

Formation ARC .NET

Ihab ABADI / UTOPIOS

SOMMAIRE

1. Introduction Architecture logiciel
2. Notion de couches dans l'architecture logiciel
3. Architecture SOA
4. Architecture microservice
5. Design patterns de base
6. Design patterns de persistance
7. Design patterns distribuée

Introduction Architecture logiciel

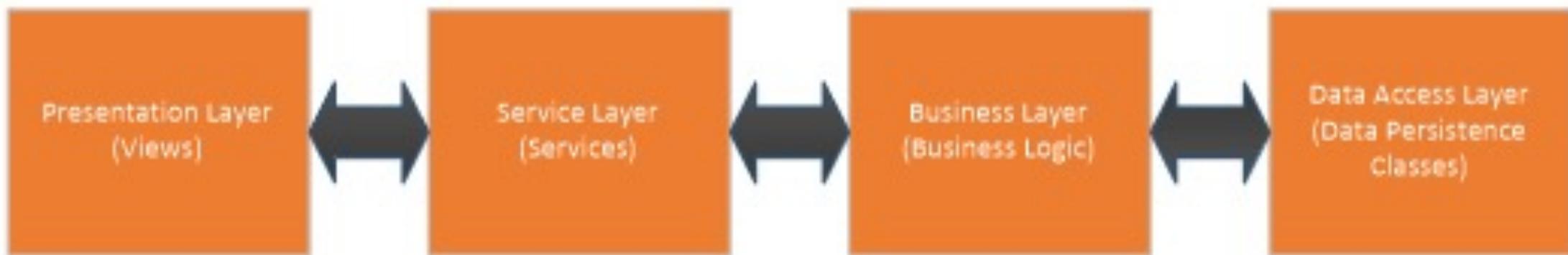
- Le métier d'architecture logiciel consiste à :
 - Créer les frameworks pour gérer l'architecture.
 - Détalier les descriptions de l'architecture.
 - Définir les feuilles de route pour définir la meilleure façon de changer/améliorer l'architecture du logiciel.
 - Définir les contraintes/opportunités.
 - Anticiper les coûts et les bénéfices.
- Évaluer les risques et les valeurs.

Notion de couches dans l'architecture logiciel

- Il existe différentes approches lors du développement d'une architecture de solution.
- La complexité ne doit être ajoutée à l'architecture que lorsque cela est nécessaire.
 - Ajouter de la complexité à l'architecture n'est pas une mauvaise pratique et est acceptable si cela résout un besoin particulier ou réduit l'effort de développement des développeurs pendant le cycle de vie du développement.
- Dans une architecture en couches, l'application se compose de différentes couches, à savoir les couches de présentation, de service, d'entreprise et d'accès aux données, et chaque couche est responsable de la réalisation de tâches spécifiques.
- L'approche en couches est l'une des approches largement utilisées dans le développement d'applications d'entreprise.
- Les couches les plus utilisées sont :
 - Couche de présentation.
 - Couche de service.
 - Couche de logique métier.
 - Couche d'accès au données

Notion de couches dans l'architecture logiciel

- c'est un système faiblement couplé.
- Les équipes peuvent travailler sur différentes couches, en parallèle, avec un minimum de dépendances vis-à-vis des autres équipes.
- Les modifications apportées à n'importe quelle couche en termes de technologie ou de logique métier ont peu d'impact sur les autres couches.
- Les tests peuvent être effectués facilement.



Couche de service

- La couche service est la couche principale, qui relie la couche présentation à la couche métier.
- Elle expose certaines méthodes via des bibliothèques de classes, des API Web ou des services Web utilisés par la couche de présentation pour effectuer certaines opérations et accéder aux données.
- Elle encapsule la logique métier et expose POST, GET et d'autres méthodes HTTP pour exécuter certaines fonctionnalités.

Couche de métier

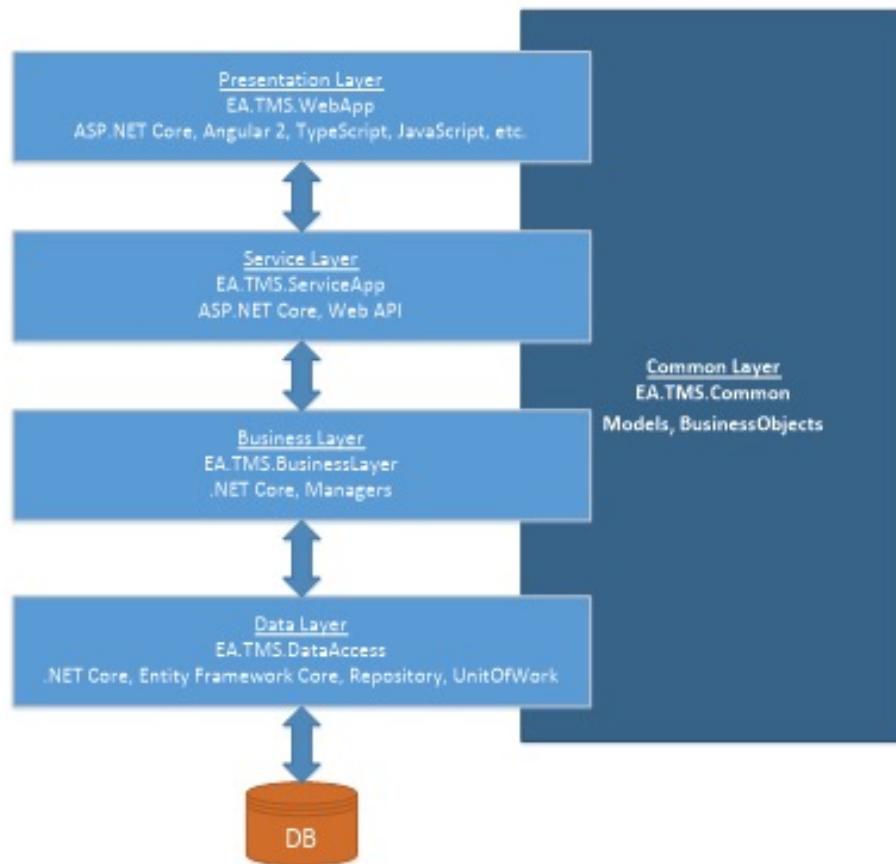
- La couche métier est la couche centrale de l'architecture en couches. Il contient la logique réelle de l'application et gère les événements qui se déclenchent à partir de la couche de présentation.
- Il existe différentes approches lors de la construction de la couche métier et chaque approche dépend de la portée du projet.

Couche de métier

- La couche métier est la couche centrale de l'architecture en couches. Il contient la logique réelle de l'application et gère les événements qui se déclenchent à partir de la couche de présentation.
- Il existe différentes approches lors de la construction de la couche métier et chaque approche dépend de la portée du projet.

Notion de couches dans l'architecture logiciel

Démo



SOA Définition

- SOA est un style architectural qui promeut principalement l'orientation service.
- L'orientation service implique des systèmes faiblement couplés qui sont fondamentalement axés sur la satisfaction des fonctions commerciales.
- En SOA, on pense en termes de services qui exécutent des processus métier de manière autonome.
- Il est plus simple de considérer la SOA comme la solution aux problèmes d'interface et d'intégration, mais elle offre bien plus.
- SOA est plus qu'un simple cadre d'intégration.
- SOA promeut une approche globale en délimitant le paysage commercial principal, en identifiant les unités commerciales et les parties prenantes concernées, puis en définissant ou en améliorant les processus commerciaux et en exposant les interfaces et les solutions en termes de services logiciels réutilisables, autonomes, interopérables et flexibles.
- La SOA impose également la mise en place de cycles opérationnels pour l'exécution de systèmes sains ainsi que pour les cycles de maintenance et de mise à niveau.
- Il active non seulement le cadre de registre et de découverte pour les services, mais fournit également la plate-forme de gouvernance des services logiciels qui, en fin de compte, surveille, signale et régit les fonctions commerciales pour les unités commerciales pertinentes d'une entreprise.

