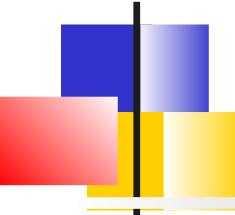


Design patterns pour les microservices

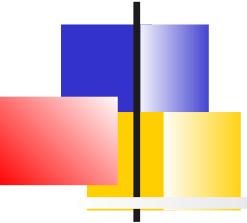
David THIBAU - 2021

david.thibau@gmail.com



Agenda

- **Introduction**
 - DevOps et Architecture
 - Inconvénients des monolithiques
 - Architecture des micro-services
 - Patterns et leurs relations
- **Décomposition**
 - Introduction
 - Business Capacity Pattern
 - Subdomain Pattern
 - Définition des APIs
- **Communication**
 - Introduction
 - Communication synchrone
 - Communication asynchrone
- **Cohérence des données et transaction**
 - Introduction
 - Saga Pattern
- **Design logique métier**
 - Introduction
 - Transactional Script Pattern
 - Patterns Orienté-Objet
 - Event Sourcing Pattern
- **Requêtage**
 - API Composition Pattern
 - CQRS Pattern
- **API Externe**
 - Gateway Pattern
- **Tests**
 - Introduction
 - Tests unitaires
 - Tests d'intégration
 - Tests de composants
 - Tests End To End
- **Vers la production**
 - Préparation
 - Infrastrcuture
 - Kubernetes et Istio



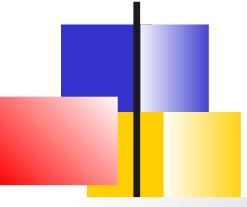
Introduction

DevOps et Architecture

Inconvénients du monolithique

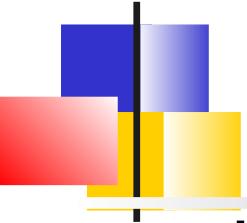
Architecture micro-services

Patterns et leurs relations



L'agilité

- Le terme agile (regroupant de nombreuses méthodes) est consacré par le manifeste Agile : <http://agilemanifesto.org/> 2001
 - Personnes et interactions plutôt que processus et outils
 - Logiciel fonctionnel plutôt que documentation complète
 - Collaboration avec le client plutôt que négociation de contrat
 - Réagir au changement plutôt que suivre un plan

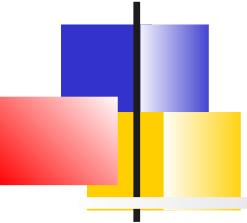


Contraintes sur la fréquence des déploiements

L'agilité suppose d'augmenter la fréquence des déploiements dans les différents environnements : intégration, recette, production afin :

- De mieux piloter le projet
- De prendre en compte rapidement les retours utilisateurs

Problème : Avant DevOps, l'organisation des services informatiques ne facilitait pas les déploiements

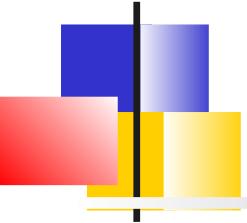


Avant DevOps : Disparité des objectifs

Avant DevOps, Ces environnements étaient traditionnellement gérés et utilisés par des équipes distinctes qui ... souvent communiquaient peu

De plus, les équipes avaient des objectifs différents et quelquefois contradictoires

- Développeur : Implémenter les fonctionnalités requises dans le temps imparti.
- Intégrateur/Testeur : Dimensionner l'architecture pour atteindre des SLA, Valider fonctionnellement le système dans des scénarios pas toujours anticipés par les dev.
- Opérations : Garantir la stabilité du système et des infrastructures

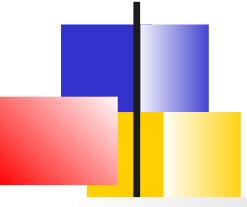


Le constat DevOps

Les différents objectifs donnés à des équipes qui se parlent peu créent des tensions et des dysfonctionnements dans le processus de mise en production d'un logiciel.

=> Pour l'équipe Ops, l'équipe de développement devient responsable des problèmes de qualité du code et des incidents survenus en production.

=> L'équipe Dev blâme son alter ego Ops pour sa lenteur, les retards et leur méconnaissance des livrables qu'elle manipule



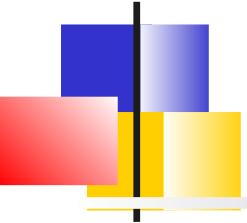
Approche *DevOps*

DevOps vise l'alignement des équipes par la réunion des "Dev engineers" et des "Ops engineers".

Cela impose :

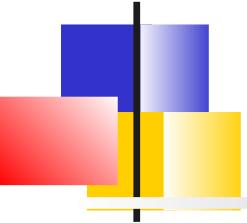
- la réunion des équipes
- la montée en compétence des différents profils.

Plus génériquement, *DevOps* signifie le regroupement de toutes les compétences nécessaires à un projet en une équipe complètement indépendante



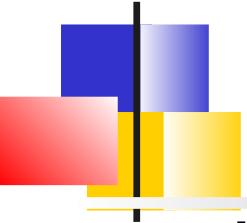
Pratiques *DevOps* (1)

- Un déploiement régulier des applications dans les différents environnements.
La seule répétition contribuant à fiabiliser le processus ;
- Un décalage des tests "vers la gauche".
Autrement dit de tester au plus tôt ;
- Une pratique des tests dans un environnement similaire à celui de production ;
- Une intégration continue incluant des "tests continus" (à chaque commit);



Pratiques *DevOps* (2)

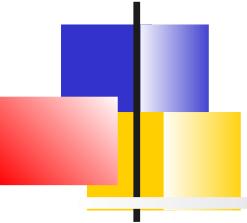
- Une boucle d'amélioration courte
i.e. un feed-back rapide des utilisateurs ;
- Une surveillance étroite de l'exploitation et de la qualité de production factualisée par des métriques et indicateurs "clé".
- Une unique source de vérité : Les configurations des différents environnements, des builds, des tests sont centralisées dans le même SCM que le code source



« As Code »

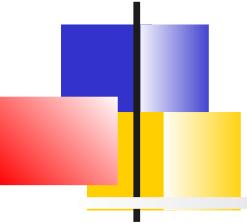
Les outils *DevOps* permettent d'automatiser/exécuter toutes les tâches nécessaires à la construction d'un logiciel de qualité (build, test, provisionnement) à partir de l'unique point central de vérité

On parle alors de *Build As Code*,
Infrastructure As Code, *Pipeline As Code*,
Load Test As Code, ...



Objectif ultime

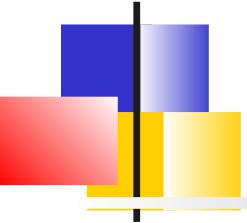
- Déployer souvent et rapidement
- Automatisation complète
- Zero-downtime des services
- Possibilité d'effectuer des roll-backs
- Fiabilité constante de tous les environnements
- Possibilité de scaler sans effort
- Créer des systèmes auto-correctifs, capable de se reprendre en cas de défaillance ou erreurs



Introduction

Avec DevOps une nouvelle architecture de systèmes visant à améliorer la rapidité des déploiements des retours utilisateur est apparu : les « **micro-services** »

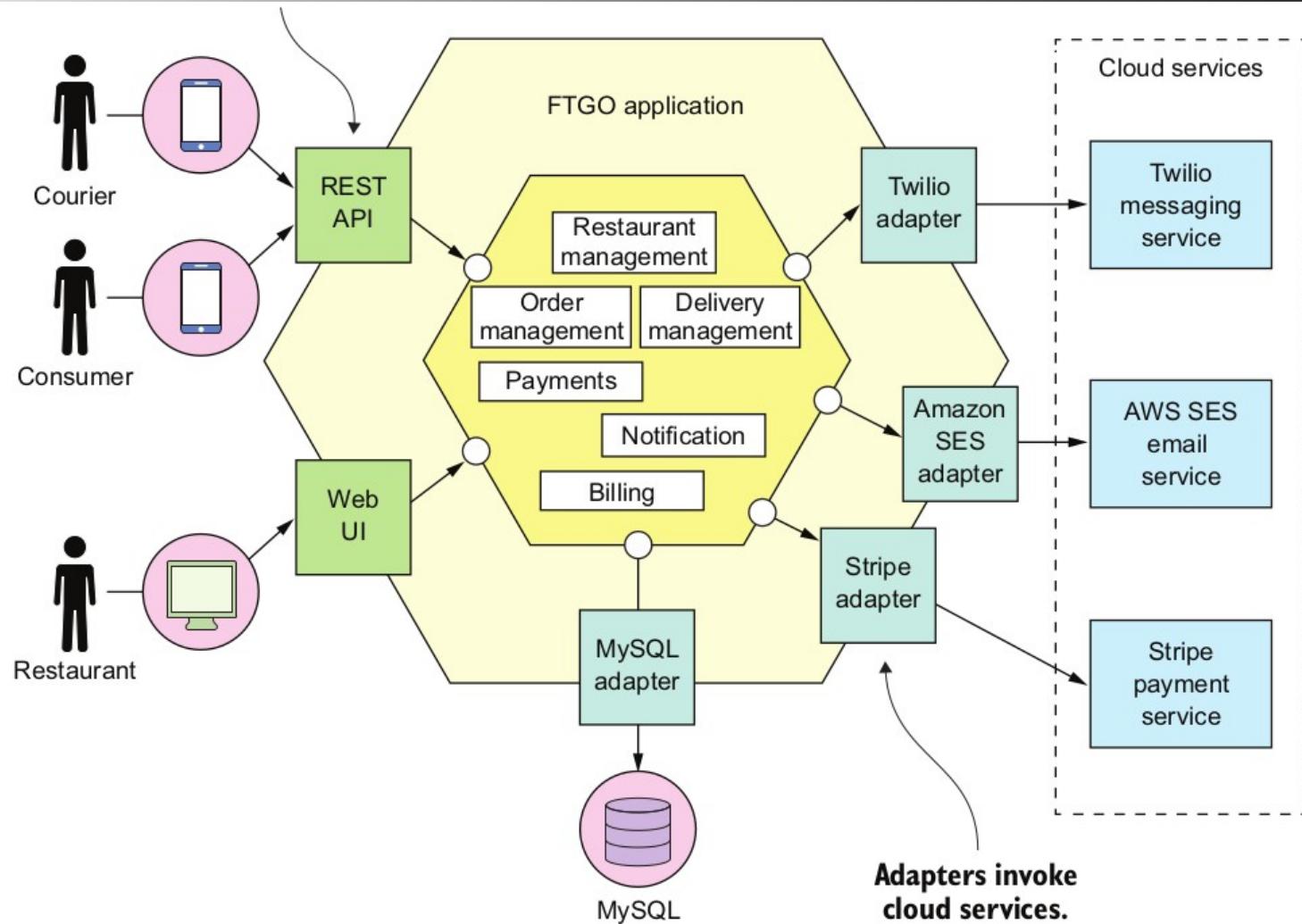
C'est le même objectif que l'approche *DevOps* : « *Déployer plus souvent* »

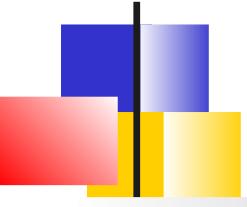


Introduction

DevOps et Architecture
Inconvénients du monolithique
Architecture micro-services
Patterns et leurs relations

Architecture Hexagonale

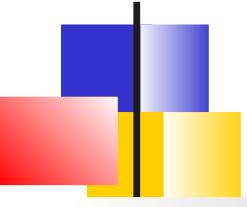




Modularité

La logique métier se compose de modules, chacun étant une collection d'objets du domaine

Malgré une architecture logiquement modulaire, l'application est packagée en un seul exécutable (Ex : un .war)



Bénéfices du départ

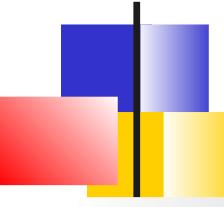
Simple à développer : les IDE et autres outils de développement se concentrent sur la création d'une seule application.

