

# Formation MicroService

Ihab ABADI / UTOPIOS

# SOMMAIRE

## Partie 1

1. Rappel Pattern Microservice et Pattern Associé.
2. Décomposition des services
3. Communication entre Microservices
4. Gestion de données en microservices
  1. Base de données par service
  2. Base de données partagée
  3. Pattern saga

# SOMMAIRE

## Partie 2

1. Requête cross microservice.
  1. Utilisation API Composition
  2. Utilisation du Pattern CQRS
2. Utilisation d'API Gateway
  1. Utilisation d'un seul d'API Gateway
  2. Utilisation de plusieurs API Gateway
3. Service Discovery
  1. Le Register
  2. Le Resolver
4. Tester des Microservices
  1. Component testing
  2. Contract testing

# Rappel Pattern MicroService et pattern associé

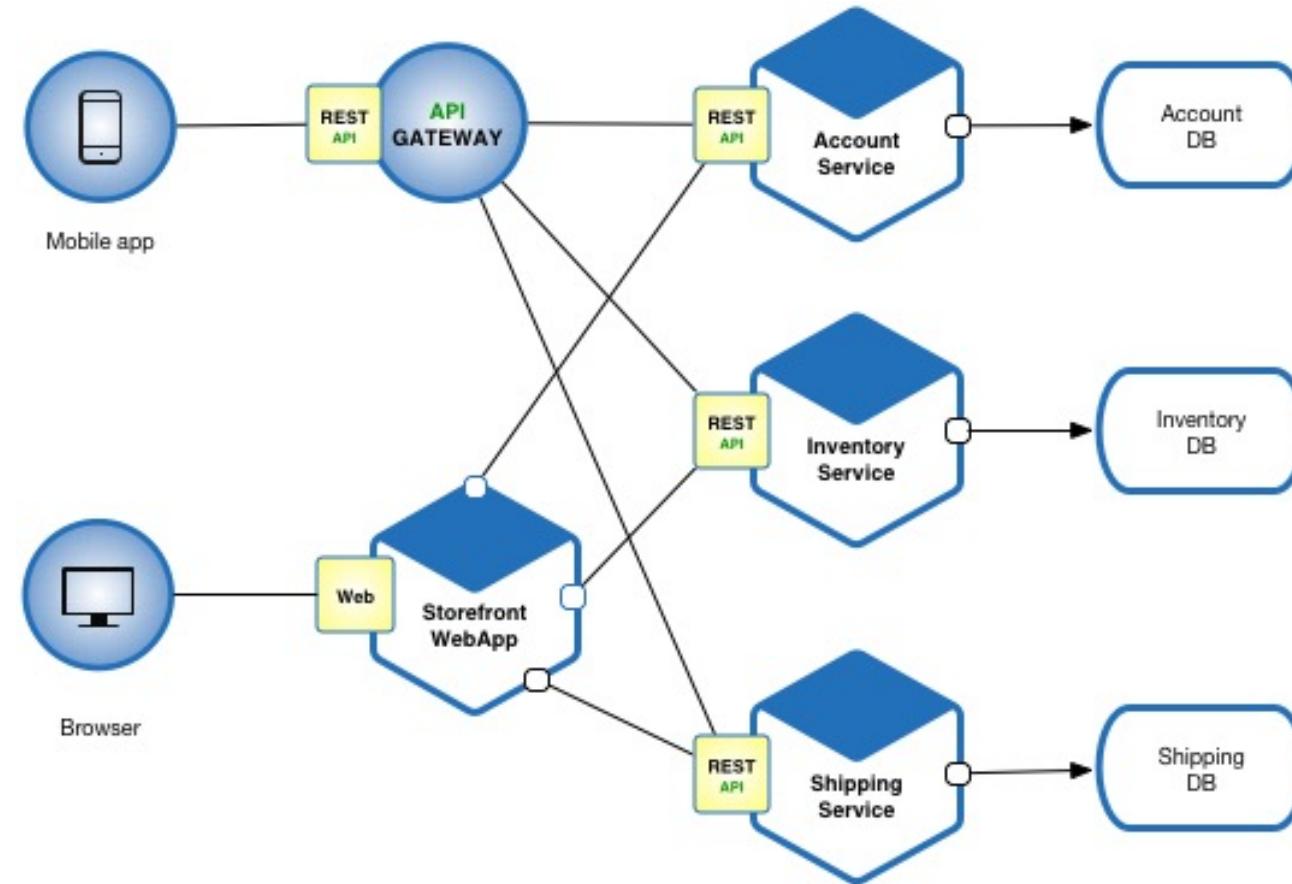
- Pourquoi choisir un Pattern MicroService ?
- Dans un contexte de réalisation d'une application qui :
  - Utilise plusieurs clients( mobile, Desktop, web).
  - Expose une API Public
  - Interagit avec d'autre application par système de messagerie
  - Exécute différent logiques métier et interagit avec plusieurs source de données.
  - Gère plusieurs protocole de communication.
- Avec comme condition :
- Possibilité de travailler à plusieurs équipes
- Facilement maintenable
- Facilement scalable
- Facilement testable.

# Rappel Pattern MicroService et pattern associé

- Solution :
  - Une approche de développement par services fortement découplés
  - Chaque service déployable indépendamment des autres
  - Chaque service totalement autonome
  - Chaque service développé indépendamment des autres

