteneurs. Celui-ci sera en charge de décider où placer les conteneurs en fonction des ressources requises par ces derniers et celles disponibles sur les hôtes.

Les bénéfices des conteneurs sont quelque peu similaires à ceux des VM, notamment en ce qui concerne la problématique d'isolation des services les uns des autres. Il est plus facile de surveiller les ressources consommées par chaque conteneur. Les conteneurs encapsulent aussi la technologie utilisée pour mettre en œuvre un service. Cependant leur technologie est beaucoup plus légère et les images de conteneur sont beaucoup plus rapide à construire. Cet avantage est particulièrement apprécié quand on doit mettre à jour les services de façon régulière.

Cependant, il a été reproché aux technologies de conteneurisation, bien qu'étant en plein essor, de manquer de maturité et par conséquent de **sécurité**, notamment du fait que plusieurs conteneurs partagent le noyau d'un même système d'exploitation (point of Failure). Il faut aussi noter que les conteneurs ont tendance à bloquer les ressources serveur qui leurs sont allouées et cela peut également impacter des coûts d'exploitation. Un autre inconvénient vient aussi de la responsabilité à administrer soi-même les conteneurs si l'on décide de ne pas s'orienter vers des solutions de conteneurs hébergés comme **Google Container Engine** ou encore Amazon **EC2 Container Service**, ce qui peut s'avérer fastidieux à grande échelle.

Le Serverless

Le Serverless est une architecture sans serveur regroupant les principes de type **BaaS** et **FaaS** (Backend as a Service et Function as a Service). L'idée est d'héberger le code sur une architecture qui serait offerte comme un service, par conséquent qu'il n'est pas nécessaire de maitriser. On accède à une console de la solution où on téléverse le code zippé, on ajoute quelque Meta data (nom du service, langage et version utilisée, etc.). Le code est ensuite exécuté, se connecte aux différents services dont il a besoin (API, base de données, bucket de fichiers statiques, etc.) et se déploie automatiquement :



Le principe du Serveless est d'externaliser complètement le déploiement et de ne payer que la consommation des services (bande passante et espace de stockage). Grace à l'élasticité, les couts d'exploitations sont optimisés et il n'est pas nécessaire de gérer l'infrastructure et sa disponibilité.

La solution la plus connue actuellement sur le marché est AWS Lambda. Les fonctions lambda qui encapsulent le code profitent de la force des solutions AWS pour fournir ce service. Google et Azure offrent aussi ce type de solution.

L'avantage certain est l'optimisation des coûts et des ressources, ce que l'on ne retrouvait pas systématiquement avec la virtualisation et la conteneurisation. Mais contrairement à ce que l'on pourrait penser, bien que tout soit externalisé et qu'il ne faille que se concentrer sur l'écriture du code, la prise en main de ce type de solution nécessite un monté en compétence qui a un cout et une veille de la solution pour avoir connaissance des différentes mises à jour et nouveautés. Cela tient du fait qu'il s'agisse d'une nouvelle technologie, pas assez mature et que l'on pourrait considérer comme nouvelle étape dans la culture DevOps.

7. Le monitoring

Le monitoring reste un élément essentiel de la gestion des systèmes d'informations et les défis associés à la surveillance des microservices sont particulièrement uniques. Les applications basées sur une architecture de microservices ont des exigences de surveillances différentes et plus intensives que celles des applications traditionnelles de type monolithique. Cela tient du fait qu'une logique métier peut être répartie sur plusieurs services distincts, que ces services sont amenés à être distribués, répliqués et portés par le cloud.

Pourquoi monitorer les microservices ?

La première raison pour laquelle on surveille un système distribué c'est parce que tous les systèmes échouent ! Une architecture de microservices est tolérante aux pannes, cela signifie qu'au sein du système, il existe des services en échec qui n'ont cependant pas d'effet de cascade sur tout le système. Il est donc important de connaitre les raisons de ces échecs, d'apporter une analyse et de trouver une solution.

Une autre raison particulièrement importante découlant de la première est la performance du système. En effet, les données liées à la performance apportent des informations sur les états de dégradations du système. Si ces états ne sont pas pris en charge, il en résultera inévitablement des échecs. Surveiller la performance du système permet ainsi de prévenir l'échec de ce dernier.

Mais la raison pour laquelle on associe indubitablement la performance au monitoring, c'est que l'ensemble des services fournis (en interne ou à l'externe), le sont dans le cadre d'un contrat de niveau de service (SLA). Sans monitoring, il est impossible de savoir si le SLA est respecté ou violé et sans surveillance de la performance il est difficile d'anticiper les défaillances qui pourraient à terme faire perdre des contrats de services, voire pire, générer des attaques en justice. Le monitoring de la performance des applications (APM) constitue un marché à part entière, sur lequel on peut retrouver un panel intéressant d'outils open source et commerciaux, permettant de publier, rassembler et stocker des données de métriques :

Figure 1. Magic Quadrant for Application Performance Monitoring

Source: Gartner (March 2019)

Source : www.gartner.com

Classement mondial des meilleures solutions de monitoring de la performance des applications

Comment déterminer les métriques à monitorer dans une architecture de microservices ?