SOA Enjeux

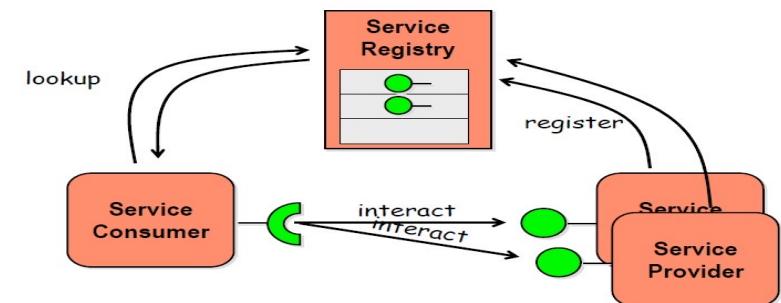
- Elle est devenue une solution incontournable pour gagner d'autres enjeux :
 - **Métier** : Produire des **SI ouverts** (à la fois interopérables + évolutifs)
 - Proposer une solution utilisant des **standards** et assurant un **couplage faible**
 - **Métier** : Réduire le **Time to Market** (le temps entre le besoin et la mise en production)
 - Proposer des moyens pour réduire le temps du cycle projet
 - **Technique** : **Fédérer les technologies**
 - Proposer des moyens pour rendre les solutions techniques réutilisables
 - **Financier** : Maîtriser les **coûts** et les **délais**
 - Coût important mais retour sur investissement à long terme

SOA Architecture

- Architecture orientée service (Service Oriented Architecture)
- Style d'**architecture distribuée** qui permet de fournir ou consommer un processus métier en tant que **service**
- Offre des services **réutilisables** et **interopérables** via des interfaces standards (construites autour de XML)
- Plusieurs partenaires peuvent communiquer et échanger des données dans le contexte de SOA **indépendamment des Plateformes et langages**

SOA Paradigme

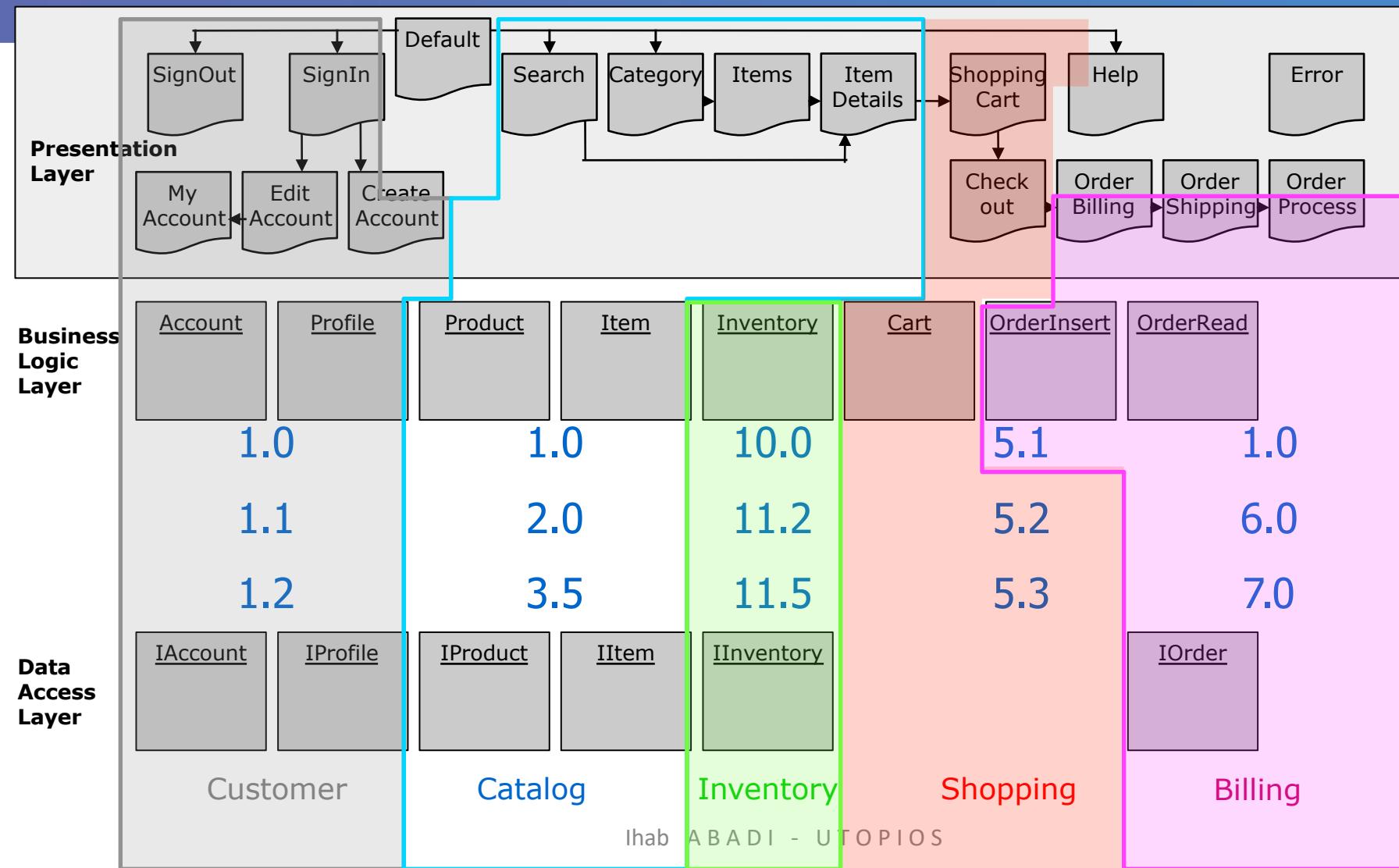
- **Fournisseur de service :**
 - **Fournit** un service accessible via une adresse
 - **publie** son contrat dans le registre de services
 - et **exécute** les requêtes des consommateurs (un **Proxy** et un cache peuvent être utilisés du côté consommateur pour délocaliser le traitement et réduire le nombre d'appels réseau)
- **Consommateur de service : application, service...**
 - **Cherche** le service dans le registre (son adresse)
 - Se **lie dynamiquement** au service (binding)
 - **Invoque** le service via une requête conforme au contrat
- **Registre de services : Annuaire des contrats de services**
 - Le **Contrat** décrit le **format d'échange** (format des requête/réponse, les pré et post conditions du service et sa QoS, ex: temps de réponse)
 - Le contrat est renouvelable par demande de nouveau bail à partir du registre



Le modèle en Couches de la SOA

- **Présentation** : renseigne les types de Clients (services Web, servlets ou pages JSP) des services
- **Orchestration** : assure la coordination des services composés et gère leur enchaînements
 - Services réutilisés pour organiser un processus métier, un workflow ou un flux de services
- **Services** : héberge et organise les services par domaine métier
- **Composants** : héberge les composants utilisés par les services pour assurer une fonctionnalité métier
- **OS/Données** : représente les sources de données (SGBD, CICS...), les EAI ou ERP déployés par l'entreprise

Le modèle en Couches de la SOA



Avantage SOA

■ Métier :

- Améliorer l'agilité et la flexibilité du métier (évolutivité)
- Réduire en temps le cycle de développement
- Faciliter la gestion des processus métier
- Améliorer le retour sur investissement

■ Techniques :

- Réduire la complexité de la solution
- Construire les services une seule fois et les utiliser fréquemment
- Garantir une intégration standardisée permettant de communiquer avec des clients hétérogènes
- Faciliter la maintenance

ESB

- Le bus de service d'entreprise ou ESB est un ensemble de fonctionnalités et d'outils.
- ESB un composant essentiel, plutôt une colonne vertébrale, pour toute implémentation SOA.