# Rappel Pattern MicroService et pattern associé

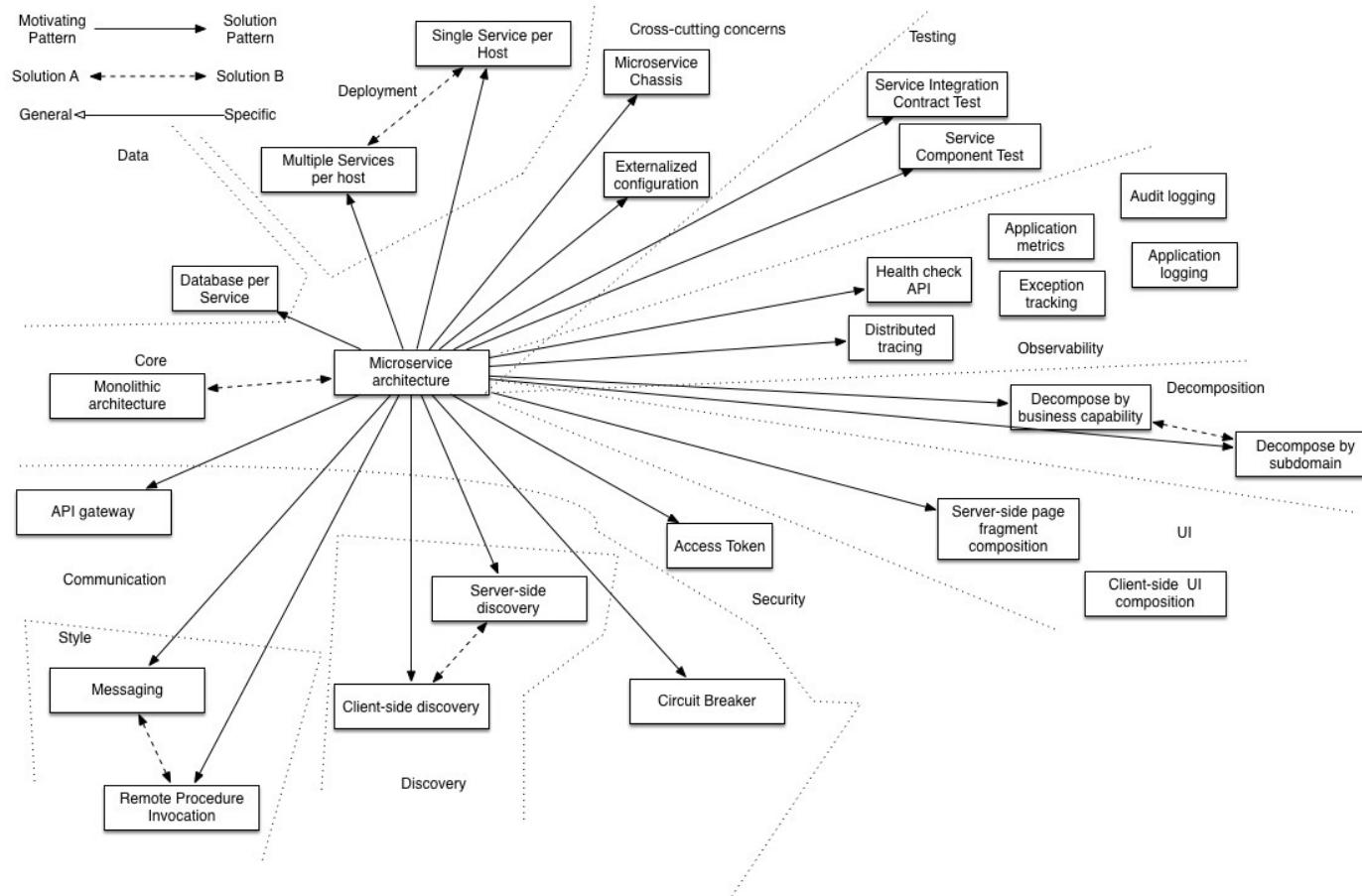
- Exemple site Ecommerce



# Rappel Pattern MicroService et pattern associé

- Problématiques liées à l'utilisation des mircoservices
- Problématiques de développement
- Problématiques de déploiement
- Problématiques de sécurités

# Rappel Pattern MicroService et pattern associé



# Décomposition des services

- Pour profiter des avantages d'une architecture microservice il faut réussir à découper notre application en service assez petit pour être facilement testable, développé par une petite équipe et également facilement déployable.
- Chaque service doit apporter une plus value sans dépendre des autres services.
- Pour réussir notre décomposition, on peut utiliser :
- Décomposition par Capacité métier
- Décomposition par sous domaine

# Décomposition par capacité métier

- Décomposition par capacité Métier est un sous pattern de la modélisation d'architecture d'entreprise.
- Capacité métier est tout élément qui apporte de la valeur ajoutée à notre application
- Le but est de décomposer l'application en service qui apporte de la valeur.
- Exemple E-Commerce :
- Products management
- Cart management
- Shipping Management
- Order Management
- Payment management
- Shipping Management
- Notifications (Email/SMS)Management

# Décomposition par DDD

- Ce modèle modélise les microservices autour du DDD (Domain-Driven Design).
- DDD est une approche de développement logiciel qui repose sur les principes et les idées de l'analyse et de la conception orientées objet.
- Dans DDD, un modèle de domaine utilise les connaissances sur un sous domaine pour résoudre le problème sur un domaine.
- Dans DDD, tout le monde utilise le même vocabulaire dans les équipes.
- Ceci est connu sous le nom de «langage omniprésent». en termes DDD.
- Exemple Ecommerce

# Communication Entre Microservice

- Pour faire communiquer deux microservices, on peut utiliser :
- Communication en http avec des API Rest
- Communication avec un protocole gRpc
- Communication avec broker de messages

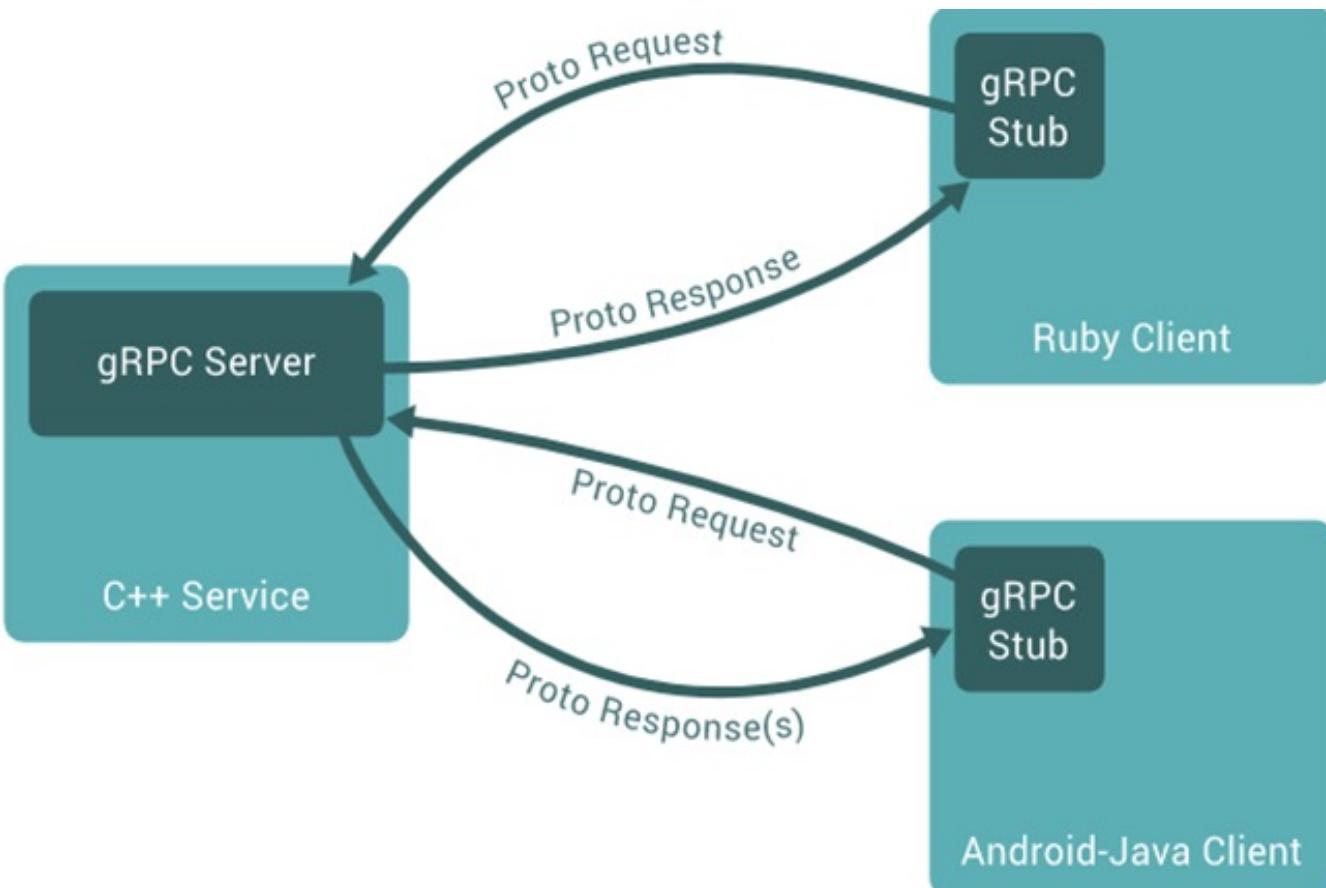
# Communication Entre Microservice API REST