Les grands systèmes distribués traitent chaque jour d'un grand volume de données, générant ainsi des giga-octets de nouvelles métadonnées sur leur comportement. On distingue cependant quelques groupes de métriques :

- **Les métriques d'application** : elles sont spécifiques à l'application et permettent donc de remonter des informations liées au comportement de cette dernière. Exemple : le nombre d'utilisateurs connectés à l'application à un instant T ; les id de ceux qui commandés un produit X, le produit Y dont la vente génère 1000 transactions par heure et que dans l'heure qui suit le nombre de transactions chute à 100 (cela pourrait par exemple lever une alerte) etc.
- **Les métriques de plateforme** : elles sont spécifiques à l'état de l'infrastructure et sont généralement retransmises sur un tableau de bord analytique des performances

dégradées susceptibles d'affecter la débit et/ou la défaillance du système. Exemple : le nombre de requêtes traitées par un service par minutes ; le taux de réussite / échec pour un service, une attaque DDoS.

- Les métriques basées sur les évènements externes aux systèmes : elles impliquent l'ensemble des informations à répertorier quand serait survenue une force externe agissant sur le système. Dans le cadre des microservices, il s'agit généralement des déploiements fréquents qui peuvent être de potentiel suspect en cas de dysfonctionnement du système après que l'un d'entre eux n'ait été effectué. Le travail ici consistera plus à journaliser ces évènements afin d'analyser des corrélations avec l'état du système. Exemple : déploiement après ajout de nouveau code, de correctifs de code, d'ajout de configuration.

Le type de métriques à monitorer sera donc en fonction de chacun de ces aspects et aidera aux choix des outils permettant d'implémenter cela.

Quelques solutions et outils de monitoring

Les technologies de monitoring se scindent habituellement en deux catégories :

- Les librairies : qui sont intégrées à une application au moment de son développement. Ces ressources peuvent être utilisée pour faire de la journalisation et améliorer les rapports de métriques
- Les plateformes : qui se concentrent sur la collecte et l'analyse des données d'applications et d'infrastructures.

Les solutions commerciales sont généralement un mix ces deux catégories. On peut également trouver des solutions open-source qui pourraient entrer dans une stratégie de mise en place du monitoring :

Solutions	Type de métriques	commentaires
ELK (Elastic Search, Logstach, Kibana)	Application	Permet de parser des logs, les stocker sur un serveur, les indexer afin d'y effectuer de la recherche
Zipkin	Application, Infrastructure	Permet de tracer les appels entre microservices et de remonter des problèmes de latence
Grafana	Application, Infrastructure	Outil de visualisation web utilisé pour créer des tableaux de bord visuel quand il est connecté à une base de données de métriques
Prometheus	Infrastructure	Outil de surveillance des évènements, qui enregistre les métriques en temps réel dans une base de données. Peux être utilisé avec Grafana et d'autres outils de visualisation
Azure Monitor	Infrastructure	Collecte analyse et affiche des données sur la santé des clusters.
Dynatrace	Application, Infrastructure, évènements	Solution commerciale complète permettant de remonter une grande variété d'informations sur tout le système, de façon automatique, avec des résultats d'analyse remonté par de l'Intelligence Artificielle. Numéro 1 mondial du monitoring

Pour ne citer que ceux-là.

En somme :

Les exigences de monitoring doivent être prises en compte dès le début du cycle de vie d'une application. La surveillance des systèmes nécessite des contributions du développement et des

opérations. C’est une partie essentielle du support opérationnel de tout système distribué. Les architectures de microservices sont encore plus distribuées que les applications monolithiques typiques. Elles nécessitent plus d'attention en temps réel et une surveillance proactive.

Il est tout aussi important de collecter des données pertinentes que d'analyser les données collectées. Il devient nécessaire que les développeurs soient initiés à la culture DevOps car ces derniers sont au début de la chaine et doivent implémenter des pratiques de capture de métriques directement dans le code des applications pour signaler les événements spécifiques à chaque application. Les équipes opérationnelles doivent collecter des données non seulement à partir d'applications, mais également des plates-formes et des systèmes de déploiement associés.

Des solutions open source et payantes sont disponibles pour prendre en charge à la fois la publication et le stockage des événements de surveillance. Ces données sont essentielles pour prendre en charge un système distribué résilient, fiable et disponible.