Modifications faciles à apporter à l'application : vous pouvez modifier le code et le schéma de la base de données, créer et déployer.

Test simple : les développeurs ont écrit des tests de bout en bout qui ont lancé l'application, invoqué l'API REST et testé l'interface utilisateur avec Selenium.

Déploiement simple : Copier le fichier WAR sur un serveur sur lequel Tomcat était installé.

Facile à scaler : Exécuter plusieurs instances de l'application derrière un équilibrEUR de charge.



Au bout d'un certain temps

Le rythme de livraison des logiciels ralentit.

Les maintenance corrective et évolutive deviennent difficiles à cause du volume et de la complexité du code.

L'application met du temps à démarrer.

La boucle edit-build-run-test prend également du temps et impacte la productivité.

Les déploiements vers la production sont longs et pénibles.

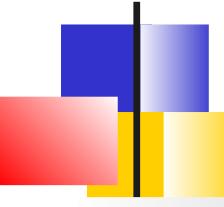
Les mises à jour sont donc rares.

La grosseur de l'équipe pose problème.

De nombreux développeurs commettent dans le même dépôt, le code est fréquemment dans un état instable.

Les fusions des branches de feature sont pénibles et nécessite une longue période de test et de stabilisation.

Les tests automatisés sont également longs



Au bout d'un certain temps

Manque d'isolation des fautes

Tous les modules s'exécutent dans le même processus.

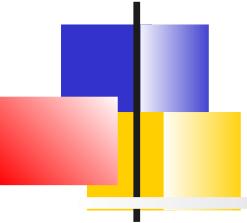
De temps en temps, un bug dans un module fait planter toutes les instances de l'application, une par une.

La pile technologique est de plus en plus obsolète.

L'architecture monolithique rend difficile l'adoption de nouveaux frameworks et langages.

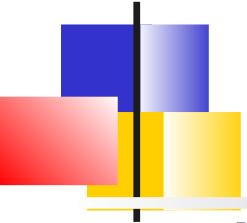
Trop coûteux et risqué de réécrire l'intégralité de l'application monolithique

Impossible expérimenter des nouveaux langages



Introduction

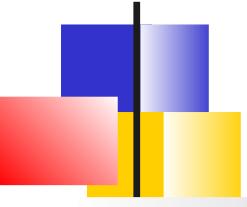
DevOps et Architecture
Inconvénients du monolithique
Architecture micro-services
Patterns et leurs relations



Architecture

Une architecture micro-services implique la décomposition des applications en très petits services

- faiblement couplés
- ayant une seule responsabilité
- Développés par des équipes full-stack indépendantes.



Bénéfices attendus

CI/CD d'applications complexes

Scaling indépendant : Seuls les services les plus sollicités sont scalés
=> Économie de ressources

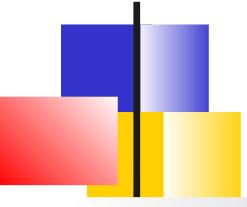
Mise à jour indépendantes : Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes
=> Agilité de déploiement

Maintenance facilitée : Les services sont plus petits
=> Corrections, évolutions plus rapide

Hétérogénéité des langages : Utilisation des langages les plus appropriés pour une fonctionnalité donnée

Isolation des fautes : Un dysfonctionnement peut être plus facilement localiser et isoler.

Equipe DevOps autonome : Full-stack team, Intra-Communication renforcée
=> Favorise le partage et les montées en compétences



Caractéristiques

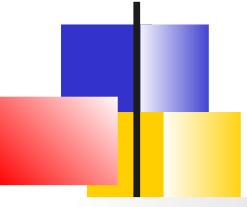
Design piloté par le métier : La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)

Principe de la responsabilité unique : Chaque service est responsable d'une seule fonctionnalité et la fait bien !

Une interface explicitement publiée : Un producteur de service publie une interface qui peut être consommée

**DURS (Deploy, Update, Replace, Scale)
indépendants** : Chaque service métier peut être indépendamment déployé, mis à jour, remplacé, scalé

Communication légère : REST sur HTTP, STOMP sur WebSocket,



Contraintes

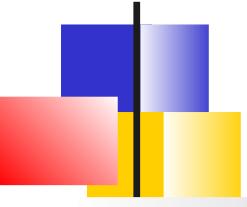
RéPLICATION : Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)

Découverte automatique : Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certains métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

Monitoring : Les services sont surveillés en permanence. Des traces sont générées et éventuellement agrégées

Résilience : Les services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

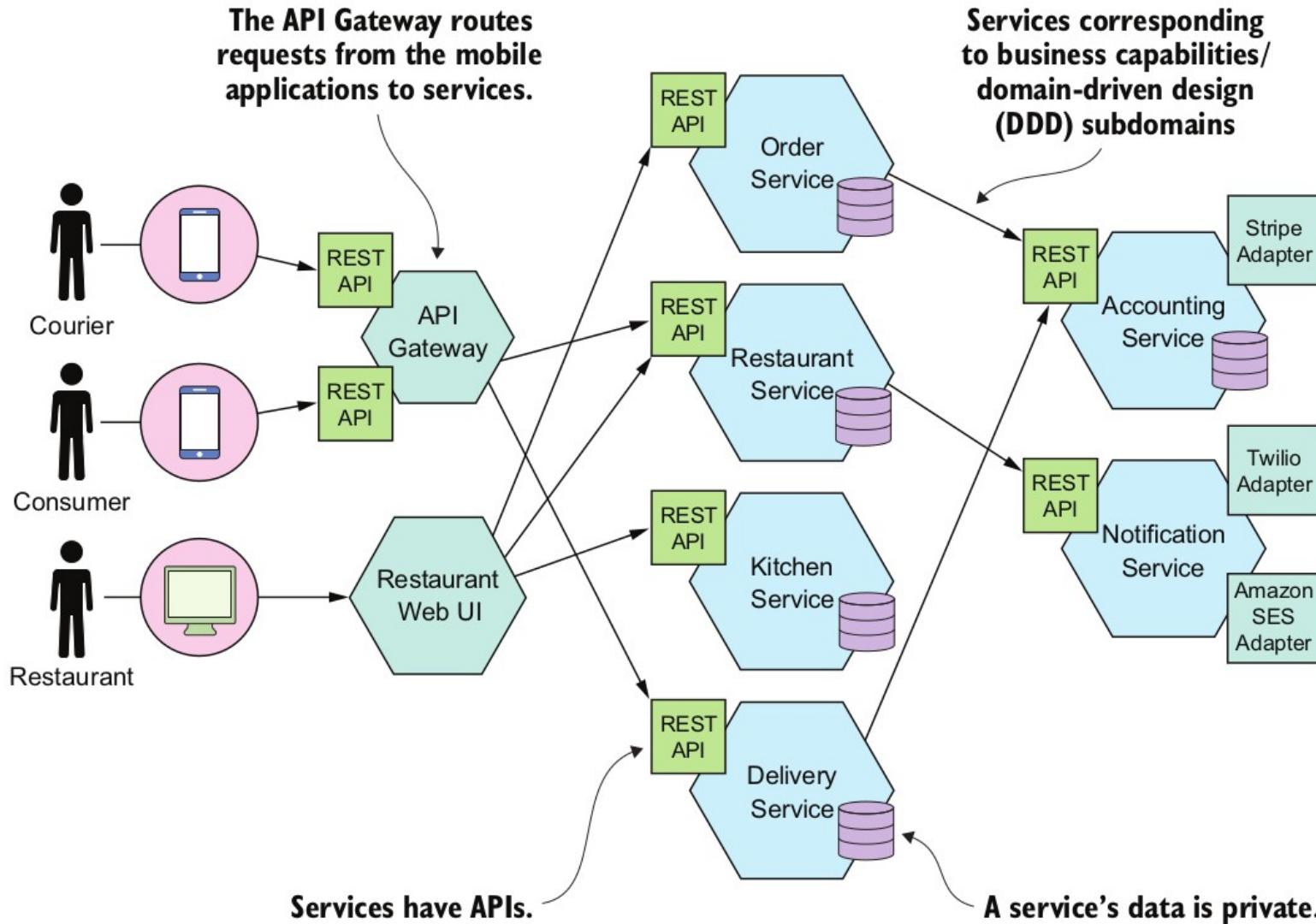
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



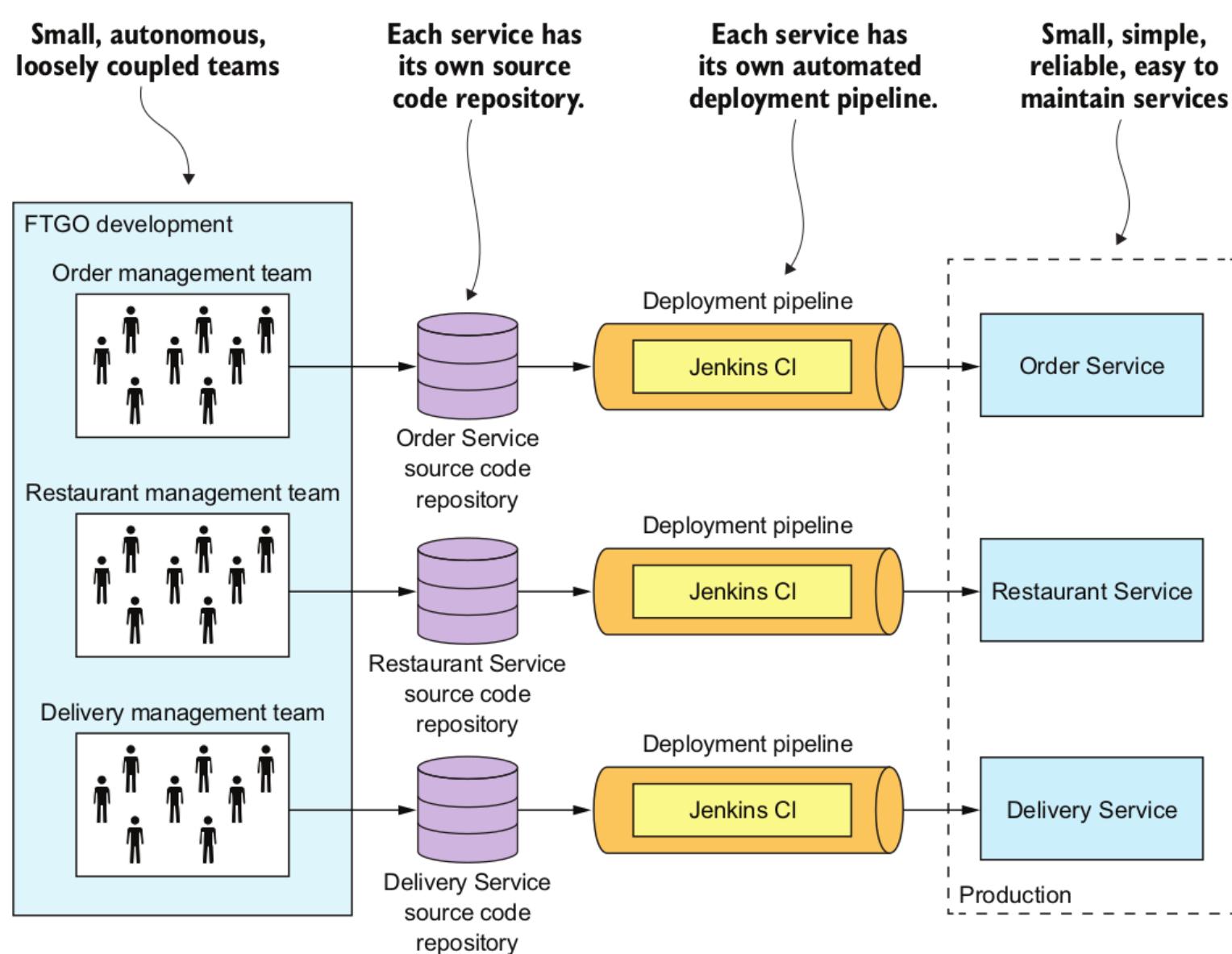
Inconvénients et difficultés

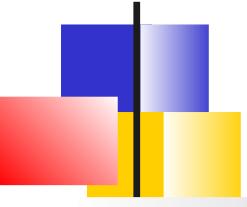
- Trouver la bonne décomposition est difficile.
Une mauvaise décomposition peut entraîner des couplages entre les micro-services
- Le côté distribué fait que le système complet est plus difficile à tester, déployer
- Le déploiement de fonctionnalités qui touche plusieurs services est plus délicat
- La migration d'une application monolithique existante vers les micro-services n'est pas simple