- Dans le cadre d'une application microservices, nos services sont en interaction en eux.
- On peut utiliser une communication en REST basée sur principe d'une requête, réponse
- Utilisation d'une communication en REST impose que les deux services soit actifs
- Utilisation d'une communication en REST impose une communication unidirectionnelle

# Communication Entre Microservice gRPC

- gRPC est un framework de communication openSource développé par google
- gRPC offre des performances très élevées
- gRPC utilise HTTP/2 pour transporter des messages binaires
- gRPC utilise un mécanisme de contrat de service défini à l'aide d'un Protocol Buffers
- Utilisation du framework gRPC permet l'utilisation de client et server issues de plusieurs technologies

# Communication Entre Microservice gRPC



# Communication Entre Microservice gRPC

- Protocol buffers est un mécanisme open source développé par Google qui permet la sérialisation et la structuration des données
- Protocol buffers permet de structurer les données sous forme de messages
- Chaque message représente une petite unité logique

# Communication Entre Microservice gRPC

- 1- Définition du protoBuf
- 2- Compilation du ProtoBuf
- 3- Génération des services
- 4- Implémentation soit client ou serveur

# Communication Entre Microservice Broker message

- La communication par Broker de message est asynchrone
- La communication par Broker de message est très fiable et garantie une stabilité dans l'utilisation
- Le choix du Broker se fait en fonction de plusieurs critères
  - Scalabilité du courtier - Le nombre de messages envoyés par seconde dans le système.
  - Persistance des données - La possibilité de récupérer des messages.
  - Capacité du consommateur - Indique si le courtier est capable de gérer des consommateurs un à un et / ou un à plusieurs.

# Communication Entre Microservice Broker message

- RabbitMQ
- Redis
- kafka
- Démo kafka

# Gestion de base de données en Microservice

- Dans toute application, il faut gérer une persistance dans une base de données
- Pour une architecture Microservice, la gestion des données peut se faire de plusieurs façon toute en respectant les paradigme du pattern

# Gestion de base de données en Microservice

- Première solution :
- Chaque service gère sa propre persistance, accessible uniquement en API, à travers :
- Tables privées uniquement par service
- Schéma par service
- Base de données par service

# Gestion de base de données en Microservice

- L'utilisation d'une base de données par service présente les avantages suivants :
  - Aide à garantir que les services sont faiblement couplés. Les modifications apportées à la base de données d'un service n'ont aucun impact sur les autres services.
  - Chaque service peut utiliser le type de base de données le mieux adapté à ses besoins. Par exemple, un service qui effectue des recherches de texte pourrait utiliser ElasticSearch. Un service qui manipule un graphe social pourrait utiliser Neo4j.
- L'utilisation d'une base de données par service présente les inconvénients suivants :
  - La mise en œuvre de transactions couvrant plusieurs services n'est pas simple.
  - La mise en œuvre de requêtes qui joignent des données qui se trouvent désormais dans plusieurs bases de données est un défi.
  - Complexité de la gestion de plusieurs bases de données SQL et NoSQL