8. Transformer un monolithe en microservices

Bon nombre des organisations actuelles ont des systèmes bâtis sur une architecture monolithique. L'avantage certain de ces derniers est qu'ils génèrent des revenus pour les entreprises qui les possèdent. Mais à la longue ces systèmes sont de plus en plus couteux :

- Les échecs et les pannes sont de plus en plus fréquents
- Les fonctionnalités tardent à être livrées
- Il est difficile d'inclure de nouvelles technologies qui pourraient s'avérer bénéfiques
- Le processus d'introduction de nouveaux employés prend de plus en plus de temps
- Les employés existants commencent à envisager un cheminement de carrière en dehors de l'organisation. Etc.

Ces symptômes, pour ne citer que ceux-là, indiquent que l'architecture du système a cessé de répondre aux exigences de la société. Il y a de plus en plus d'organisation qui ont lancés des processus de « rénovation digitale » de leurs systèmes. Une importante question reste donc de savoir, comment migrer d'une application monolithique vers une application de microservices ?

Voici quelques éléments de réponse avec un approche « étapes par étapes »

Etape 1 : Bien définir les raisons qui poussent à la migration

La mise en œuvre d'une telle démarche constitue un défi important et nécessite des investissements. Il faut s'assurer que les démarches sont motivées et permettent de répondre positivement à la question de savoir s'il faille une architecture de microservices.

Quelques questions à se poser :

- Quels sont les objectifs, les priorités et les besoins de l'entreprise ?
- Est-ce que le système actuel est éprouvé sur le marché ?
- Y' a-t-il eu une évaluation de l'existant ?
- Les ressources compétentes existent-elles en interne ? Les collaborateurs sont -ils prêts ?
- Etc.

En général on pourrait citer 4 raisons principales pour lesquelles la migration devrait être amorcée :

a- Une partie du système va opérer de grands changements (refactoring structurant, rénovation digitale, etc.)
b- La structure de l'équipe change (multi office, multi régionale, internationale, etc.)
c- Des problématiques de sécurité (échecs, attaques, pertes de données, etc.)
d- L'obsolescence de la technologie en cours (désuète, non maintenue par les concepteurs, ressources compétentes de plus en plus rare, etc.)

Etape 2 : Définir la stratégie de conception en sous-domaines

Selon Martin Fowler, toutes les personnes qui ont commencé à développer des microservices dès le départ, ce sont heurté à des difficultés car ils n'ont pas su découper l'application aux bons endroits et ont engagés des refactoring massifs. Donc, il serait plus simple de partir d'un monolithe vers une architecture de microservices car on est en mesure de savoir comment découper le système vu que l'on sait déjà où sont les problèmes, autrement dit les couplages.

Comme préconisé en début de ce document, une première stratégie serait d'implémenter des techniques de Domain-Driven-Design pour définir les différents contextes bornés, structurer les équipes et établir la carte des contextes.

Il serait aussi intéressant d'utiliser des Seams. Dans son livre « Working Effectively With Legacy Code », Michael Feathers, pionner et chef de file du mouvement Agile, définit le concept de Seams comme un « Morceau de Code pouvant être isolé et sur lequel on peut travailler sans impact sur le reste de la codebase ». En effet, dans une stratégie de migration, ces éléments sont très vite connus, et il s'avère que l'extraction des seams est une approche plutôt populaire, le but est donc d'identifier les seams qui pourraient devenir des microservices ou des APIs. Ce travail nécessite une collaboration entre l'équipe de développement qui a une vision limitée à son domaine de développement et l'équipe d'architecture qui a une vision un peu plus globale du système et qui validera que les seams sont bien des seams dans la mesure où leur extraction n'aura aucun impact sur le reste du système. Le code restant nécessitera un travail de conception en sous-domaine en fonction du volume restant.

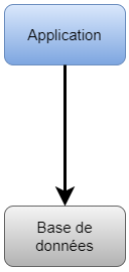
Etape 3 : Procéder par itérations

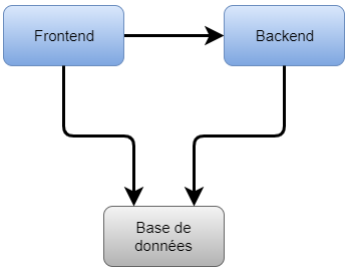
Qu'il s'agisse de construire une application en microservices correspondant à un monolithe en partant de zéro ou en démantelant graduellement le monolithe, il faut penser à procéder par itération à l'aide des techniques d'agilité.

Se donner des objectifs macro que sont les Milestones et les découper en objectifs micro que sont les Sprints et essayer de tenir la feuille de route.