Une architecture micro-service



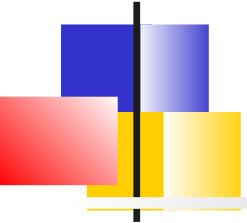
Organisation DevOps





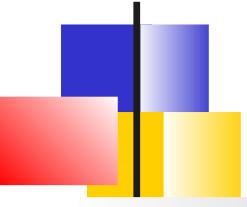
Les 12 facteurs de réussite

- I. Outil de scm** : Unique source de vérité
- II. Dépendances**: Déclare explicitement et isole les dépendances du code source
- III. Configuration** : Configuration séparée du code
- IV. Services** d'appui (backend) : Considère les services d'appui comme des ressources attachées, possibilité de switcher sans modification de code
- V. Build, release, run** : Permet la coexistence de différentes releases en production
- VI. Processes** : Exécute l'application comme un ou plusieurs processus stateless.
Déploiement immuable
- VII. Port binding** : Application est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP
- VIII. Concurrence** : Montée en charge grâce au modèle de processus
- IX. Disposability** : Renforce la robustesse avec des démarrages et arrêts rapides
- X. Dev/prod parity** : Garder les environnements de développement, de pré-production et de production aussi similaires que possible
- XI. Logs** : Traiter les traces comme un flux d'événements
- XII. Processus d'Admin** : Considérer les tâches d'administration comme un processus parmi d'autres



Introduction

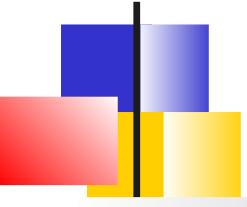
DevOps et Architecture
Inconvénients du monolithique
Architecture micro-services
Patterns et leurs relations



Design patterns

Un **patron de conception** (plus souvent appelé **design pattern**) est un arrangement caractéristique de modules ou classes, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

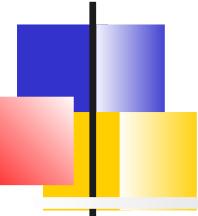
- Il décrit une solution standard, utilisable dans la conception de différents logiciels.
- Il est issu de l'expérience des concepteurs de logiciels.
- Il décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées en fonction des besoins.
- Ils ont une influence sur l'architecture logicielle d'un système informatique.



Formalisme

La description d'un patron de conception suit un formalisme fixe :

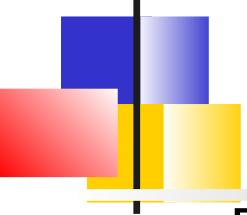
- Nom : Permettant de communiquer entre concepteurs
- Description du problème à résoudre
- Description de la solution : les éléments de la solution, avec leurs relations. i.e. diagramme de classes
- Conséquences : résultats issus de la solution.
- Patterns associés : Les patterns qui s'adressent à la même problématique



Patterns micro-service

Les patterns concernant les architectures micro-services peuvent être découpés en 3 couches :

- **Pattern d'infrastructure** : Problématique en dehors du développement concernant l'infrastructure d'exécution des systèmes distribués
- **Pattern applicatif d'infrastructure** : Problématique d'infrastructure qui impacte le développement.
(Par exemple quel mode de communication offre un message broker)
- **Pattern applicatif** : Problématique purement de développement.



Problèmes à résoudre et design patterns

Décomposition en services, Patterns :

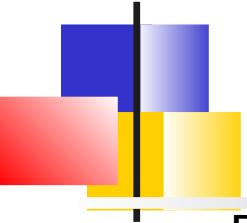
- DDD ou sous-domaines
- Fonctionnalités métier

Communication entre service, Aspects et patterns:

- Style (RPC, Asynchrone, etc.)
- Découverte des services, (Self-registry pattern, ...)
- Fiabilité : Circuit Breaker Pattern
- Messagerie transactionnelle
- APIs

Distribution des données, Aspects

- Gestion des transactions : Transactions distribuées ?
- Requêtes avec jointures ?



Patterns et problèmes à résoudre

Déploiement des services, Patterns :

- Hôtes uniques avec différents processus
- Un container par service, Déploiements immuables, Orchestration de Containers
- Serverless

Observabilité afin de fournir des *insights* applicatifs :

- Health check API, Agrégation des traces, Tracing distribué, Détection d'exceptions, Métriques applicatifs, Audit

Tests automatisés :

- Service en isolation, Tests des contrats (APIs)

Patterns transverses :

- Externalisation des configurations, Pipelines CD, ...

Sécurité :

- Jetons d'accès, oAuth, ...

Key

Predecessor	→ Successor
Alternative A	↔ Alternative B
General	← Specific
Problem area	

Application patterns

Decomposition

Database architecture
Querying

Maintaining data consistency

Testing

Monolithic architecture

Microservice architecture

Application architecture

Application infrastructure patterns

Cross-cutting concerns

Security

Transactional messaging

Communication style

Reliability

Observability

Infrastructure patterns

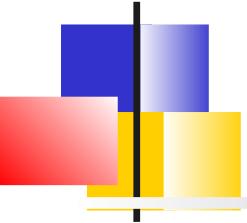
Deployment

Discovery

External API

Communication patterns

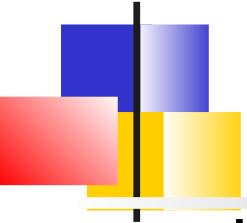
Microservice patterns



Stratégies de décomposition

Introduction

Business capability Pattern
Subdomain Pattern
Définition de l'API



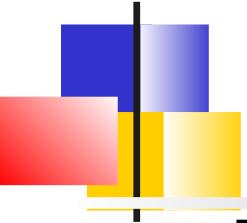
Indépendance des micro-services

Les micro-services collaborent seulement via l'APIs, pas par la BD

Les librairies partagées, même si elles réduisent la duplication de code, ne doivent pas introduire de couplage entre les services et sont donc rarement utilisées.

L'objectif à garder :

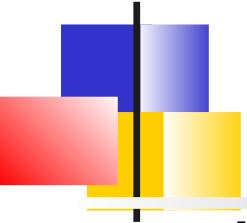
Capable de développer chaque service par une petite équipe avec un délai d'exécution minimal et avec une collaboration minimale avec les autres équipes.



Processus de définition des services

Processus itératif à 3 étapes :

- Transformer la demande métier en opérations système (requête) de 2 types :
 - Des commandes => Écriture
 - Des requêtes => Lecture de données
- Décomposer en services. 2 stratégies organisées autour de concepts métier plutôt que techniques
 - Selon les capacités métier
 - Selon les sous-domaines d'une approche DDD
- Déterminer l'API pour chaque service



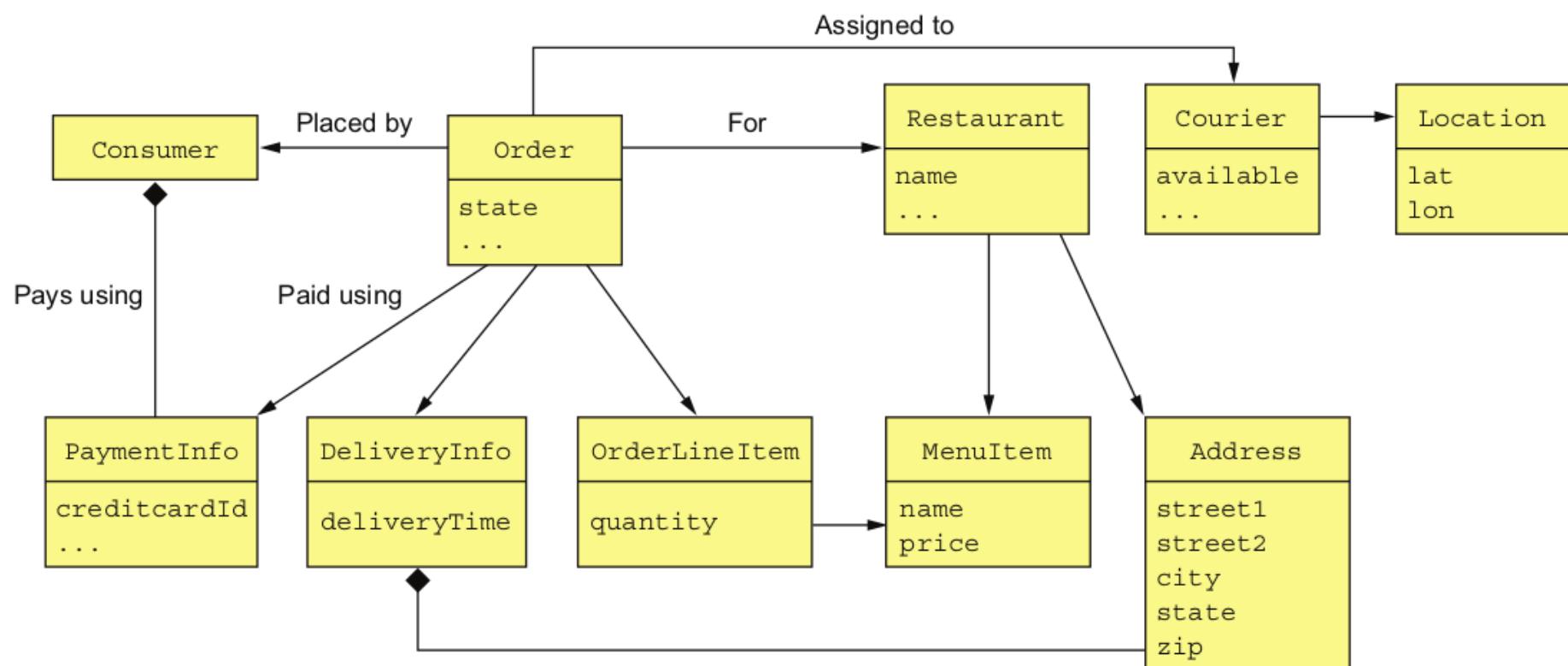
Modèle du domaine de Haut-niveau

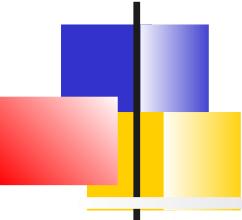
La première étape du processus de définition des opérations consiste à esquisser un modèle de domaine de haut niveau

Le modèle est utile pour définir le vocabulaire utilisé pour décrire le comportement des opérations système.