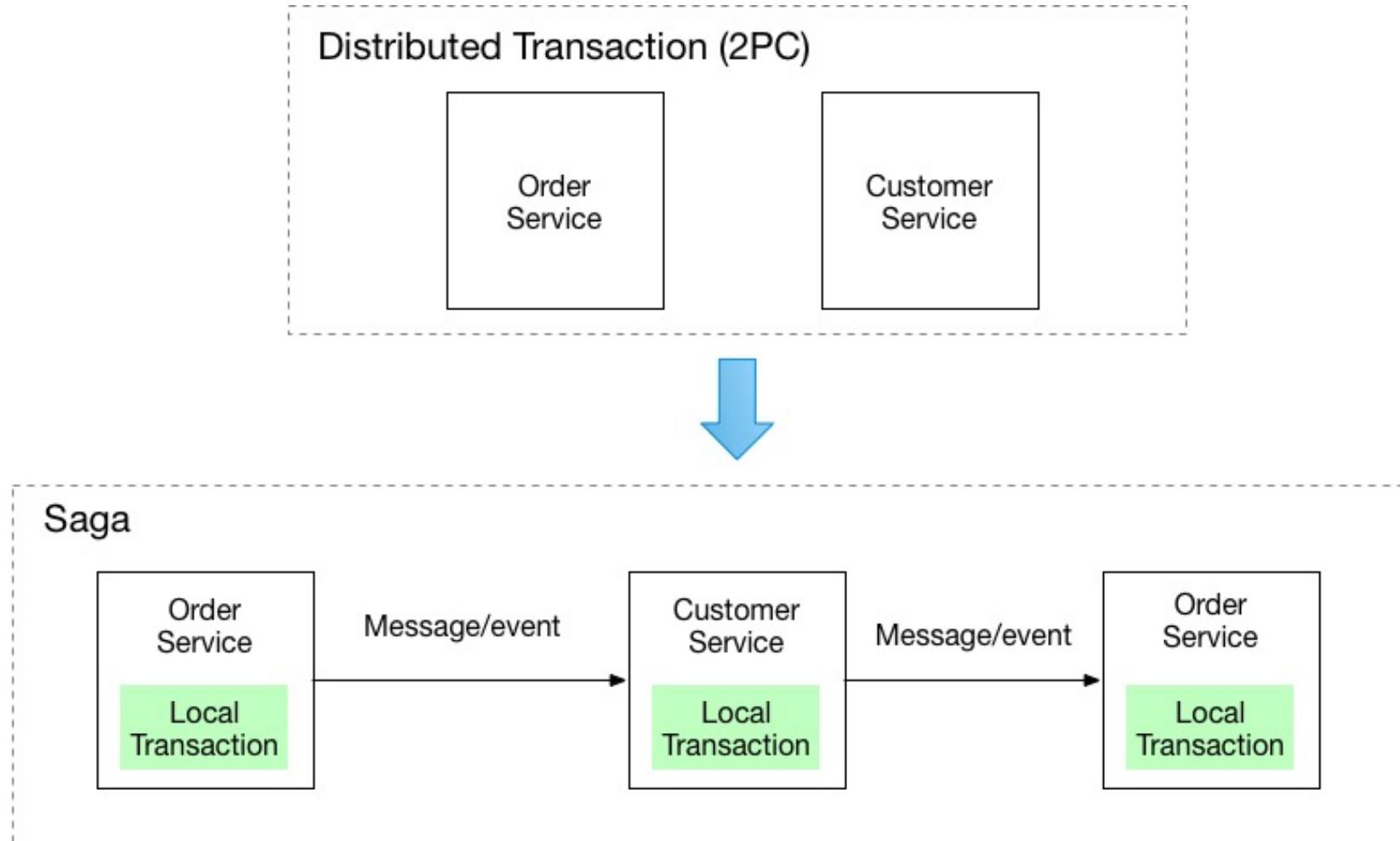
# Gestion de base de données en Microservice

- Deuxième solution :
- Avoir des données partagées entre les services
- Les avantages de ce modèle sont :
  - Un développeur utilise des transactions et procédures simples pour assurer la cohérence des données
  - Une seule base de données est plus simple à exploiter
- Les inconvénients de ce modèle sont :
  - Couplage du développement - un développeur travaillant sur le service de commande devra coordonner les changements de schéma avec les développeurs d'autres services qui accèdent aux mêmes tables. Ce couplage et cette coordination supplémentaire ralentiront le développement.
  - Couplage d'exécution - parce que tous les services accèdent à la même base de données, ils peuvent potentiellement interférer les uns avec les autres. Par exemple, si une longue transaction CustomerService détient un verrou sur la table ORDER, le OrderService sera bloqué.
  - Une seule base de données peut ne pas répondre aux exigences de stockage de données et d'accès de tous les services.

# Microservice SAGA

- Dans le cadre d'une base de données par service, pour résoudre la problématique des transactions cross services, on peut utiliser le pattern SAGA
  - Une Saga est une séquence de transactions locales.
  - Chaque transaction locale procède à la mise à jour de la base de données et publie un message pour déclencher la prochaine transaction locale
  - Si une transaction locale échoue, on exécute des transactions de compensation qui annulent les transactions locales
- Implement each business transaction that spans multiple services is a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions. précédentes