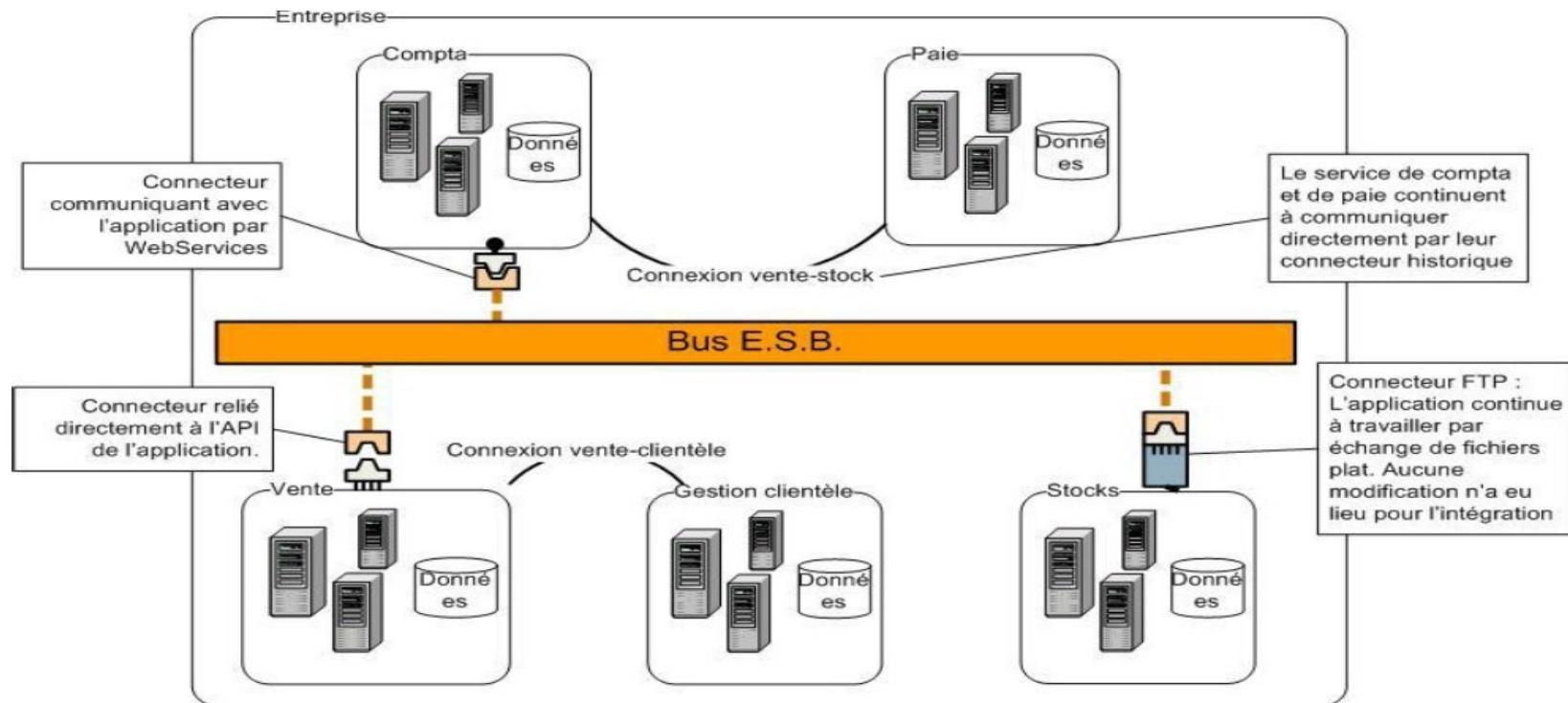
ESB – Fonctionnalités

- Storage resources (in memory, persistent, transfer)
- Gateway services (protocols, security)
- Message broker (as explained previously)
- Solution designer (IDE)
- Service adapters (data and services connectors to various resources including third parties)
- Management Interface (administration, installation of components, management, and monitoring)

ESB – Intérêts

- Il implémente une architecture distribuée et fournit des services de **transformation de données, routage, orchestration** de services, **sécurité, transaction et interopérabilité**
- Il permet en fait aux applications hétérogènes de communiquer de façon simple, mais **standardisée** à la différence des EAI basés sur une logique d'intégration propriétaire
- Il est donc recommandé pour **éviter le couplage fort** entre fournisseur et consommateur, mais il n'est pas obligatoire
- Il n'est pas, mais intègre un moteur d'orchestration