Il est créé à l'aide de UserStories et les réunions avec le métier

Exemple

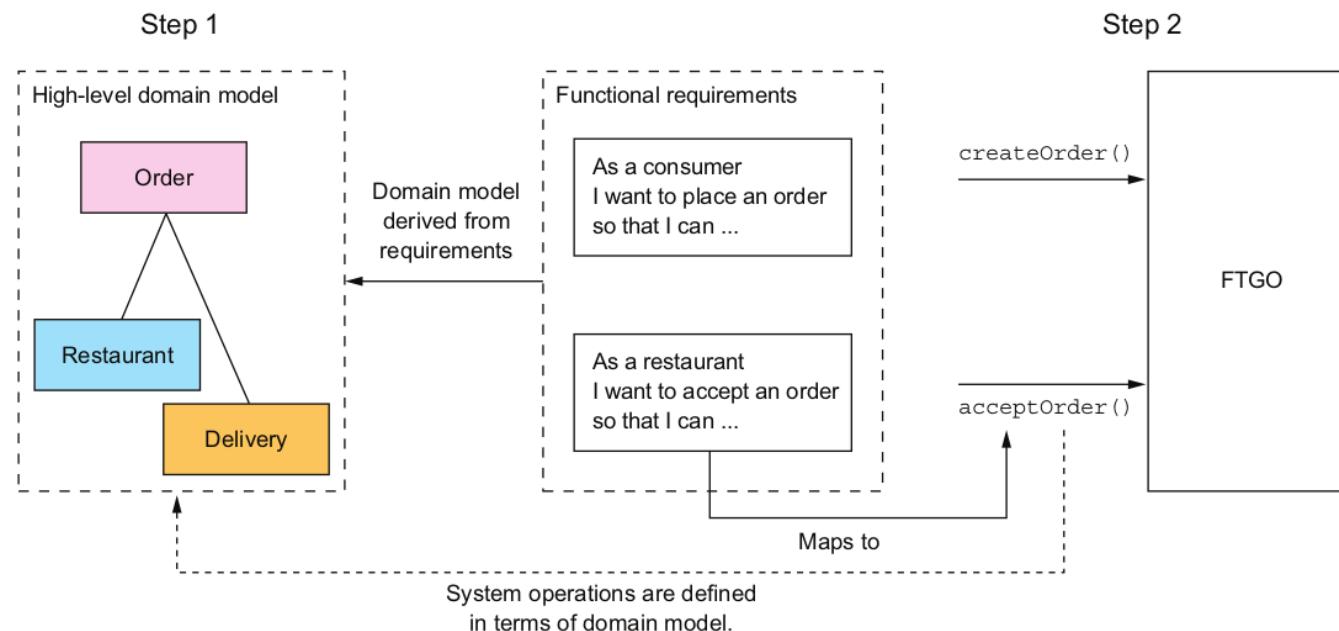


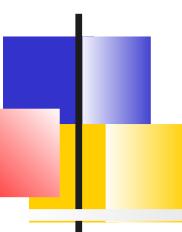


Exemple Opération clé : Commandes

Les opérations systèmes sont également déduite de l'analyse des UserStory

De nombreuses UserStory sont mappées directement à des commandes.



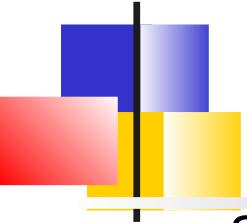


Spécification des commandes

Une commande a une spécification qui définit :

- ses paramètres d'entrée
- sa valeur de retour
- son comportement en termes de classes de modèle de domaine.
- Les préconditions lorsque l'opération est appelée
- Les post-conditions après que l'opération soit appelée

Given/When/Then du UserStory, puis du test



Exemple

Signature :

createOrder (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)

Retour :

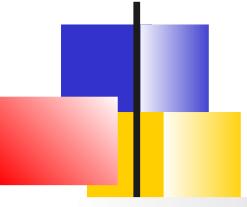
orderId,

Pré-conditions :

- Le consommateur existe et peut effectuer des commandes.
- Les lignes de la commande correspondent aux entrées du menu
- L'adresse et l'heure de livraison peuvent être assurées par le restaurant

Post-conditions :

- La carte de crédit du consommateur était autorisée pour ce montant
- Une commande est créée dans l'état PENDING_ACCEPTANCE

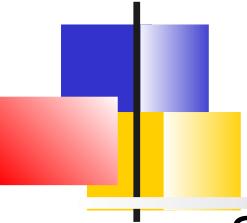


Requêtes

Une application doit également implémenter des requêtes.

Les requêtes fournissent à l'utilisateur le moyen de prendre des décisions.

De la même façon, l'identification des requêtes se fera à la lecture des UserStory



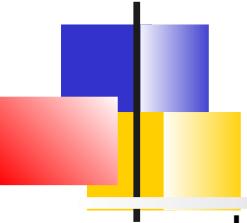
Exemple

Scénario :

1. L'utilisateur entre une heure et adresse de livraison
2. Le système affiche les restaurants disponibles
3. L'utilisateur sélectionne un restaurant
4. Le système affiche le menu
5. L'utilisateur sélectionne des plats
6. Le système crée une commande

Le scénario précédent implique les requêtes :

- *findAvailableRestaurants(deliveryAddress, deliveryTime)*
- *findRestaurantMenu(id)*



Force des patterns de décomposition

L'architecture doit être stable

Les services doivent être cohérents. Un service doit implémenter un petit ensemble de fonctions fortement liées.

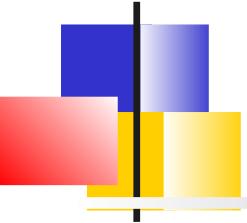
Les services doivent être conformes au *Common Closure Principle* : garantir que chaque changement n'affecte qu'un seul service

Les services doivent être faiblement couplés - chaque service a une API.
L'implémentation peut être modifiée sans affecter les clients

Un service doit être testable

Chaque service doit être suffisamment petit pour être développé par une équipe de 6 à 10 personnes

Chaque équipe qui possède un ou plusieurs services doit être autonome. Une équipe doit être capable de développer et de déployer ses services avec une collaboration minimale avec les autres équipes.



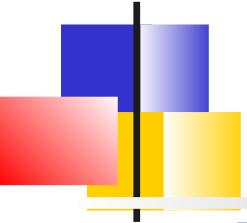
Stratégies de décomposition

Introduction

Business capability Pattern

Subdomain Pattern

Définition de l'API

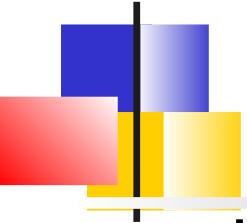


Pattern

Decompose by business capability

Pattern¹ : Définir des services correspondant aux capacités métier de l'entreprise

Une capacité métier est une activité de l'entreprise destinée à générer de la valeur.



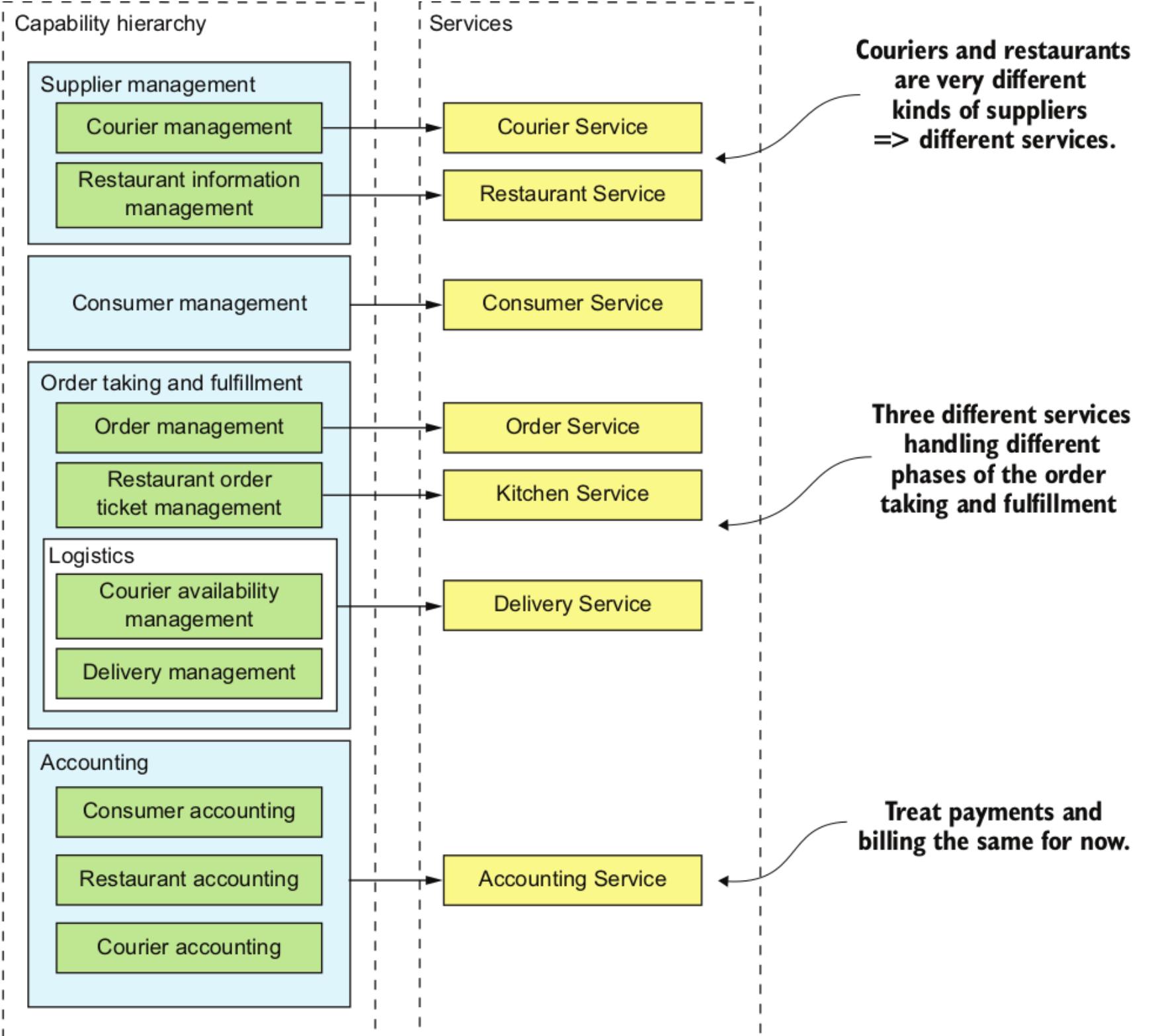
Décomposition par Capacité business

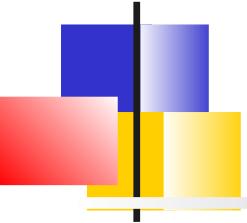
Les capacités métier capturent ce qu'est l'essence d'une organisation et sont donc généralement stables (contrairement à la façon dont une organisation mène ces activités).

Chaque capacité métier donne lieu à un service orienté métier (et non pas technique).

Sa spécification se compose de divers composants, y compris les entrées, les sorties et les SLA.

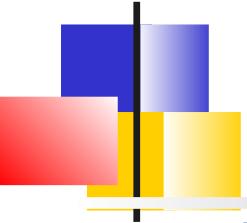
Une capacité peut souvent être décomposée en sous-capacités.