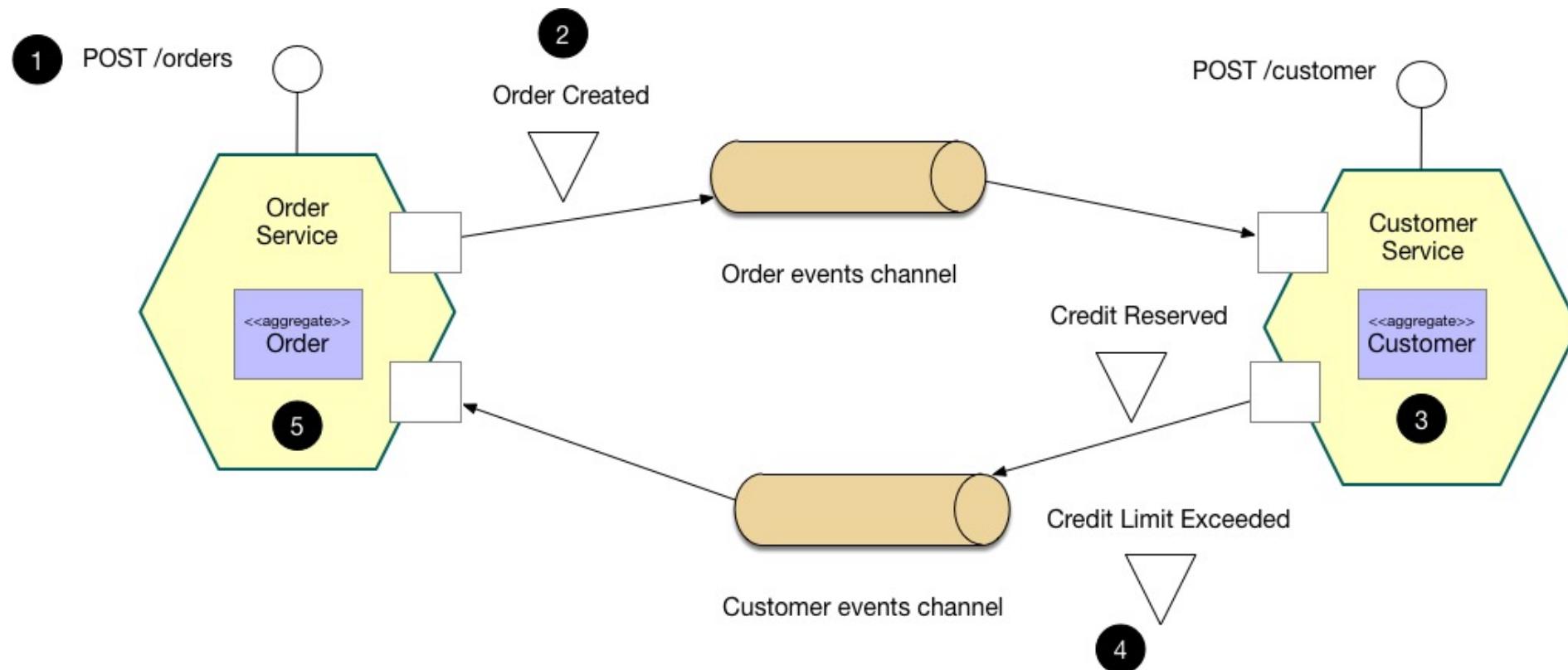
# Microservice SAGA



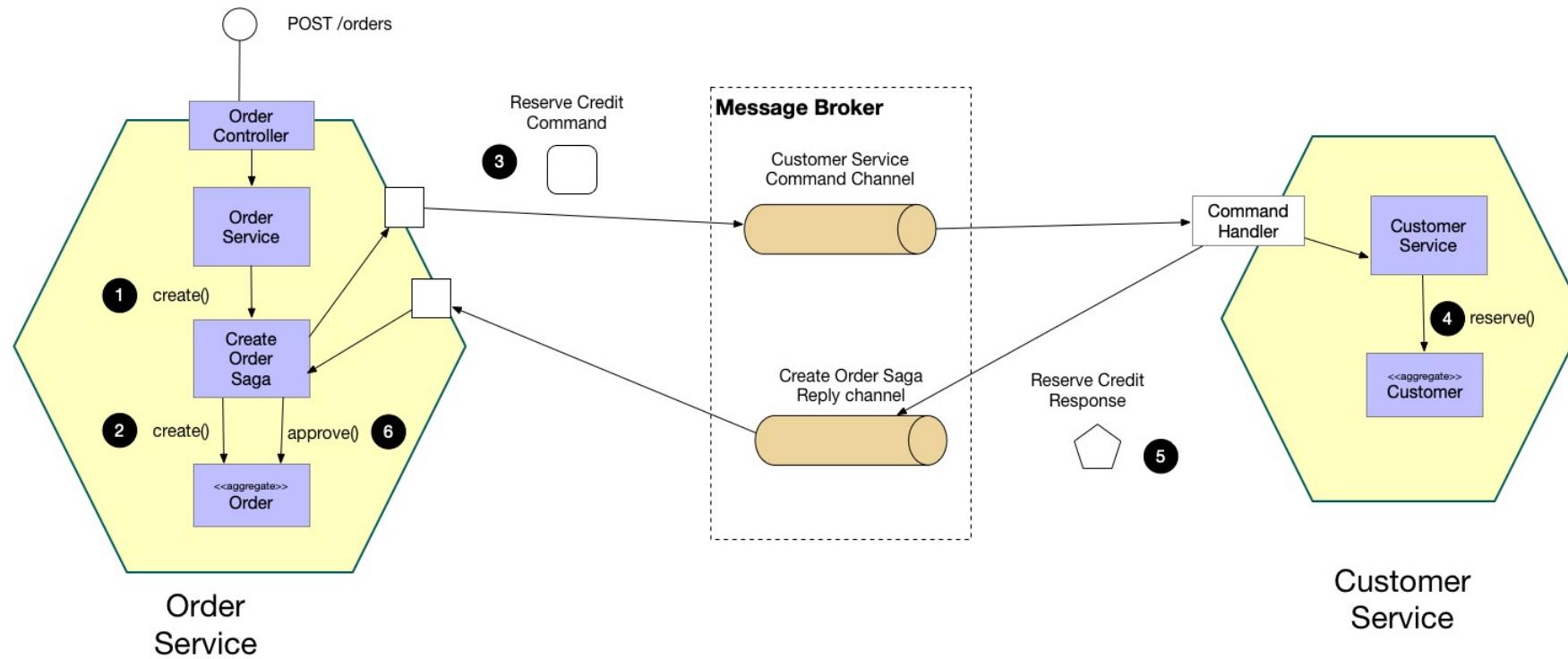
# Microservice SAGA

- On peut implémenter le pattern SAGA Soit de manière :
  - Chorégraphique
  - Par orchestration

# Microservice SAGA Chorégraphique



# Microservice SAGA Orchestration

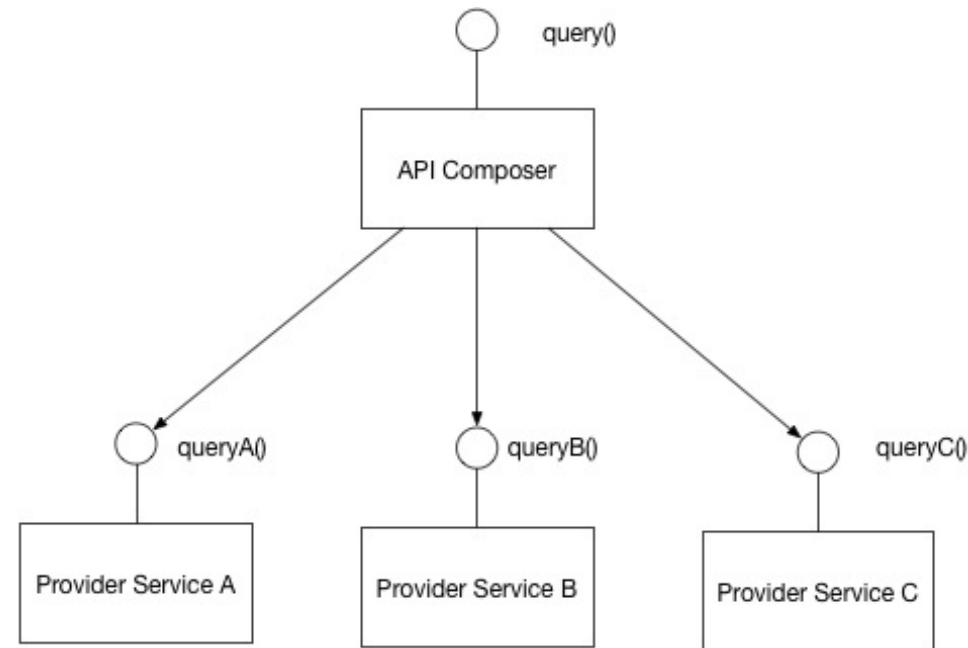


# Requête cross services

- Dans une architecture microservices, avec une base de données par service (par exemple), il devient compliquer de faire des jointures sur des tables relationnelles (par exemple).
- Pour répondre à cette problématique, on trouve une multitude de patterns par exemple:
- API Composition
- Command Query Responsibility Segregation (CQRS)

# Requête cross services (API Composition)

- API Composition est un pattern qui consiste, dans le cadre d'une requête sur plusieurs services, à utiliser un service intermédiaire qui appelle chaque services et procède à une jointure.