ESB – Exemple



Démo

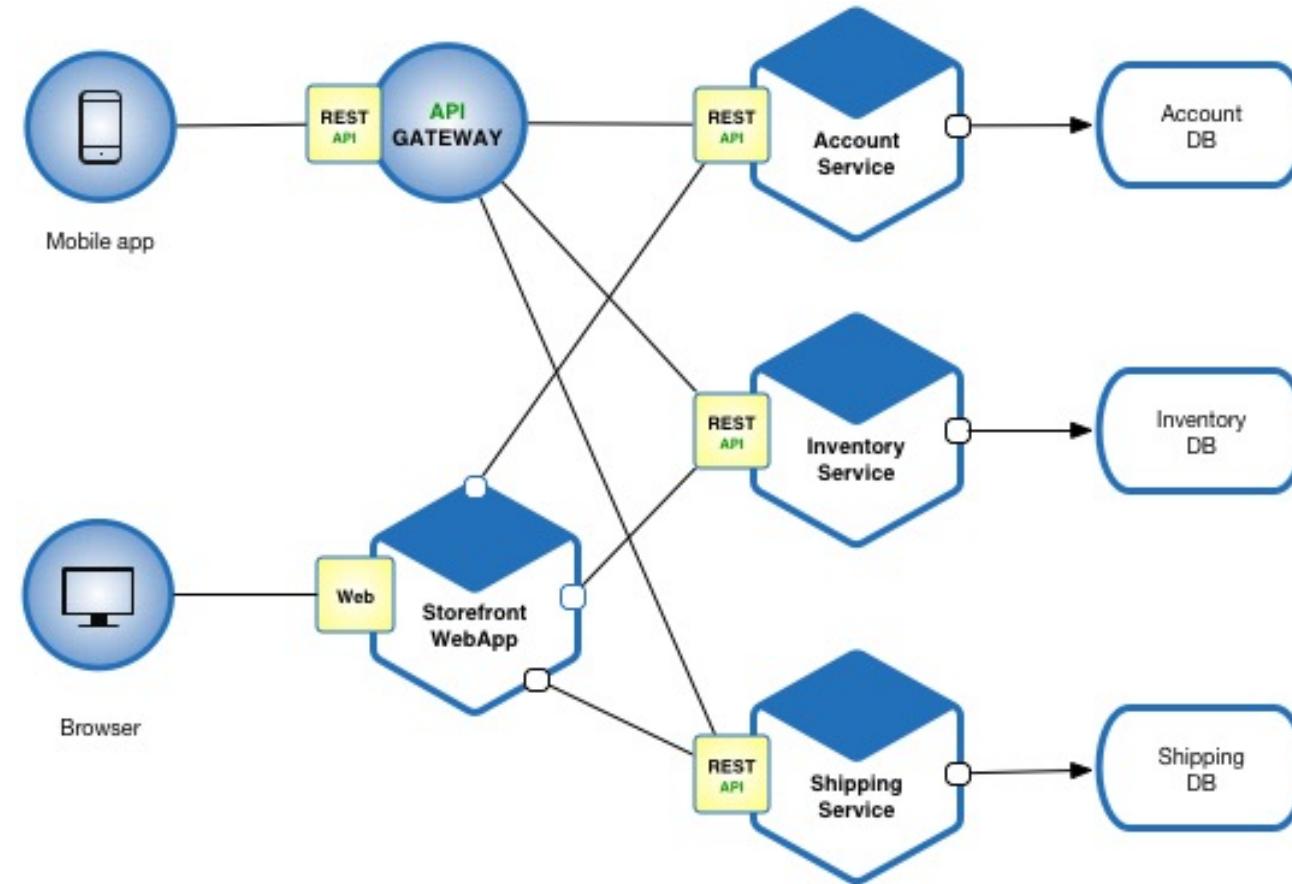
- Démo SOA
- Démo ESB

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

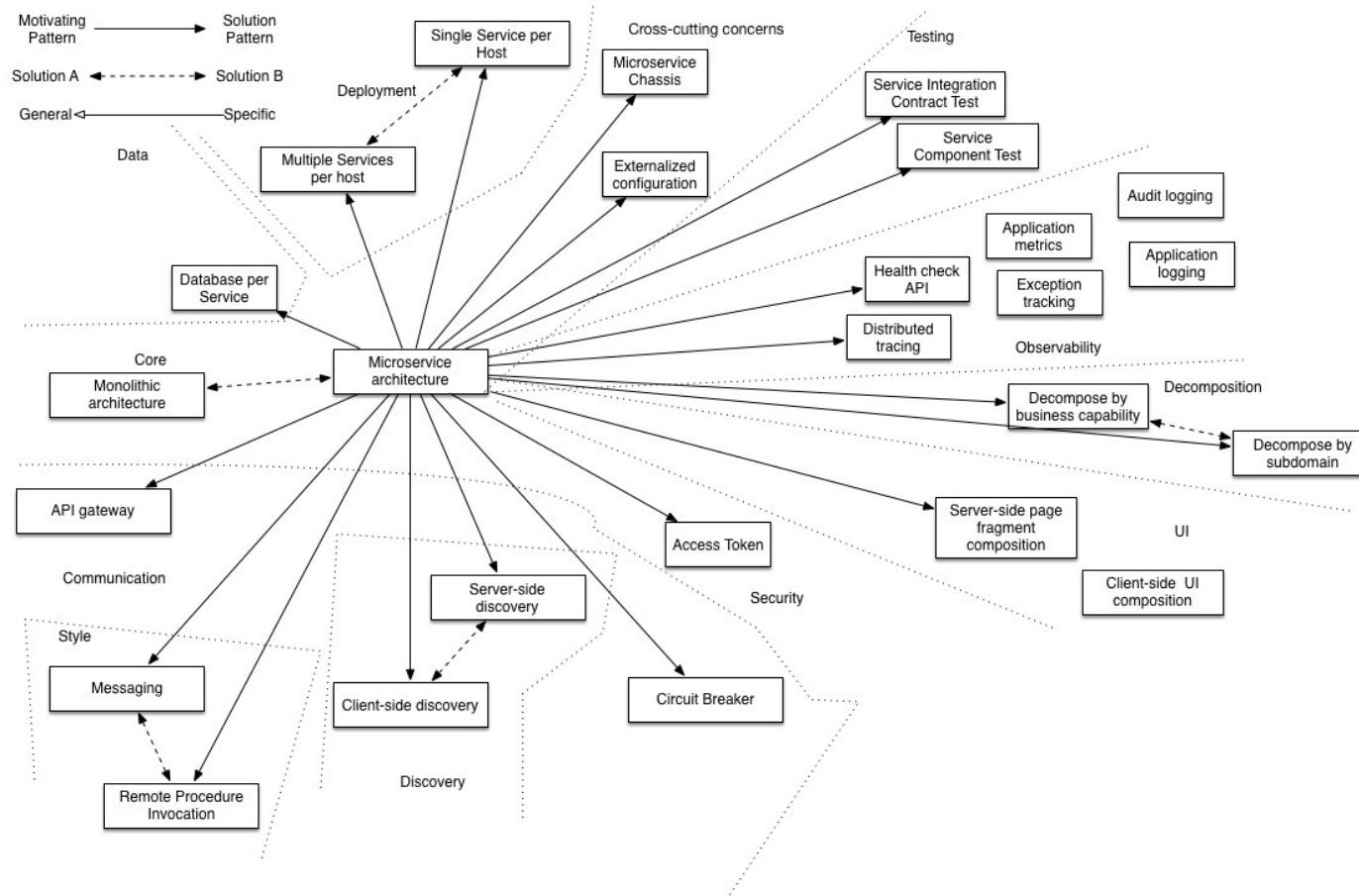
- Exemple site Ecommerce



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

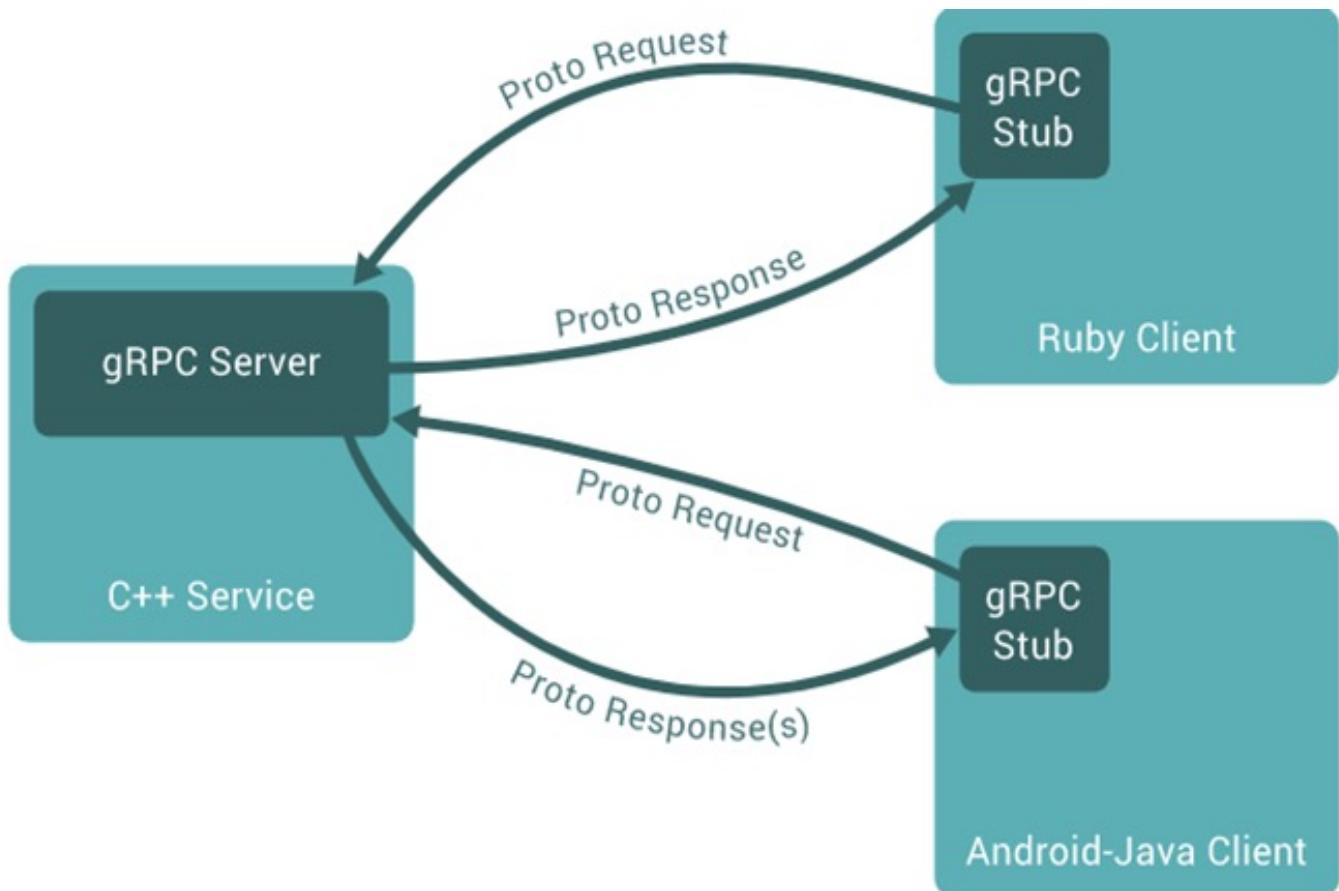
Communication Entre Microservice API REST