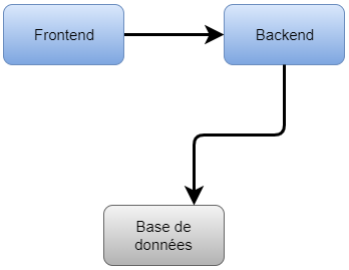
Etape 4 : Stratégie de transformation graduelle

Admettons que la conception en sous domaine ait été préalablement effectuée et partons d'un exemple général pour illustrer la transformation : Soit une application monolithique classique :

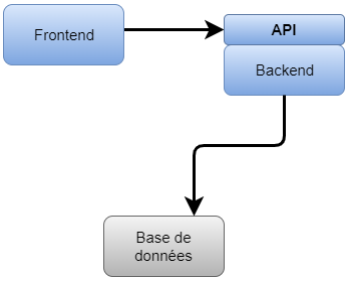




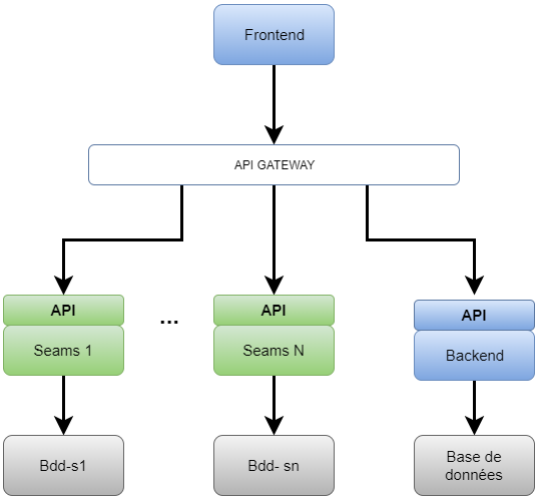
1- Séparer le Frontend du Backend de façon partielle : Le traitement des commandes de la logique applicative est délégué à la couche Service et les requêtes prenant en charge les vues sont dirigées vers la base de données. A ce stade aucune modification de la base de données ne doit être effectuée.



2- Séparer le Frontend et le Backend de façon logique : Mettre en place une couche Controller dans le Backend qui regroupera tous les points d'entrées dont a besoin de Frontend pour accéder à la base de données via des services.



3- Séparation physique du Frontend et du Backend : Créer une API pour le Backend et établir la communication avec Frontend via l'API

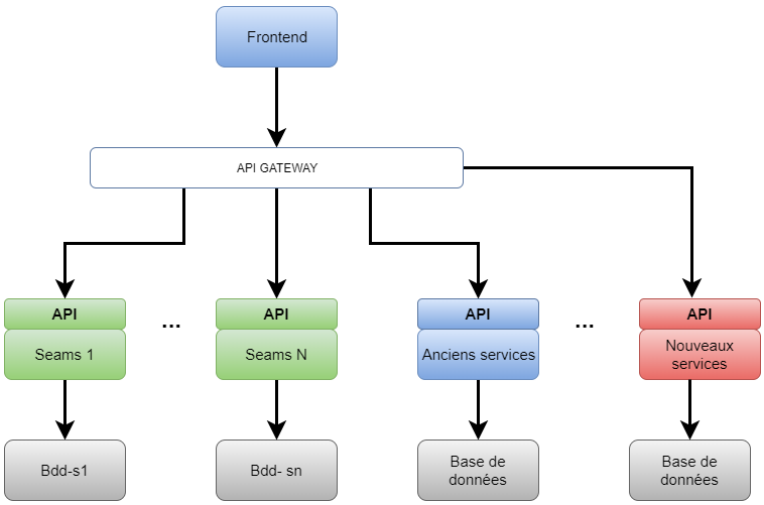


4- Extraction des seams dans le Backend : Créer des API pour tous les services ayant été identifié comme naturellement isolés et sans impacts et mettre en place l'API Gateway.

5- Pour le reste de l'application, implémentation du « Strangler Pattern »

Le pattern Strangler est un modèle de conception populaire permettant de transformer progressivement une application monolithique en microservices en remplaçant une fonctionnalité particulière par un nouveau service. Quand la nouvelle fonctionnalité est prête, l'ancien composant est « étranglé », autrement dit hors service. Tout nouveau développement est effectué dans le cadre du nouveau service et surtout pas dans le monolithe, les deux fonctionnalités coexistent jusqu'à ce que le nouveau service réalise entièrement la tache de l'ancien. Pour implémenter ce modèle, il faut itérer en 3 temps :

On devrait avoir la configuration ci-dessous en cours de migration :



En somme,

Une fois la migration terminée, il est utile de vérifier si les résultats attendus ont été obtenu en introduisant des procédés de monitoring (voir chapitre 7) afin de mesurer la performance, le temps moyen pour localiser et éliminer les défaillances, les temps de réponses etc… afin d'évaluer l'effet de la transformation. Le processus de migration d'un monolithe vers des microservices demande un investissement qui devrait apporter des 1ers résultats avant la fin du processus complet.