# Requête cross services (API Composition)

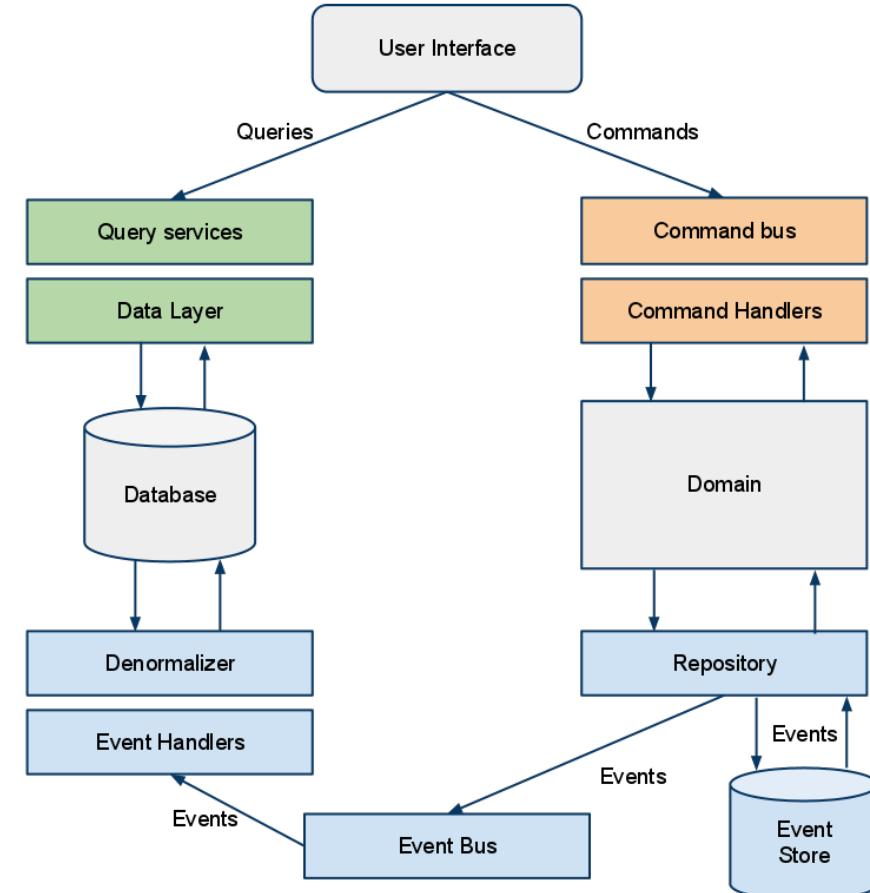
- Démo Avec le framework NestJs:
- On utilisera des services HttpClient
- On utilisera également un observable zip de la librairie RxJS

# Requête cross services (API Composition)

- Exercice
- En se basant sur notre projet de la dernière séance, on souhaite mettre en place une API composition pour récupérer à la fois le client et ses commandes.

# Requête cross services (CQRS)

- CQRS consiste à avoir un microservice dédié aux requêtes associé à une vue de nos données (en mode lecture seul)
- La vue est régulièrement mise à jour à partir d'évènement qui écoute des commandes dans les autres microservices



# Requête cross services (CQRS)

- Démo

# Requête cross services (CQRS)

- Exercice
- En se basant sur notre projet de la dernière séance, on souhaite mettre en place le pattern CQRS pour récupérer à la fois les commandes et les livraison.