- 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

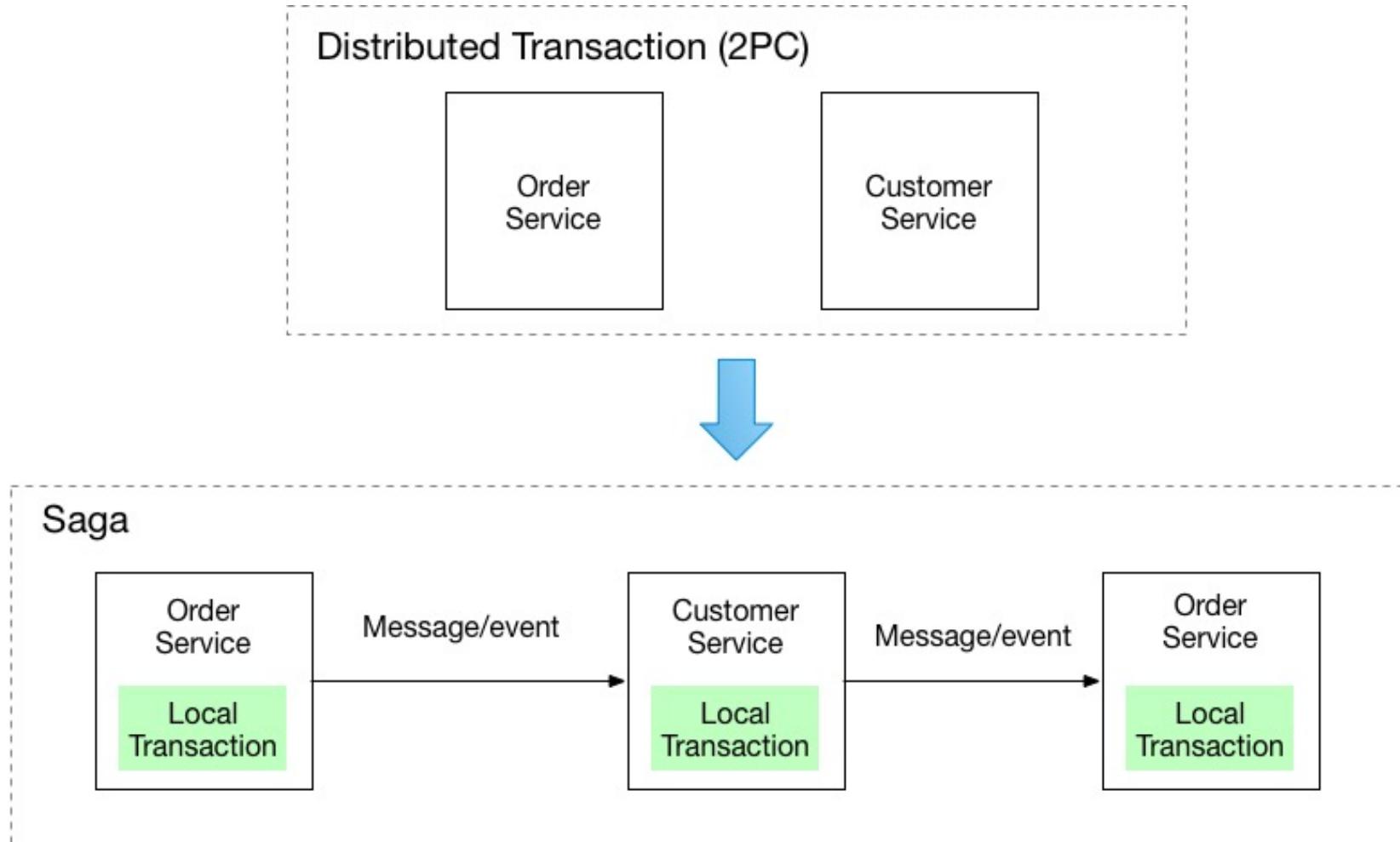
Gestion de base de données en Microservice