Stratégies de décomposition

Introduction
Business capability Pattern
Subdomain Pattern
Définition de l'API

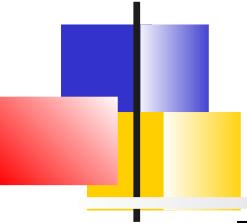


Pattern

Decompose by subdomain Pattern¹ :

Définir des service correspondant aux sous-domaines de DDD²

1. <https://microservices.io/patterns/decomposition/decompose-by-subdomain.html>
2. *Domain Driven Design, Eric Evans, Addison-Wesley Professional, 2003*

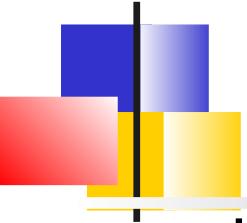


Décomposition par sous-domaines

DDD est une approche de construction d'applications complexes centrée sur le développement d'un modèle de domaine orienté objet.

DDD a deux concepts très utiles lors de la décomposition en microservices :

- les sous-domaines
- les contextes délimités.

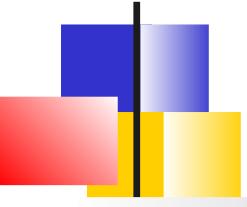


Approche traditionnelle de modélisation objet

Les approches traditionnelles fournissent un unique modèle pour l'ensemble de l'entreprise

- Les différentes entités de l'organisation ont du mal à s'accorder sur le modèle
- Le modèle final est trop complexe pour les besoins de certaines entités
- Le modèle peut porter à confusion. Car le même terme peut avoir des significations différentes à travers les entités

DDD évite ces problèmes en définissant plusieurs sous-modèles de domaine, chacun avec une portée explicite.



DDD

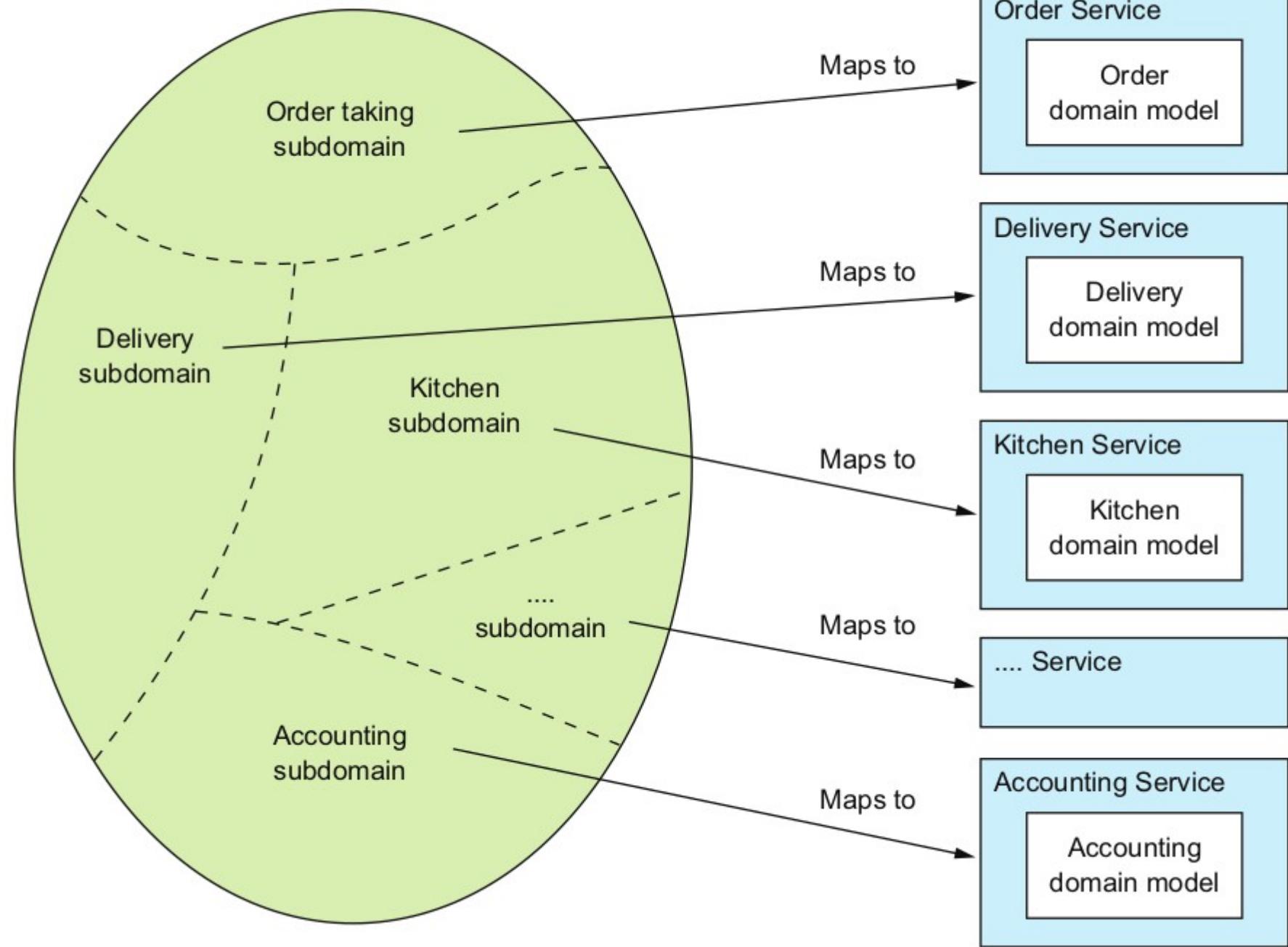
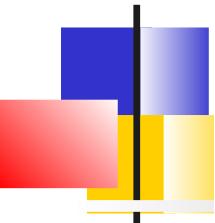
DDD définit un modèle de domaine distinct pour chaque sous-domaine.

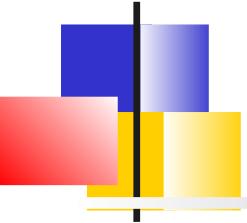
- Les sous-domaines sont identifiés selon la même approche que l'identification des capacités métier : analyser le métier et identifier les différents domaines d'expertise.

La portée d'un sous-domaine est appelé **contexte limité**.

- Un contexte limité inclut les artefacts de code qui implémentent le modèle.
- Avec l'architecture micro-service, chaque contexte délimité est un service ou éventuellement un ensemble de services.

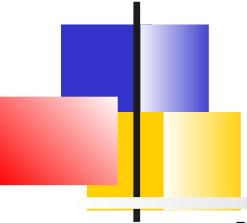
FTGO domain





Stratégies de décomposition

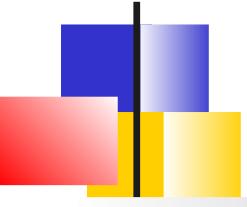
Introduction
Business capability Pattern
Subdomain Pattern
Définition de l'API



Définition de l'API

Une opération d'API existe pour 2 raisons:

- Elle correspond à une opération système.
Elle est alors invoquée par des clients externes et peut-être par d'autres services.
- Elle permet la collaboration entre les services.
Ces opérations ne sont invoquées que par d'autres services

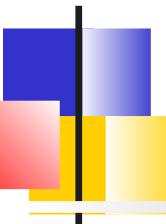


Définition de l'API

Le point de départ pour définir les APIs de service consiste à mapper chaque opération système à un service.

Ensuite, nous décidons si un service doit collaborer avec d'autres pour mettre en œuvre une opération système.

- Si une collaboration est requise, nous déterminons ensuite quelles API ces autres services doivent fournir afin de prendre en charge la collaboration.



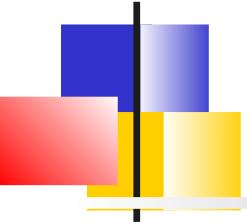
Obstacles à la décomposition

La **latence réseau** : Certaines décomposition sont infaisables à cause de trop nombreux aller-retours entre les services.

Les **communications synchrones** entre services réduisent la disponibilité

La contrainte de maintenir la **cohérence des données** entre services

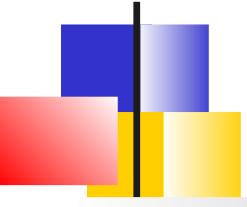
Les classes utilisées par tous les services.
DDD nous dit : duplication !!



Communication

Introduction

Communication Synchrone
Communication Asynchrone



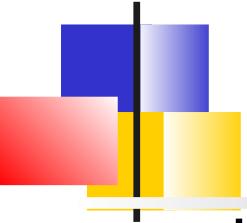
Introduction

Différentes technologies pour la communication entre micro-services :

- Synchrones sur le mode requête/réponse : REST, gRPC, GraphQL
- Asynchrones, Communications via message : AMQP, STOMP

Différents formats :

- Texte : JSON, XML
- Binaires : Avro, Protocol Buffer

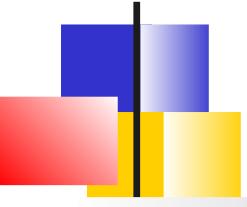


Rôles de l'API

Les API ou interfaces sont au cœur du développement logiciel.

- Une interface/API bien conçue expose des fonctionnalités utiles tout en masquant la mise en œuvre. Il permet à la mise en œuvre de changer sans impact sur les clients
- L'interface/API est un contrat entre le service et ses clients

Avec un langage typé, si l'interface change, le projet ne compile plus et l'on règle le problème. Il faut avoir le même mécanisme dans le cadre des micro-services

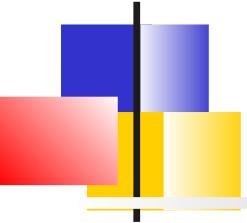


Design By Contract

Il est important de définir précisément l'API d'un service en utilisant une sorte de langage de définition d'interface (IDL).

La définition de l'API dépend du style d'interaction :

- OpenAPI pour Rest
- Canaux de messages, format et type du message pour l'asynchrone

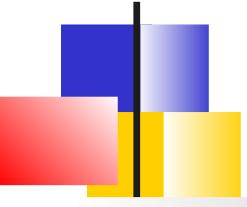


Evolution de l'API

Avec les micro-services, changer l'API d'un service est beaucoup plus difficile.

Les clients étant développés par d'autres équipes.

- => Il est impossible de forcer tous les clients à se mettre à niveau en même temps.
- => Les anciennes et les nouvelles versions d'un service devront s'exécuter simultanément.



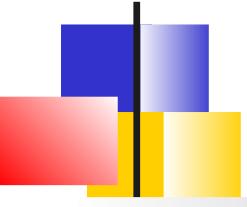
Versionning

Semvers¹ définit le numéro de version composé de trois parties : MAJOR.MINOR.PATCH :

- MAJEUR : Une modification incompatible à l'API
- MINEUR : Améliorations rétrocompatibles à l'API
- PATCH : Correction de bogue rétrocompatible

Les changements de version doivent être contrôlés. Le n° de version est alors présent dans l'interaction (URL de l'API Rest, Mime-type ou contenu d'un message asynchrone)

1. <https://semver.org/lang/fr/>



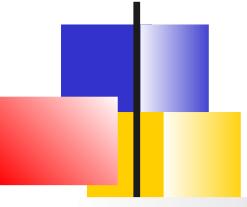
Changements mineurs

Les changements rétrocompatibles sont des ajouts :

- Ajout d'attributs optionnelles à la requête
- Ajout d'attributs à la réponse
- Ajout de nouvelles opérations

Si les services ont été développés selon pour fournir des valeurs par défaut des attributs de requête manquant, d'ignorer les attributs de réponse dont on n'a pas besoin, les changements mineurs ne posent pas de problème. (Voir également Robustness principle¹⁾)

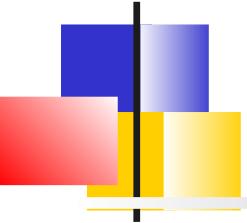
1.https://en.wikipedia.org/wiki/Robustness_principle



Changements majeurs

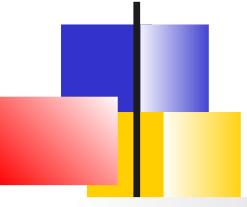
En cas de changements majeurs, les services doivent supportés pendant un certain temps plusieurs versions d'API

Afin de prendre en charge plusieurs versions d'une API, les adaptateurs du service qui implémentent les API contiendront une logique qui traduit les requêtes des anciennes version dans la nouvelle version.