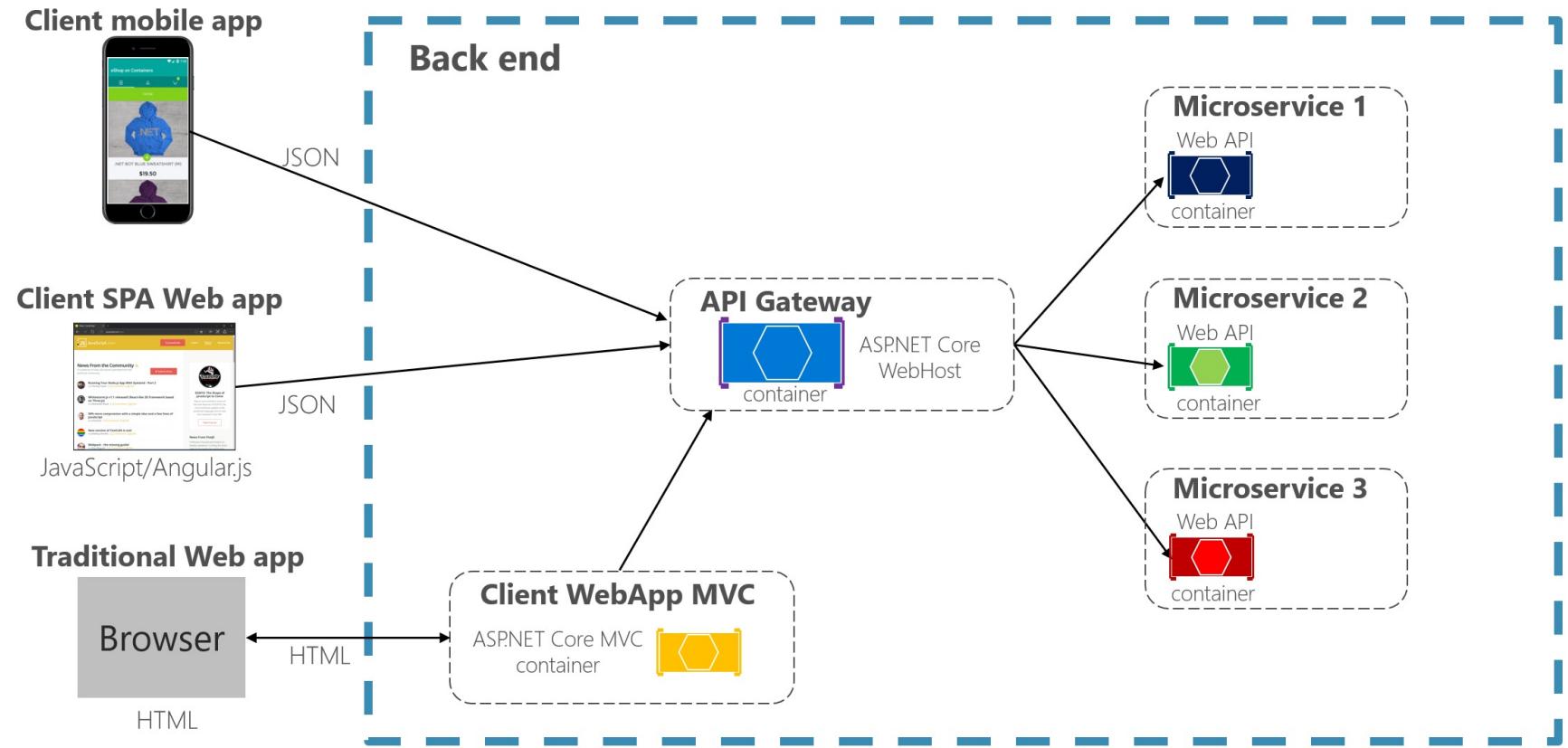
# API GateWay

- Dans le cadre d'un projet qui contient plusieurs clients (Application Web, Application Mobile,...), chaque partie utilise plusieurs modules et chaque module invoque un ou plusieurs microservice différents.
- Pour éviter de devoir enregistrer à la main les différents service (Host et port), on utilise un nouveau microservice qui joue le rôle de notre relais vers les autres microservices.
- Une application peut avoir un seul GateWay, ou plusieurs

# API GateWay

- Exemple 1 gateWay

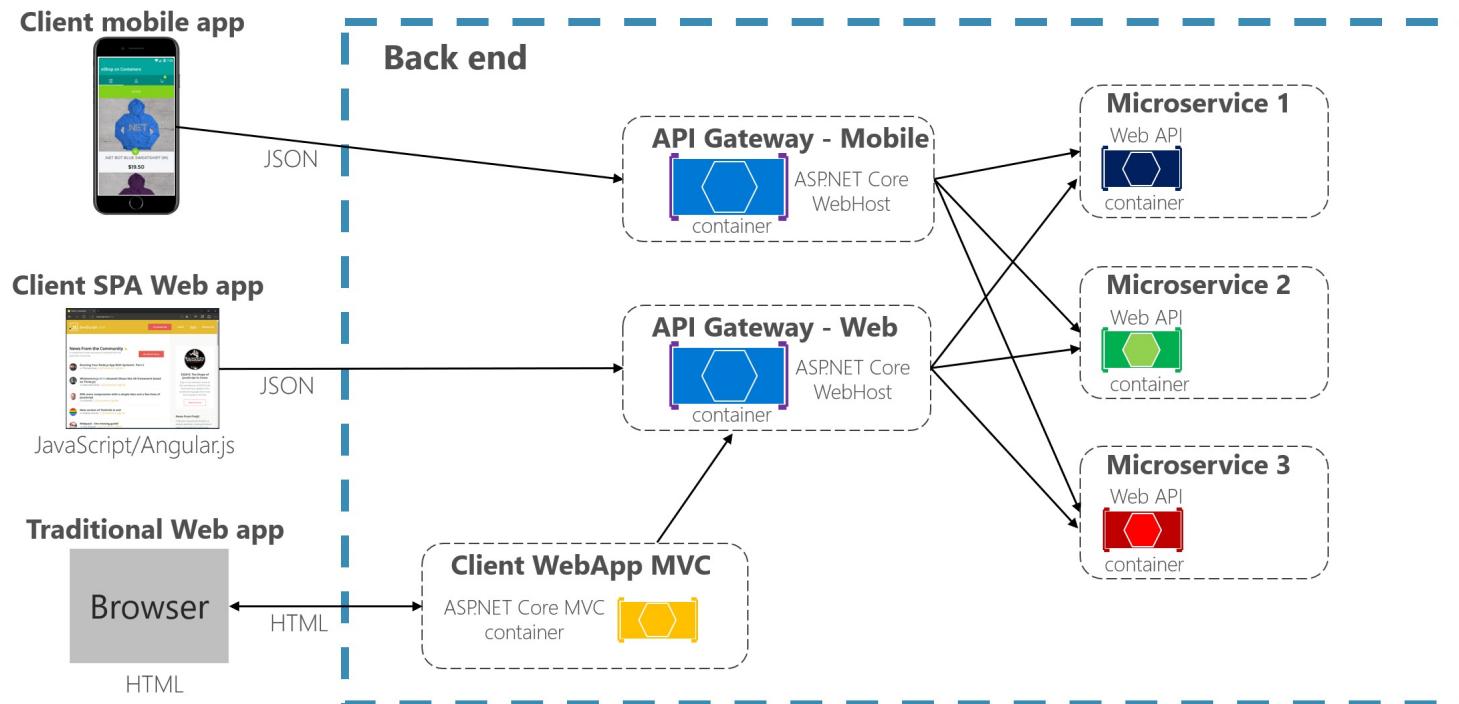
## Using a single custom **API Gateway service**