- 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

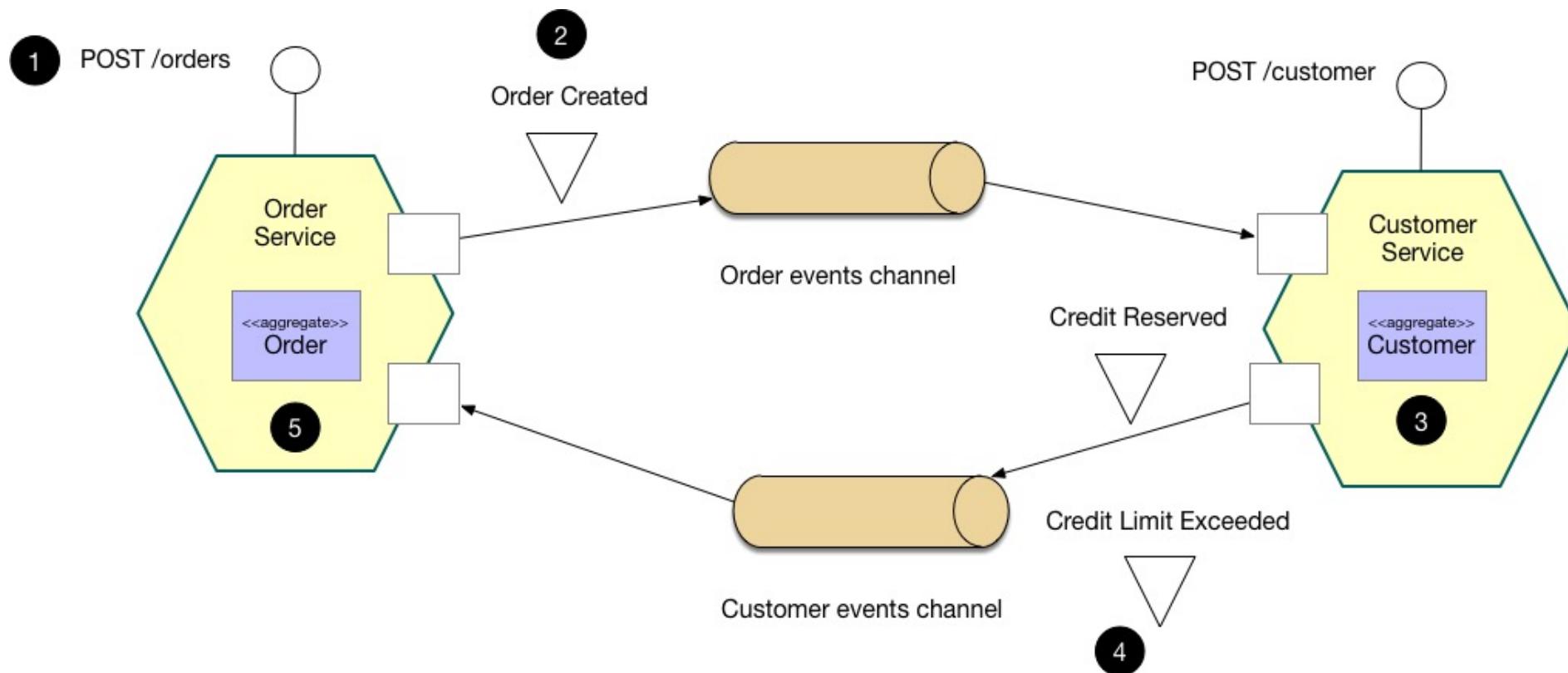
Microservice SAGA



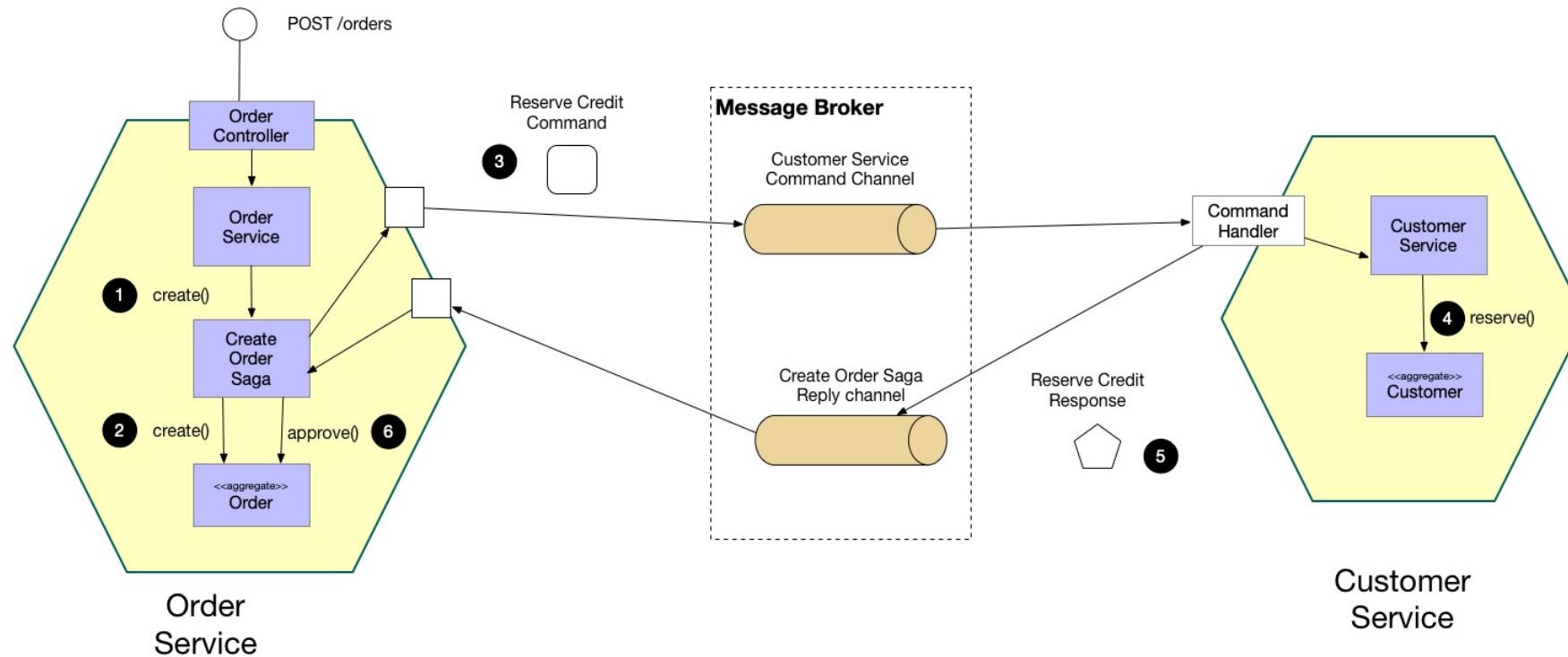
Microservice SAGA

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

Microservice SAGA Chorégraphique

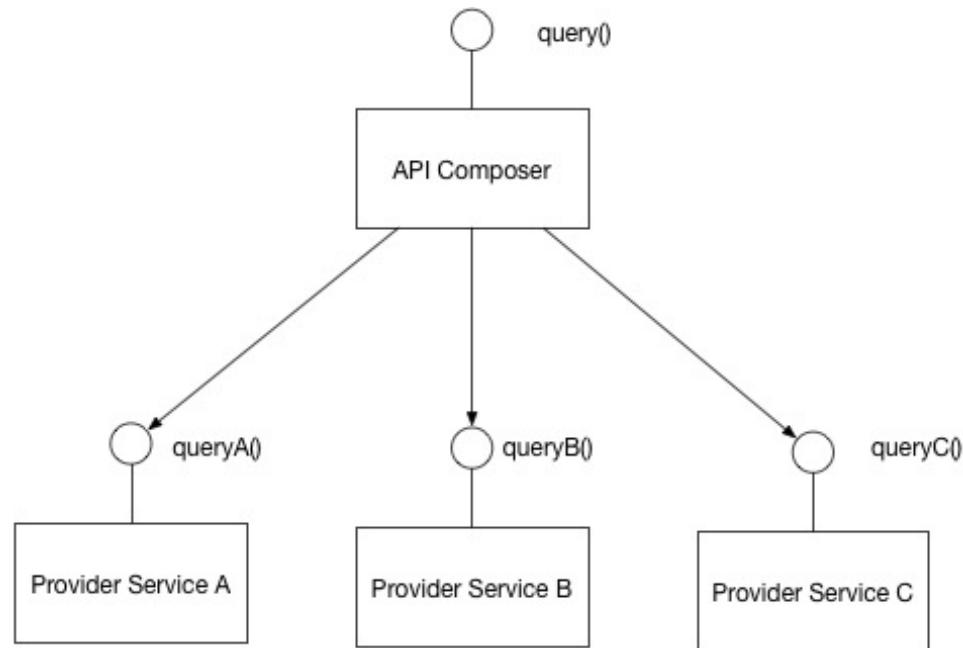


Microservice SAGA Orchestration



Requête cross services (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.