Conclusion générale

Tout au long de ce document nous avons commencé par comprendre pourquoi les architectures cloud natives ont émergées durant la dernière décennie et en quoi elles constituent la signature d'une nouvelle ère digitale centrée autour des technologies du cloud. Due à un intérêt fort de la part des organisations traditionnelles à se tourner vers cette nouvelle architecture, il était nécessaire d'explorer les concepts de microservices et d'API qui sont au cœur de cette architecture.

En partant du principe qu'on serait en train de monter cette architecture de zéro, la première phase a consisté à mettre l'accent sur la nécessité à définir correctement le découpage fonctionnel du système à l'aide de techniques de Domain Driven Design. A la suite de quoi, nous avons vu quelques éléments essentiels pour permettre la communication entre les microservices, tel que les API RESTful, le choix du JSON comme format d'échange, l'importance d'une API Gateway dans une ère digital très axée Mobile-First, les notions de synchronisme et d'asynchronisme, l'importance de la chorégraphie par rapport à l'orchestration en ce qui concerne les enchainements des appels de services et le Service -Discovery Pattern. A ce stade, on a constaté qu'une bonne pratique, si ce n'est la plus importante est de bien regrouper les microservices et d'utiliser une API comme interface de communication. Dans un souci de gouvernance, il était important d'introduire la notion d'API management afin de se doter des outils permettant de gérer efficacement un système découpé en APIs.

Les données ayant une valeur économique et leur manipulation étant régie par un ensemble de règles strictes de non-divulgation, un point sur lequel il a été important de mettre l'accent a été celui de la sécurité. Nous avons vu comment l'accès aux microservices devait être authentifié et autorisé, comment les communications entre microservices devaient être sécurisées, les restrictions d'accès aux APIs devaient être mise en place et comment implémenter la traçabilité des opérations. Mais ceci n'était que des informations d'entrée dans le vaste domaine de la sécurité, raison pour laquelle, une liste de 10 risques les plus fréquents a été révélée afin d'aider à rechercher un ensemble de mesure sécuritaires pour le système.

Afin de permettre une bonne compréhension des notions introduites jusque-là, une étude de cas d'une application mobile basée sur une architecture de microservices, a été proposée. On a pu voir conformément aux exigences fonctionnelles et non-fonctionnelles, comment concevoir le système en établissant une carte des contextes bornés (technique de DDD). Une architecture cible d'intégration (d'intercommunication) en interne et avec des systèmes externes a été proposée ainsi qu'un choix de technologies divers et varié permettant de montrer qu'il est tout à fait possible et recommandé de choisir pour l'implémentation d'une API, la technologie la plus à même de répondre au besoin

fonctionnel. Nous avons également vu comment cette architecture a mutée avec la mise en place d'une brique d'API management, qui permet la gestion des API, l'APIsation (API economy) du système et l'imbrication avec des systèmes tiers. Un choix technologique succinct de solution d'API management pouvant convenir à tout type de budget a été soumis. Une implémentation de la sécurité à la carte, a été proposée pour l'application.

Une architecture cible de déploiement a été proposée afin de permettre la compréhension du fonctionnement d'une chaine d'intégration et de livraison continue. Un choix de technologies a été proposé, pour permettre la création d'une solution maison. Mais cette architecture, proposée à titre informatif a permis d'introduire les notions d'intégration et de livraison continue, les stratégies de déploiement et solutions de CI/CD de plus en plus florissante sur le marché.

Nous avons également abordé l'importance du monitoring des architectures basées sur les microservices, car bien qu'étant tolérantes aux pannes, il est important de récolter les informations sur les pannes survenues et les résoudre afin d'améliorer la performance du système sur les tous les aspects architecturaux (entreprise, logiciel et infrastructure). Une analyse de quelques solutions de monitoring disponible sur le marché a été proposée.

Après avoir consulté un document comme celui-ci, les organisations se demandent comment transformer leurs systèmes, qui sont en grande partie des monolithes, en architecture de microservices afin de pouvoir bénéficier de tous les avantages qu'apportent le cloud. Le dernier chapitre propose une stratégie en plusieurs étapes : déterminer les motivations de la migration, utiliser les techniques de DDD, mais également les Seams (morceaux de code isolés dont l'extraction a un impact minime sur le système), procéder par itération à l'aide des techniques d'agilité et faire une transformation graduelle en introduisant Strangler Pattern (exemple à l'appui).