Communication

Introduction
Communication Synchrone
Communication Asynchrone

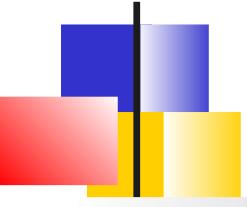


Introduction

Lors de l'utilisation d'un mécanisme IPC (Inter Process Communication) basé sur l'invocation de procédure à distance, un client envoie une demande à un service, et le service traite la demande et renvoie une réponse.

Certains clients peuvent bloquer l'attente d'une réponse, et d'autres peuvent avoir une architecture réactive et non bloquante.

Mais contrairement à l'utilisation de la messagerie, le client suppose que la réponse arrivera en temps opportun.



Pattern

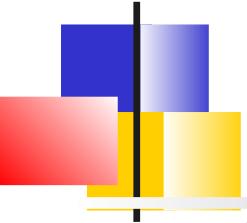
Remote procedure invocation

Pattern¹ : Un client invoque un service utilisant un protocole d'invocation de procédure à distance comme REST

D'autres protocoles modernes existent :

- GraphQL (Au départ Facebook, mais a tendance à se propager)
- Netflix Falcor
- gRPC (Google)

1. <http://microservices.io/patterns/communication-style/messaging.html>



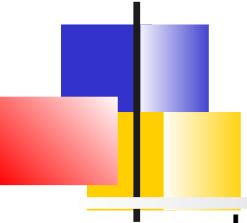
Avantages de REST

C'est simple et familier.

Facile à tester, navigateur avec Postman, curl

HTTP est compatible avec les pare-feu.

Ne nécessite pas de broker intermédiaire, ce qui simplifie l'architecture.



Inconvénients

Un seul type de communication

Il ne prend en charge que le style demande/réponse.

Les clients doivent connaître les emplacements (URL) des instances de service.

Nécessité d'utiliser des mécanismes de découverte de service pour localiser les instances de service.

Disponibilité réduite.

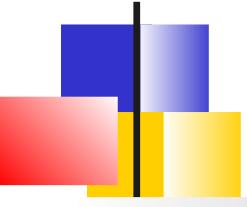
Le client et le service communiquent directement sans intermédiaire , ils doivent tous les deux fonctionner pendant toute la durée de l'échange.

Granularité fine

La récupération de plusieurs ressources en une seule requête est un défi.

Peu de verbes HTTP

Il est parfois difficile de mapper plusieurs opérations de mise à jour sur des verbes HTTP.

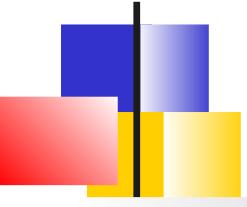


Service discovery

Les instances de service ont des emplacements réseau attribués dynamiquement.

De plus, l'ensemble des instances de service change de manière dynamique en raison de l'autoscaling, des échecs et des mises à niveau.

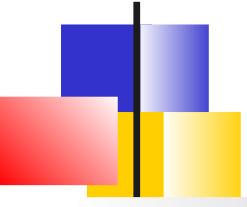
Par conséquent, votre code client doit utiliser une découverte de service.



Mise en oeuvre

2 manières principales de mettre en œuvre la découverte de services :

- Les services et leurs clients interagissent directement avec le registre des services.
- L'infrastructure de déploiement gère la découverte de service.



Patterns

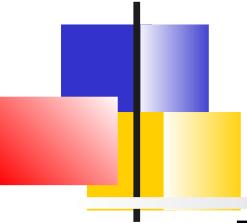
Self registration Pattern¹ : Une instance de service s'enregistre auprès du service de discovery
Exemple : Eureka

Client-side discovery Pattern² : Un service client récupère la liste des services disponibles auprès du service de discovery et équilibre la charge
Exemple : Eureka, Consul

3rd party registration Pattern³ : Les instances de services sont automatiquement enregistrés par un agent tiers
Exemple : Consul, Kubernetes

Server-side discovery Pattern⁴ : Un client effectue une requête vers un routeur responsable de la découverte de service.
Exemple : Kubernetes

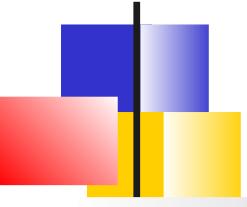
1. <http://microservices.io/patterns/self-registration.html>
2. <http://microservices.io/patterns/client-side-discovery.html>
3. <http://microservices.io/patterns/3rd-party-registration.html>
4. <http://microservices.io/patterns/server-side-discovery.html>



Infrastructure vs Code

Fourniture par la plateforme de déploiement :

- Ni le service, ni son client ne contiennent de code pour la découverte de services
- Le service est disponible quelque soit le langage de développement du micro-service
- Par contre, tous les services doivent être déployés sur la même infrastructure.
Par exemple Kubernetes

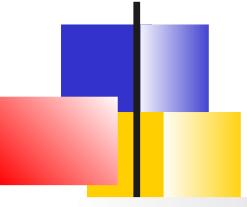


Disponibilité

Pour pallier, les soucis de disponibilité réduite du
au fait que une dépendance d'un micro-service
peut :

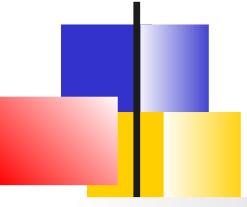
- Être en échec
- Être en surcharge
- Être en train de faire une montée de version
- ...

Les services doivent implémenter la résilience, le
pattern le plus commun étant le ***Circuit
Breaker Pattern***



Circuit Breaker Pattern

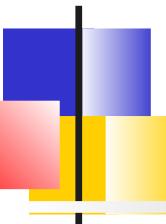
Lorsqu'un service en appelle un autre de manière synchrone, il est toujours possible que l'autre service ne soit pas disponible ou présente une latence si élevée qu'il est essentiellement inutilisable. Des ressources précieuses telles que des threads peuvent être consommées dans l'appelant en attendant que l'autre service réponde. Cela pourrait conduire à l'épuisement des ressources, ce qui rendrait le service appelant incapable de traiter d'autres demandes. La défaillance d'un service peut potentiellement se répercuter sur d'autres services dans l'application.¹



Solution

Un client doit invoquer un service distant via un proxy qui fonctionne de la même manière qu'un disjoncteur électrique.

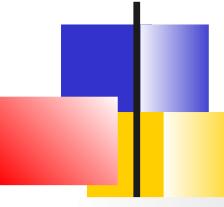
- Lorsque le nombre de défaillances consécutives dépasse un seuil, le disjoncteur se déclenche et, pendant la durée d'un délai d'expiration, toutes les tentatives d'appel du service distant échouent immédiatement.
- Une fois le délai expiré, le disjoncteur autorise le passage d'un nombre limité de demandes de test. Si ces demandes aboutissent, le disjoncteur reprend son fonctionnement normal. Sinon, en cas d'échec, le délai d'expiration recommence.



Spring Cloud Breaker Pattern

Spring Cloud Breaker fournit une abstraction et donc une API cohérente à utiliser pour les librairies implémentant le pattern circuit breaker. Il supporte :

- Netflix Hystrix
- Resilience4J
- Sentinel
- Spring Retry



Patterns Resilience4J

Retry : Répéter les exécutions échouées

De nombreux fautes sont transitoires et peuvent s'auto-corriger après un court délai.

Circuit Breaker : blocages temporaires des défaillances possibles

Lorsqu'un système éprouve de sérieuses difficultés, il est préférable d'échouer rapidement que d'attendre les clients.

Rate limiter : Limiter les exécutions

Limitez le taux des requêtes

Time Limiter : Limiter le temps d'exécution des requêtes

Au-delà d'un certain intervalle d'attente, un résultat positif est peu probable

Bulkhead : Limiter les exécutions concurrentes

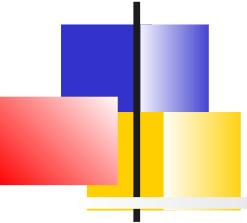
Les ressources sont isolées dans des pools de sorte que si l'une échoue, les autres continuent de fonctionner.

Cache : Mémoriser un résultat réussi

Une partie des demandes peut être similaire.

Fallback : Fournir un résultat alternatif pour les échecs

Les choses échoueront toujours - planifiez ce que vous ferez lorsque cela se produira.

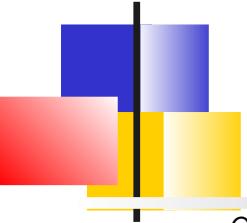


Exemple SpringCloud

Spring Cloud peut injecter un
CircuitBreakerFactory.

Sa méthode *create()* permet de définir une classe ***CircuitBreaker*** dont la méthode *run()* prend 2 lambda en arguments :

- Le code à exécuter dans l'autre thread
- Optionnellement, un code de fallback

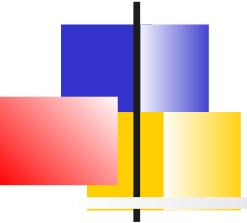


Exemple

```
@Service
public static class DemoControllerService {
    private RestTemplate rest;
    private CircuitBreakerFactory cbFactory;

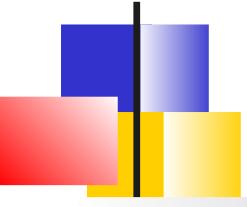
    public DemoControllerService(RestTemplate rest, CircuitBreakerFactory cbFactory)
    {
        this.rest = rest;
        this.cbFactory = cbFactory;
    }

    public String slow() {
        return cbFactory.create("slow").run(
            () -> rest.getForObject("/slow", String.class),
            throwable -> "fallback"
        );
    }
}
```