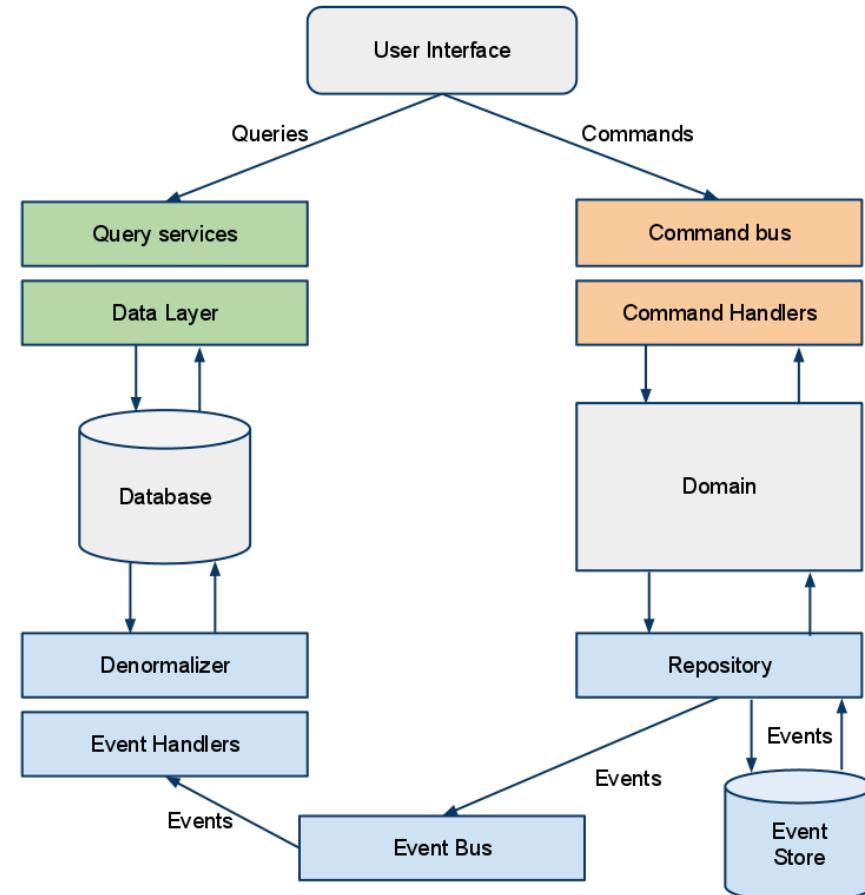


Requête cross services (API Composition)

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

Requête cross services (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

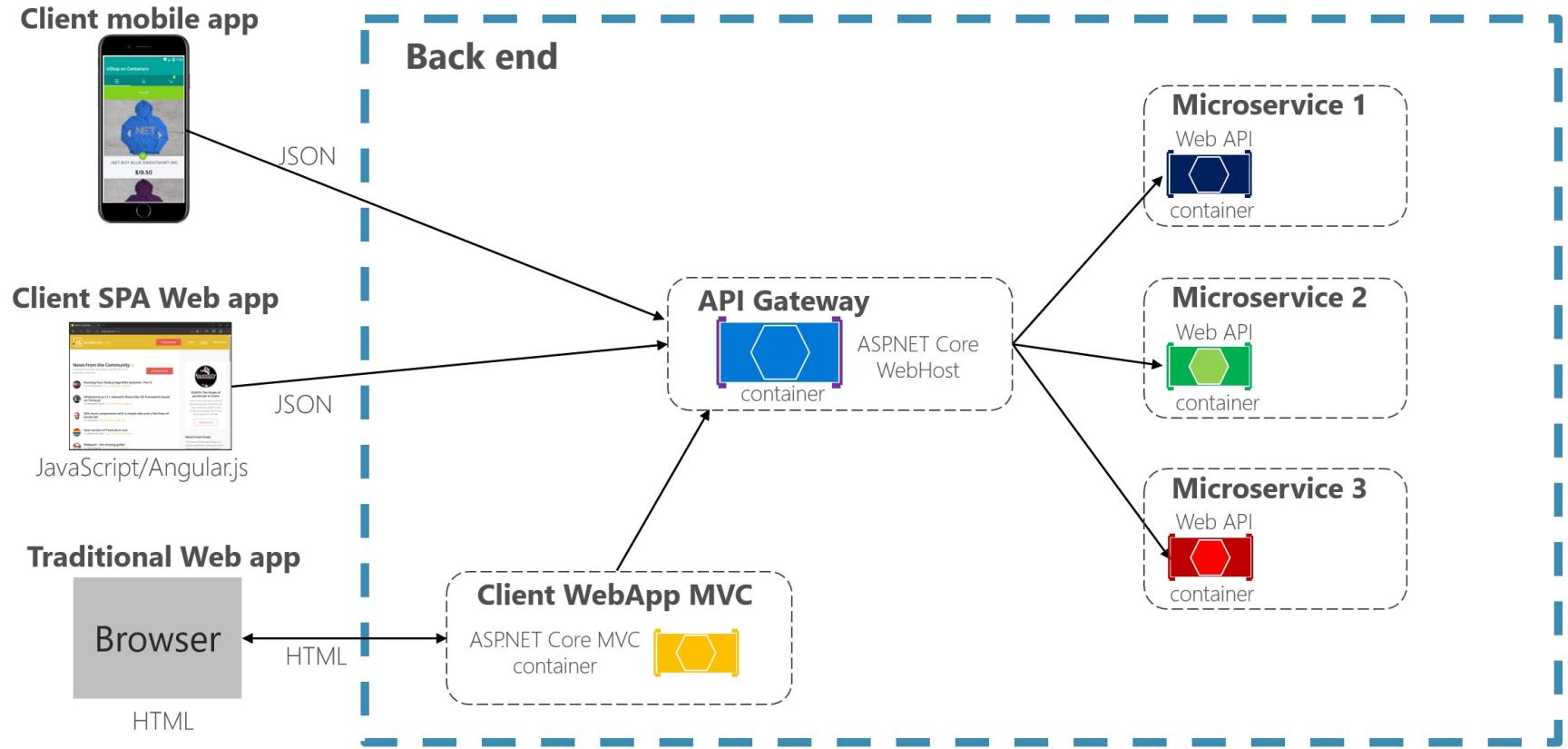
API GateWay

- 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

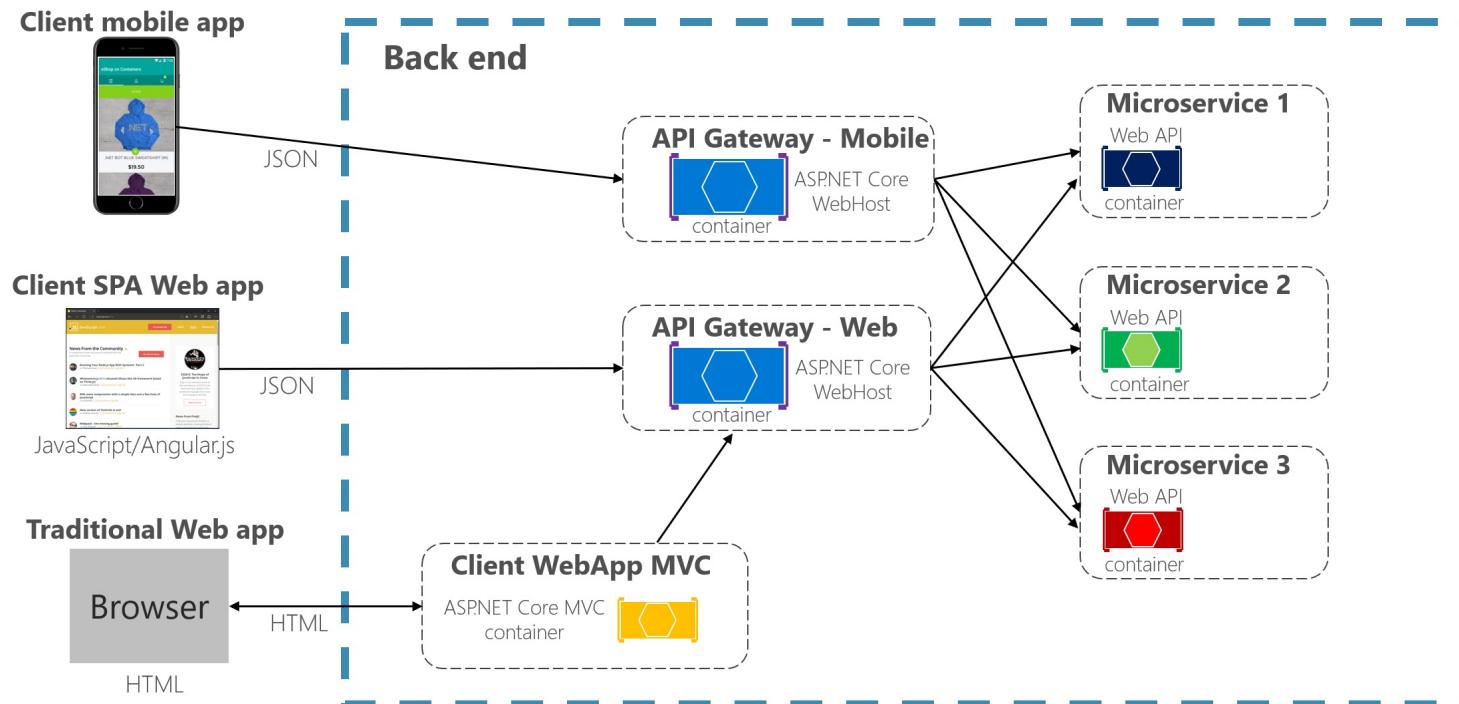
Using a single custom **API Gateway service**