Ci-dessous, un tableau synthétique des notions introduites dans ce documents et les réponses apportées :

Notions	Questionnement	Eléments de réponses
Contexte	Pourquoi faire des applications cloud natives ?	• Solutions apportées contre les difficultés rencontrées par les géants du Web. • Besoin de rapidité pour être market-ready • Besoin de sécurité • Nécessité d'une mise à l'échelle horizontale • Pour les applications centrées sur le mobile • Par souci d'innovation en utilisant les dernières technologies • Bénéficier de l'API Economy qui est un nouveau modèle économique basé sur la monétisation de l'accès aux ressources de son système.
Architectures de microservices		
Microservice, API, API Management	De quoi s'agit-il ?	• Microservice : service métier regroupant un ensemble de service technique de manipulation des ressources • API : Interface de programmation permettant d'accéder à un ou plusieurs microservices liés par une logique métier • API management : gestion de toutes les APIs constituant le système et de celles externes aux systèmes et qui

		apportent de la complétude au service fourni par le système.
Conception des microservices	Quelle est la recommandation pour une MSA ?	• Prendre connaissance des techniques de Domain-Driven-Design pour savoir comment découper un système • Insister sur les notions de contexte borné qui est un cadre logique et limité dans lequel un modèle fonctionnel devra évoluer • Etablir une carte des contextes afin de voir comment ces derniers communiquent les uns avec les autres pour donner le modèle global
Communication entre les microservices	Quelles sont les bonnes pratiques ?	• Garder les APIs « technology-agnostic » • Rendre les services simples pour les consommateurs • Masquer les détails d'implémentations • Préférer le HTTP comme protocole de communication • Faire des API RESTFul (de niveau 3 avec le contrôle Hypermédia) • Préférer le JSON comme format d'échange • Faire de la veille sur gRPC et protobuf • Utiliser systématiquement une API Gateway lors d'une approche Mobile-first • Déterminer les besoins de synchronisme et d'asynchronisme

		pour implémenter les solutions adéquates • Toujours favoriser les enchainements d'appels en chorégraphie au profit de ceux en orchestration • Implémenter systématiquement le Service Discovery afin de retrouver les services quand ils seront distribués sur le Cloud.
API Management		
API publiques et privées	Quelle différence ?	• API publique : ouvertes à tous et conçues pour être accessible par une large communauté de développeurs • API privées internes : utilisées et exposées à l'interne d'une organisation. • API privée partenaire : Elles peuvent être ouverte à un autre système dans le cadre d'une stratégie économique (B2B, API Economy)
Outil d'api management	Quels sont les services principaux ?	• L'API Gateway • Le service de routage : mappage d'URL, distribution de service, regroupement des connexions, load-balancing, orchestration des services • Contrôle et gestion des version des APIs • Le Caching • La documentation • La gestion du trafic : fixation des quotas de consommation, amorçage

		des arrêts rapides, limitations des utilisations • Les métriques sur : le trafic, la popularité, • Le portail de développeur : enregistrement et connexion des utilisateurs, console pour tester l'API, enregistrement des applications et gestion des Tokens
La sécurité		
Authentification et l'autorisation	Quelles sont les connaissances à maitriser ?	• OAuth2 • L'OpenID Connect (OIDC) • HTTPS
La sécurité entre les microservices	Quelles sont les connaissances à maitriser ?	• L'authentification HTTPS basique • Les certificats clients (TLS) • Le code d'authentification d'une empreinte cryptographique de message avec clé (HMAC)
La sécurité entre les APIs	Quelles sont les connaissances à maitriser ?	• L'authentification d'une API via une clé : UUID • L'autorisation à l'aide d'un token • La médiation d'identité.
Les menaces	Sur quoi faut-il être vigilant	• L'injection • L'authentification mal gérée • L'exposition des données sensibles • Le DDoS (Deny-of-Service) • Le manque de contrôle des accès • Les mauvaises configurations • Le Cross-site Scripting (XSS) • La désérialisation non sécurisée • L'utilisation des APIs externes non sécurisées • Le manque de logs et de monitoring
Etude de cas		

Application COMOVE	Que va-t-on y voir ?	Un exemple de cas pratique de mise en place d'une architecture de microservices sur une application axée mobile-first et portée par le cloud • Conception DDD • Architecture cible d'intégration • Architecture incluant la mise en place d'une brique d'API management • Implémentation d'un modèle de sécurité • Architecture cible de déploiement maison (Bonus)
Déploiement		
	Que retenir ?	• L'intégration continue : 1 build par microservices • La livraison continue : Faire des pipelines pour gagner en performances • Les stratégies de déploiements : la virtualisation (AWS), La conteneurisation (Docker), le ServerLess (AWS, Google, Azure)
Le Monitoring		
	Pourquoi monitorer une architecture de microservices ?	• Pour détecter les pannes, les analyser et apporter des solutions • Pour améliorer les performances du système. • Pour augmenter la qualité d'un contrat de niveau de service (SLA)
	Quelles métriques doit-on surveiller ?	• Les métriques d'applications • Les métriques de plateformes • Les métriques basées sur les évènements externes au systèmes