Communication

Introduction
Communication Synchrone
Communication Asynchrone



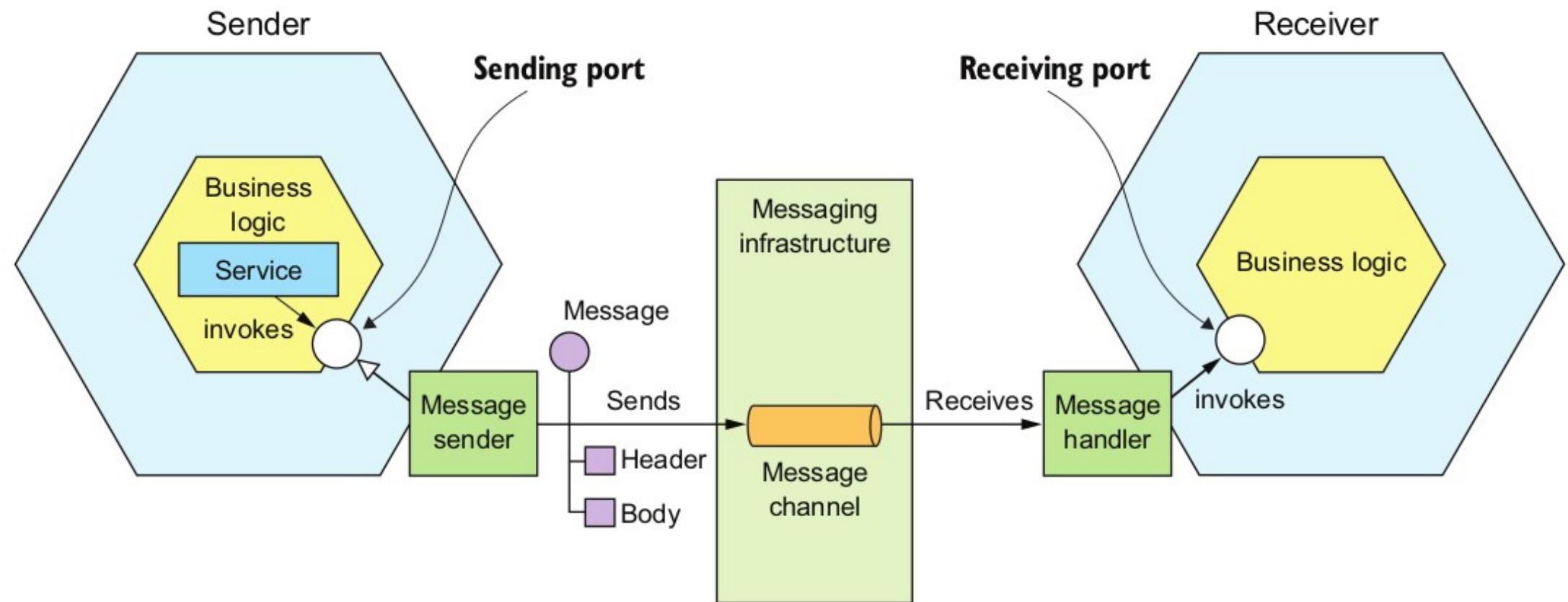
Introduction

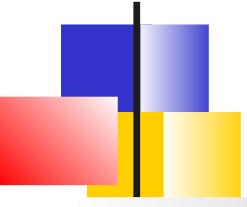
Messaging Pattern¹ : Un client invoque un service en utilisant une messagerie asynchrone

- Le pattern messaging fait souvent intervenir un message broker
- Un client effectue une requête en postant un message asynchrone
- Optionnellement, il s'attend à recevoir une réponse

1. <http://microservices.io/patterns/communication-style/messaging.html>

Architecture



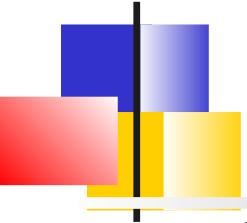


Message

Un message est constitué d'entêtes (ensemble de clés-valeurs) et d'un corps de message

On distingue 3 types de messages :

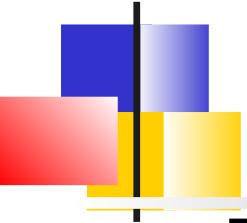
- **Document** : Un message générique ne contenant que des données Le récepteur décide comment l'interpréter
- **Commande** : Un message spécifiant l'action à invoquer et ses paramètres d'entrée
- **Événement** : Un message indiquant que quelque chose vient de se passer. Souvent un événement métier



Canaux de messages

2 types de canaux :

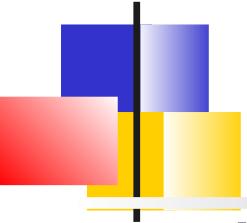
- **Point-to-point** : Le canal délivre le message à un des consommateurs lisant le canal.
Ex : Envoie d'un message commande
- **PubAndSub** : Le canal délivre le message à tous les consommateurs attachés (les abonnés)



Styles d'interaction

Tous les styles d'interactions sont supportés :

- Requête/Réponse synchrone.
Le client attend la réponse
- Requête/Réponse asynchrone
Le client est notifié lorsque la réponse arrive
- One way notification
Le client n'attend pas de réponse
- Publish and Subscribe :
Le producteur n'attend pas de réponse
- Publish et réponse asynchrones
Le producteur est notifié lorsque les réponses arrivent

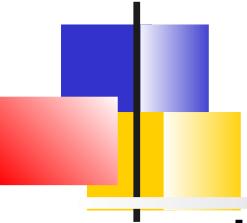


Spécification de l'API

La spécification consiste à définir

- Les noms des canaux
- Les types de messages et leur format.
(Typiquement JSON)

Par contre à la différence de REST et
OpenAPI pas de standard



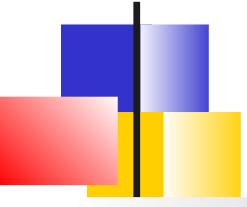
Message Broker

Un message broker est un intermédiaire par lequel tous les messages transitent

- L'émetteur n'a pas besoin de connaître l'emplacement réseau du récepteur
- Le message broker bufferise les messages

Implémentations courantes :

- ActiveMQ
- RabbitMQ
- Kafka
- AWS Kinesis



Facteurs de choix (1°)

Langages de programmation supportés

C'est mieux si il en supporte plusieurs

Standard de messaging supportés

AMQP, STOMP ou propriétaire

Ordre des messages

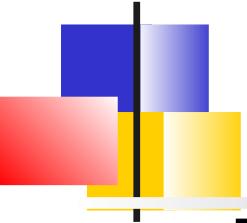
Le message broker préserve t il l'ordre d'émission des messages

Garanties de livraison

At-most-Once, At-Least-Once ou Exactly-Once

Persistiance

Les messages survivent-ils au crash ?



Facteurs de choix (2)

Durabilité

Si un consommateur se reconnecte au broker, récupère t il les messages qui ont été envoyés entre temps

Scalabilité

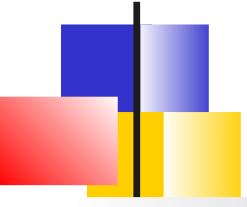
Le broker est-il scalable ?

Latence

Quel est le délai entre l'émission et la réception ?

Consommation concurrente

Le message broker permet-il que les messages d'un canaux soient entre des récepteurs répliqués



Inconvénients d'un broker

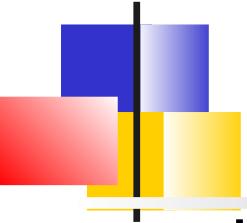
Potentiel goulot d'étranglement

Potentiel single point of failure

Essentiel que le broker soit hautement disponible

Complexité des opérations accrue

Le broker est un composant supplémentaire qu'il faut installé, configuré et opéré



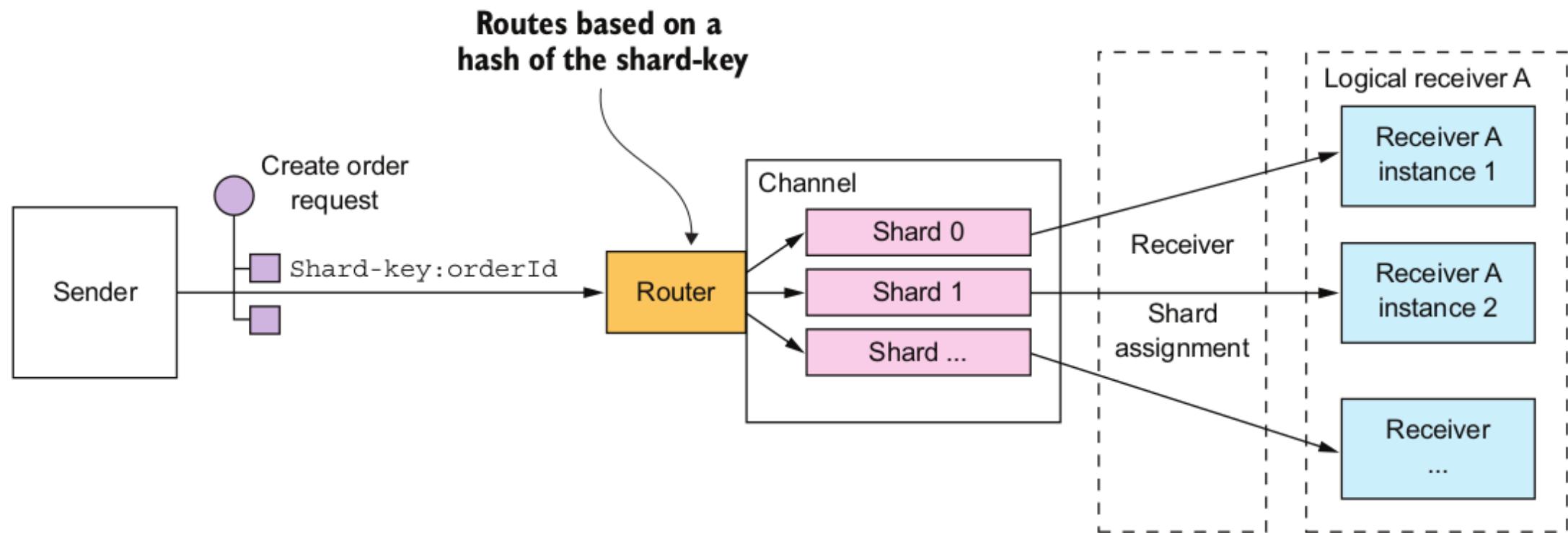
Ordre et scaling des récepteurs

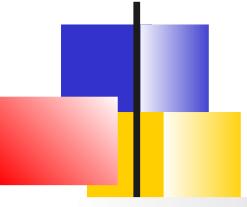
L'un des défis est de savoir comment scaler les récepteurs de messages tout en préservant l'ordre des messages.

Une solution fréquente (Kafka, AWS Kinesis) est l'utilisation de canaux partitionnés (shards)

- Un canal est constitué de plusieurs partition. Chacune se comporte comme un canal
- L'émetteur spécifie la clé de partition dans une entête de message
- Le broker rassemble plusieurs instances d'un récepteur afin de les traiter comme un seul

Sharding





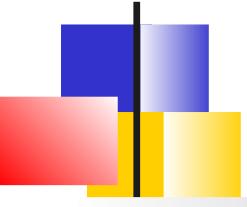
Gestion des doublons

Un broker devrait idéalement délivré chaque message une et une seule fois. Mais cela est en général trop coûteux, la plupart des brokers garantissent une livraison ***at-least once***.

En cas de dysfonctionnement, des doublons peuvent alors apparaître

Différentes façons pour gérer les doublons :

- Écrire des gestionnaires de message idempotent
- Déetecter et éliminer les doublons.
En stockant un id de message par exemple



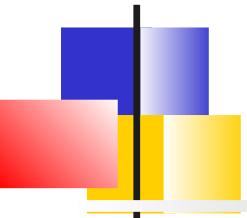
Messaging transactionnel

L'envoi d'un message fait souvent partie d'une transaction mettant à jour une base.

L'envoi du message doit faire partie de la transaction

- La solution traditionnelle est d'utiliser une transaction distribuée qui englobe la base de données et le message broker mais cela n'est pas adapté aux applications modernes.

Kafka par exemple ne supporte pas les transactions distribuées



Transactional outbox

Le pattern ***Transactional outbox***¹ consiste à utiliser une table OUTBOX supplémentaire dans la base

Lors de la transaction locale métier, cette table est mise à jour avec les propriétés ACID

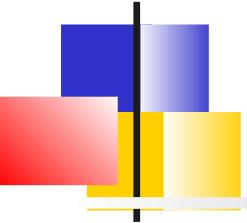
Le composant *MessageRelay* lit la table OUTBOX et envoie vers le broker.