API GateWay

- Exemple plusieurs gateWays

Using multiple **API Gateways / BFF**

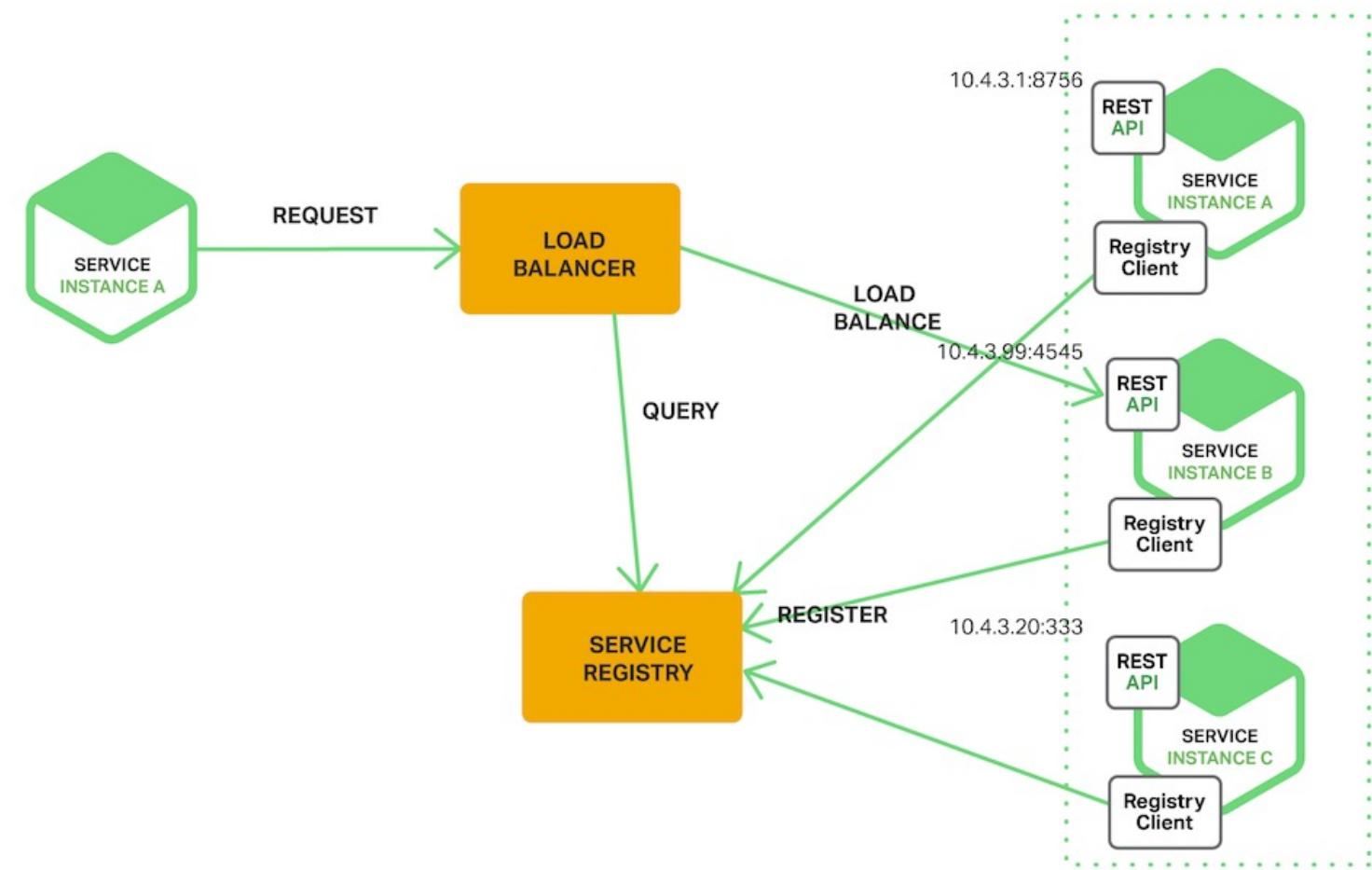


Service Discovery

- 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.

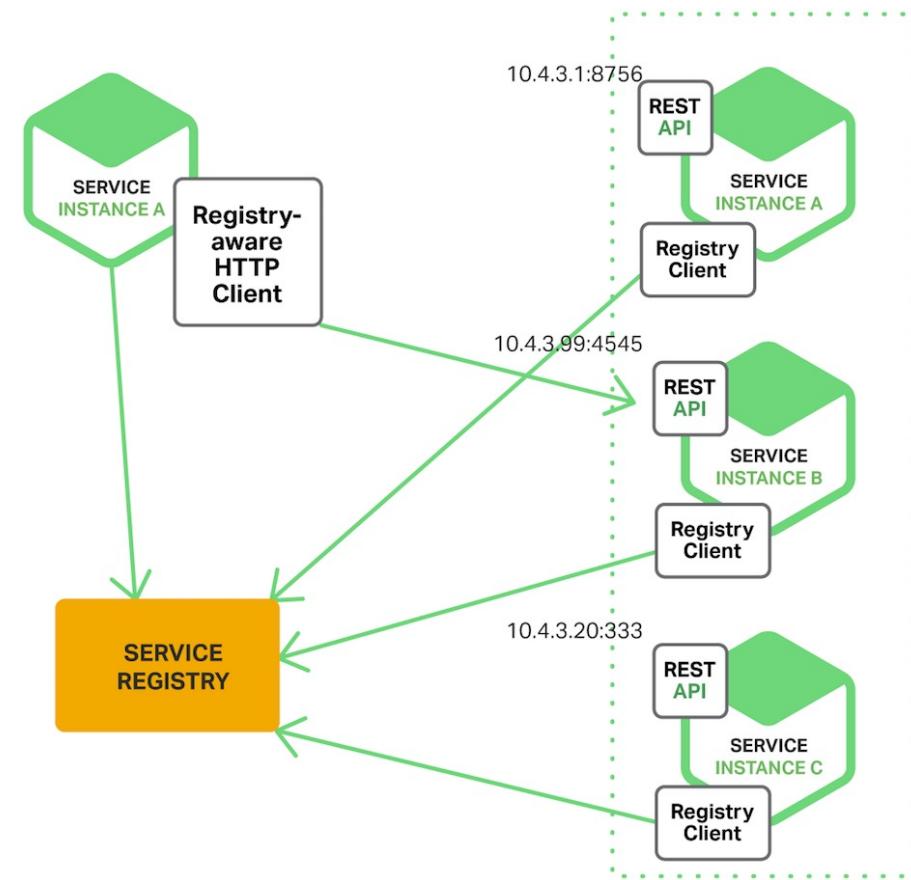
Service Discovery

- 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