Transformation digitale		
Transformer un monolithe en microservices	Comment procède-t-on ?	• Bien définir les raisons qui poussent à la migration • Définir la stratégie de conception en sous-domaines : contextes bornées, seams et carte de contextes • Procéder par itérations • Utiliser une stratégie de transformation graduelle via le Strangler Pattern

Au final…

Il y'a plus d'une décennie, on présentait les architectures SOA comme une solution intrinsèque pour mettre fin à la principale problématique rencontrée sur les solutions en monolithe qui était la capacité à intégrer entre elles, des applications hétérogènes, les rendant ainsi, plus flexibles, adaptables et agiles. Cependant, c'était sans compter sur les plateformes d'ESB complexes qui devaient effectuer le transcodage des données et leurs orchestrations et qui étaient le point central de ces nouvelles architectures. La déception des grandes entreprises réside dans le fait que non seulement ces plateformes étaient couteuses et de plus étaient le point des faiblesses de leur architecture. En effet, le moindre problème survenu sur cet élément avait un impact immédiat sur l'architecture et par conséquent sur le métier de l'entreprise. L'idée d'avoir une deuxième ESB comme solution de bascule en cas de problème semblait être la solution la plus appropriée.

Au même moment, les géants du Web (Google, Facebook, Amazon…) ont bâti des systèmes d'informations d'une incroyable performance, en gérant des volumes de données et de requêtes gigantesques, avec des règles métiers simples tout en se basant sur des architectures de services à taille réduite dont l'implémentation était inspirée sur quelques concepts seulement des architectures SOA. L'architecture de microservices avait fait ses preuves.

L'architecture des microservices ne s'oppose pas à la SOA, puisque qu'elle en est l'implémentation à un niveau de granularité plus fin et avec certaines normes qui ne nécessite pas forcement la présence d'un élément central pour rendre les applications autonomes. Si la SOA demeure une technique d'urbanisation du SI et de ses applications, une approche basée sur les microservices remet en cause

l'architecture au sein même des applications en réduisant leur base de code, en découplant au maximum les éléments de différents domaines fonctionnels et en normalisant les échanges de données à un seul format de données (JSON) et en communiquant via un seul protocole (HTTP). Cette séparation des responsabilités permet à un système d'améliorer sa résilience globale, de le décomplexifier et de le faire évoluer tant sur le plan technique que technologique (capacité à innover).

La mise en place d'une architecture de microservices, nécessite une organisation et une structuration des échanges de données entre les services ainsi qu'une harmonisation en termes de sécurité et des informations de l'ensemble d'entre eux. Cette gestion induit de disposer d'une vue macroscopique sur les microservices. C'est pour cette raison que le concept d'interface d'entrée/sortie qui permet d'accéder aux différentes ressources via les microservices est si précieux. En effet, les APIs permettent de consommer de la donnée des microservices sans exposer la complexité de leur implémentation et également à ces derniers d'échanger des données entre eux.

Dans un processus de croissance et de gouvernance, la brique d'API management permet de référencer différentes API en les gérant de façon centralisée via des fonctions d'authentification, de métriques, de contrôle d'usage (throttling), de documentation, de transformations de données, d'orchestration et de sécurité. L'API management permet d'implémenter une stratégie de transformation digitale en gérant le cycle de vie des API qui regroupent chacune des groupes d'un ou plusieurs microservices.

Bien qu'une approche en microservices consiste à éviter les risques d'avoir des systèmes trop complexes et trop couplés afin de garder le contrôle de l'architecture, il n'en demeure pas moins que son implémentation nécessite une certaine rigueur, un environnement adapté et un respect des normes et bonnes pratiques afin de ne pas accumuler les inconvénients. Le choix d'une solution d'API mangement ne doit pas se faire de façon précipitée ni anodine afin d'éviter une dépendance avec le fournisseur de solution. Dans tous les cas, il faut se poser les questions de savoir si la mise en place d'une architecture de microservices :

- Permet de résoudre les problèmes qui surviennent dans le système d'information
- Est en accord avec la stratégie de l'entreprise
- Est possible avec les conditions existantes
- Est adaptable aux besoins (présents et futurs) de l'entreprise
- Peut avoir des impacts mesurables sur le système d'information

Une fois décidé, une architecture de microservices peut être une bonne solution à implémenter surtout s'il s'agit d'éviter des systèmes complexes et de garantir la flexibilité à l'innovation.