# API GateWay

- Exemple plusieurs gateWays

## Using multiple **API Gateways / BFF**



# API GateWay (Exercice)

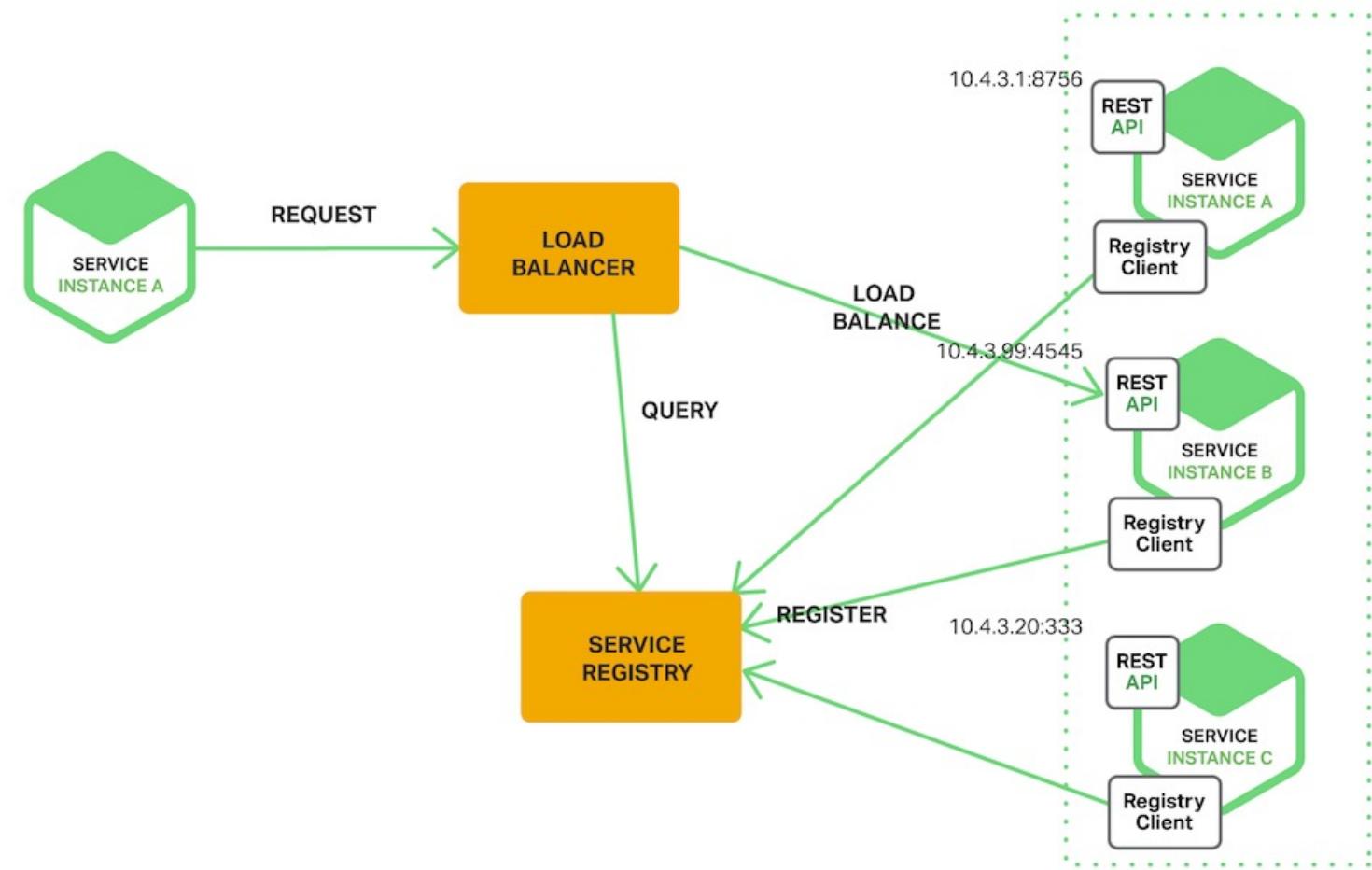
- Réaliser une API Gateway qui permet à des applications cliente de récupérer les produits, commandes, clients

# Service Discovery

- Dans une architecture microservice, les services communiquent entre eux.
- Dans une application monolithique, les services s'invoquent les uns les autres via des appels de méthode ou de procédure au niveau du langage.
- Dans un déploiement de système distribué traditionnel, les services s'exécutent à des emplacements fixes et bien connus (hôtes et ports) et peuvent donc facilement s'appeler à l'aide de HTTP/REST ou gRPC.
- Une application moderne basée sur des microservices s'exécute généralement dans des environnements virtualisés ou conteneurisés où le nombre d'instances d'un service et leurs emplacements changent de manière dynamique.
- Par conséquent, on doit implémenter un mécanisme permettant aux clients de service d'adresser des requêtes à un ensemble d'instances de service éphémères changeant dynamiquement.

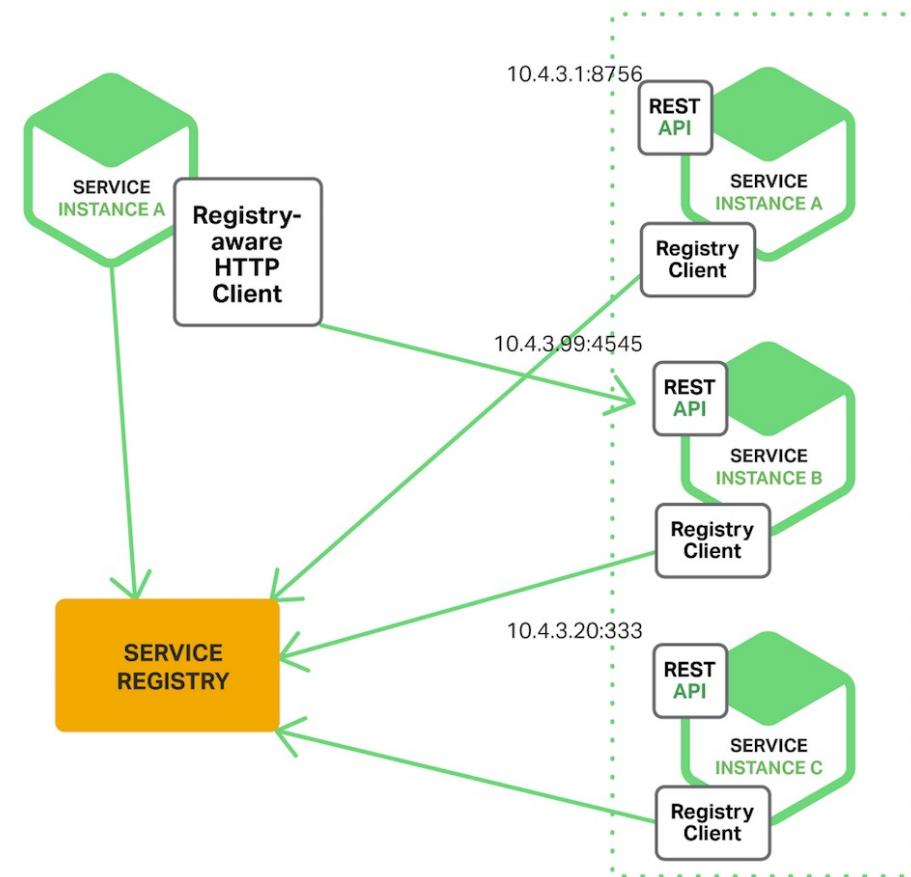
# Service Discovery

- On peut distinguer deux types de service Discovery
- Server side



# Service Discovery

- On peut distinguer deux types de service Discovery
- Client side



# Service Discovery

- Pour mettre en place un service Discovery il faut avoir :
- Une base de données (clé, valeur) très performante.
- Un mécanisme de register qui va permettre à chaque microservice d'indiquer sa position.
- Un mécanisme de resolver qui va permettre de récupérer l'adresse de chaque microservice

# Service Discovery

- Il existe une multitude de service Discovery open source exemple :
- Consul
- Krakend
- Etcd
- Démo

# Service Discovry (Exercice)

- Modifier notre application, ainsi que notre API GateWay pour y ajouter un mécanisme de service Discovry

# Tester des microservices

- Pour tester efficacement une application microservice, on peut se baser sur 4 types de test:
- Test Unitaire
- Test d'intégration
- Test de composants
- Test contractuel