BIBLIOGRAPHIE

LIVRES :

- Buliding Microservices, Samuel Newman. O'REILLYMedia.
- SOA, microservices & API management : le guide de l'architecte des SI agiles, Morel Fournier. Dunod.
- API management : An architect's guide to developping and managing APIs for your organization, DE Brajesh. Apress.
- Restful API Design : Best practices in API design with REST, Mathias Biehl.
- Enterprise API management, Luis Augusto Weir.
- REST API Design Rukebook : Designing consistent RESTful web service Interfaces, Mark Masse. O'REILLY
- API development : A practical guide for business implementation success, Sascha Preibisch.
- OAuth 2.0: Getting started in API security, Mathias Biehl.
- Amazon webservices for dummies, Bernard Golden.
- Microservice architeture : alignin principles, practices and culture, Nadareishvili Irakli. O'REILLY .
- APIs are different than integration, Ed Anuff. Apigee.
- APIs for dummies – Sharif Nijim, Brian Pagano. Apigee Special Edition.
- Microservices Reference Architecture, Chris Stetson. NGINX. O'REILLYMedia.
- Spring: Microservices with Spring Boot, Rao Ranga Karanam. Packt Publishing.
- Developping microservices with Node.js, David Gonzalez. Packt Publishing.
- Docker : Déploiement des microservices sous Linux ou windows, Jean-Philippe Gouigoux. Edition ENI.
- RESTful Web APIs : Services for a changing worl, Leonard Richardson, Mike Amunden & Sam Ruby. O'REILLY.
- Practical Monitoring: Effective strategies for real world, Mike Julian. O'REILLY.
- Monolith to Microservices: Evolutionary Patterns to transform your monolith, Sam Newman. O'REILLY.

AUTRE DOCUMENTS :

- Microservices and containers, Parminder Sing Kocher. KINDLE.
- DevOps and Microservices HandBook, Stephen Fleming. KINDLE.
- Microservices on AWS, (AWS white paper). KINDLE

- OWASP, TOP 10 2017. PDF ici
- Domain-Driven-Design Quickly, PDF ici
- Security in microservices architectures, PDF ici
- Microservices API security, PDF ici
- Security Strategies for Microservices-based Application Systems, PDF ici

COURS

- Construisez des microservices - cours Openclassroom
- Utilisez des API REST dans vos projets web - Openclassroom
- Optimisez votre architecture microservices - Openclassroom
- Google Maps Javascript API V3 – Openclassroom
- Optimisez votre déploiement en créant des conteneurs avec Docker - Openclassroom
- Utiliser l'API MySQL dans vos programmes – Openclassroom
- Développez des applications web avec Angular 5.x – Openclassroom
- Découvrez le cloud avec Amazon Web Services – Openclassroom
- Maîtrisez les bases de données NoSQL - Openclassroom
- Apprendre Node.js & créer une API REST de A à Z – Udemy
- Les web Components par la pratique – cours Udemy
- Learn to Develop for Cloud with Pivotal Cloud Foundry – Udemy
- Restful Webservices, Java Spring Boot, Spring MVC, JPA and AWS – Udemy
- REST API Design, Management, Monitoring and Analytics – Udemy
- Swagger and the Open API specification – Udemy
- Apache Kafka Series, Kafka monitoring and operations - Udemy
- Microservices Architecture – Pluralsight
- Microservices Architectural Design Patterns – Pluralsight
- Java Microservices with Spring Cloud : Coordinating Services – Pluralsight
- Enterprise Strength Mobile Security – Pluralsight

BLOGUES :

- www.martinfowler.com
- https://microservices.io/
- https://www.infoq.com/
- https://developers.redhat.com/blog/category/microservices
- https://blog.octo.com/
- https://www.nginx.com/blog/
- https://informationisbeautiful.net/

- https://developers.googleblog.com/
- https://eng.uber.com/
- https://dzone.com/
- https://www.baeldung.com/

VIDEOS (YouTube) :

- Design microservice Architectures the Right way
- Principles of Microservices by Sam Newman
- DDD, en vrai pour le développeur (Cyrille Martraire)
- Introduction to Microservices, Docker, and Kubernetes
- DDD: et si on reprenait l'histoire par le bon bout? (Thomas Pierrain - Jérémie Grodziski)
- Microservice communication and integration: what are my options?
- End-to-End Automated Testing in a Microservices architecture – Emily Bache
- Distributed API management in a hybrid cloud environment
- API management best Practices (Cloud Next' 18)
- Authentification et autorisation dans une architecture microservices
- Mastering Chaos - A Netflix Guide to microservices
- developing microservices with aggregates - Chris Richardson
- Message-based Microservices architectures- benefits and Practical Matters
- GOTO 2017. The many meanings of event-Driven Architecture. Martin Fowler
- Monitor Your Microservices with Logs, Metrics, Pings and Traces (David Pilato)
- Three Microservice Patterns to Tear Down Your Monoliths
- Revitalizing Aging architectures with microservices
- GOTO 2016. From Monolith to Microservices at Zalando . Rodrigue Schaefer
- Migrating a monolithic application to Microservices 9cloud Next, 9)
- Building high performance microservices with Kubernetes, GO and gRPC