- Soit en pollant régulièrement la base avec un SELECT Pattern Polling Publisher²
- Soit en accédant au journal de transactions. Pattern Transaction Log Tailing³

1. <http://microservices.io/patterns/data/transactional-outbox.html>

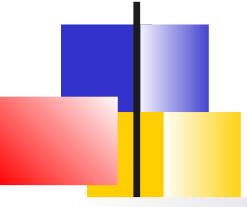
2. <http://microservices.io/patterns/data/polling-publisher.html>

3. <http://microservices.io/patterns/data/transaction-log-tailing.html>



Cohérence des données et transaction

Introduction Saga Pattern

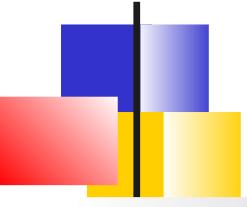


Introduction

La gestion des transactions est simple dans une application monolithique accédant une unique BD (JDBC, Framework), plus compliqué si le monolithique accède plusieurs bases et brokers.

Dans une architecture micro-service, les transactions englobe plusieurs multiple services qui ont chacun leur base de données.

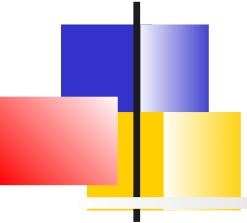
C'est sûrement un des plus gros obstacle pour passer d'un monolithe à une architecture micro-services



Transaction distribuée

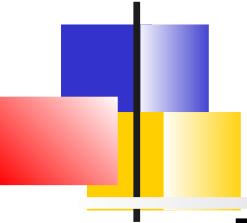
L'approche traditionnelle repose sur X/Open Distributed Transaction Processing (DTP) Model qui utilise un protocole de commit à 2 phases.

Les ressources transactionnelles doivent être compatibles avec XA ce qui n'est pas le cas des bases NoSQL, de RabbitMQ, de Kafka



Cohérence des données et transaction

Introduction
Saga Pattern

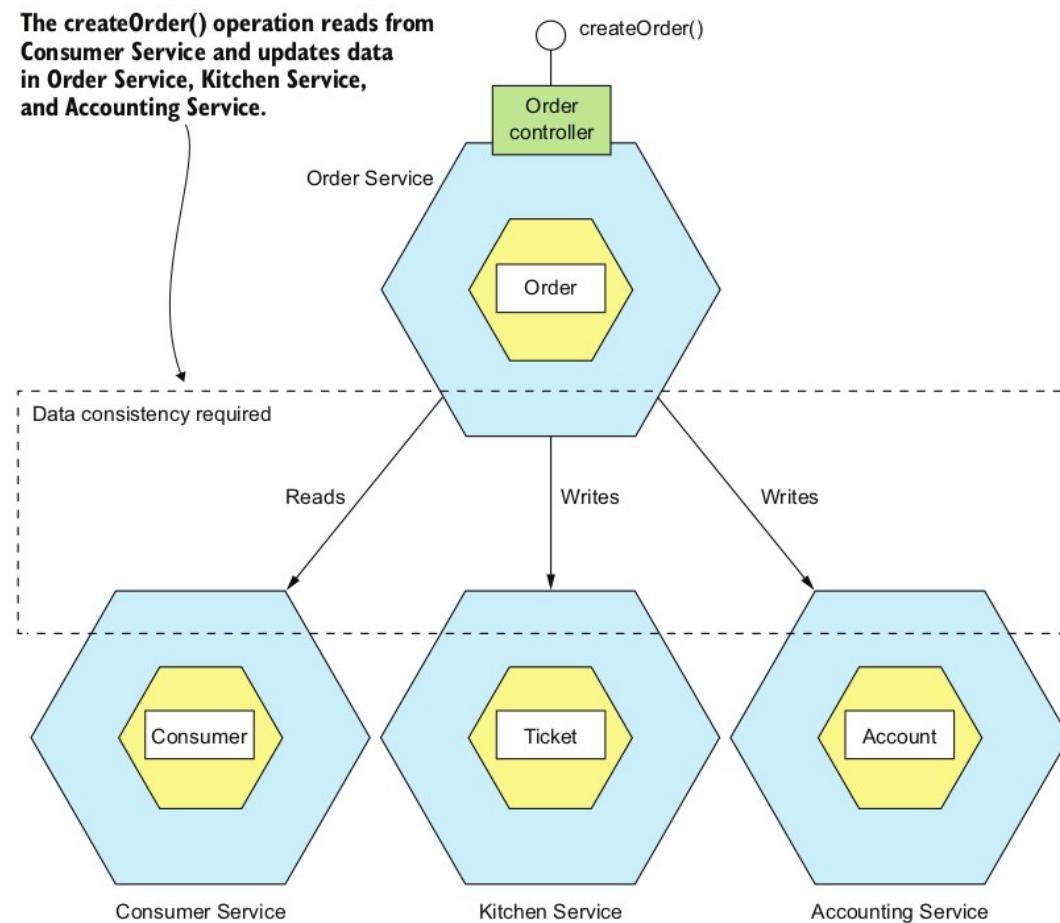


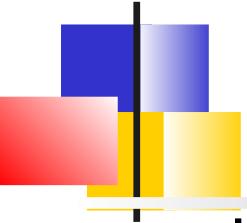
Exemple : Gestion des transactions avec Saga

Problème : Comment implémenter des transactions qui englobent plusieurs services, i.e des opérations qui mettent à jour des données dispersées dans plusieurs services

Solution : Une **saga**, séquence de transactions locales basée sur des messages. Cette séquence présente les propriétés ACD (Atomicité, Cohérence, Durabilité) mais la propriété d'isolation n'est pas respectée. En conséquence, l'application doit utiliser des contre-mesures pour empêcher ou réduire l'impact des anomalies de concurrence.

Exemple

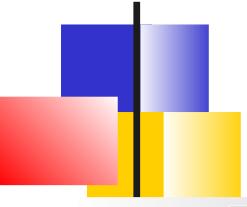




Structure des transactions

Une saga consiste en une séquences de 3 types de transactions :

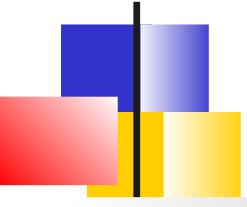
- **Transaction compensable** : peuvent potentiellement être annulées à l'aide d'une transaction compensatoire
- **Transaction pivot** : Si elle est validée, la saga s'exécutera jusqu'à la fin.
Une transaction pivot peut être une transaction qui n'est ni compensable, ni réessayable. Cela peut s'agir de la dernière transaction compensable ou de la première transaction réessayable
- **Transaction réessayable** : Elles suivent la transaction pivot transaction et sont assurés de réussir



Exemple

Etape S	Service	Transaction	Tr. de compensation
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetail()	-
3	KitchenService	createTicket()	rejectTicket()
4	AccounningService	authorizeCreditCard()	
5	RestaurantService	ackRestaurantOrder()	
6	OrderService	approveOrder()	

- 1,2,3 : Sont des transactions compensables
- 2 : Une opération de lecture n'a pas besoin de compensation
- 4 : Transaction pivot. Si elle réussit, *createOrder* doit aller jusqu'à la fin
- 5,6 : Transaction réessayable, jusqu'à ce qu'elles réussissent

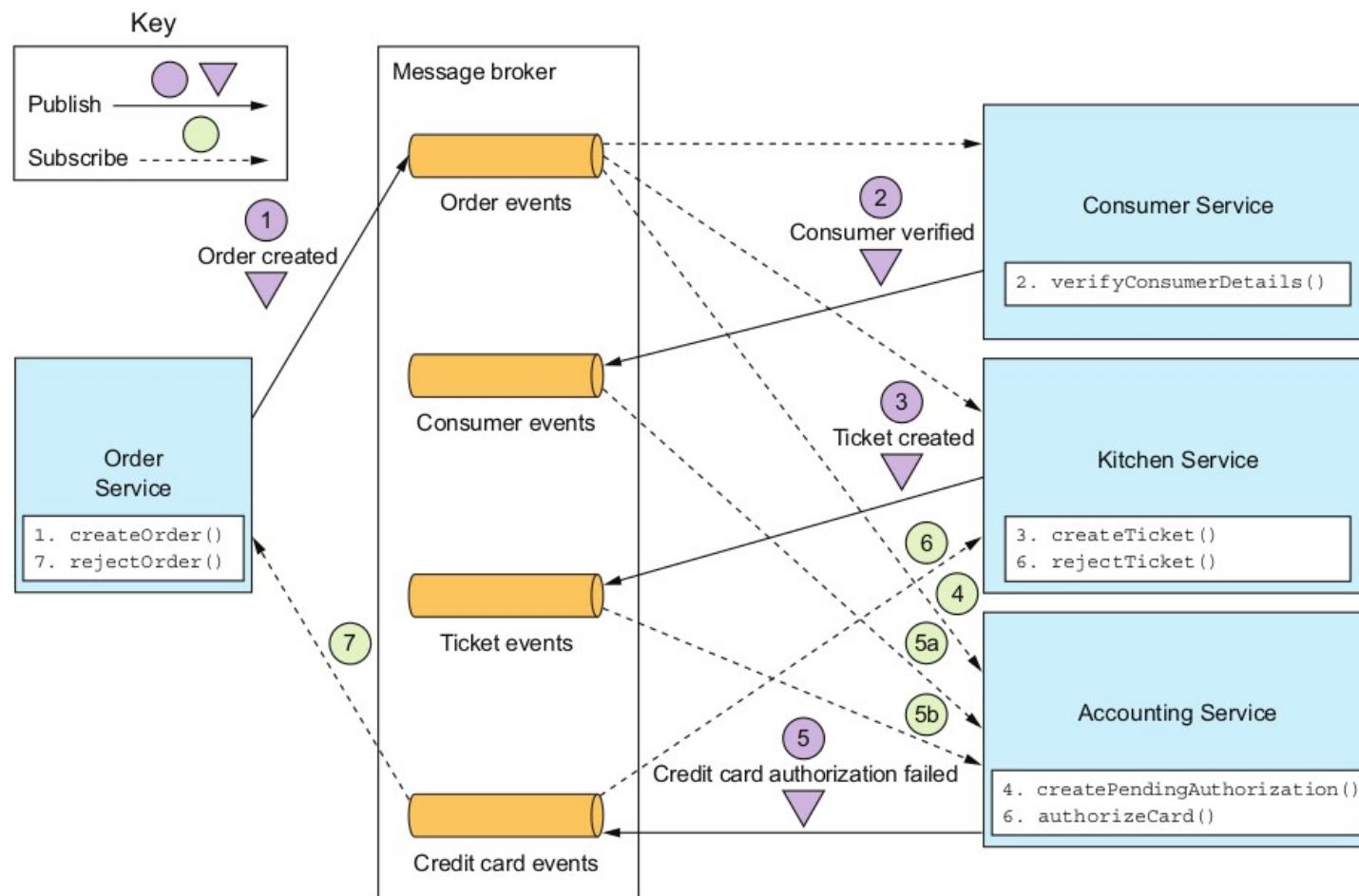


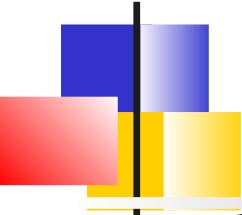
Implémentation

Il existe 2 façons de coordonner une saga:

- **chorégraphie** : Répartir la prise de décision et le séquençage parmi la saga de participants.
Les participants à la saga échangent des événements
- **orchestration** : un orchestrateur coordonne les transactions des participants.
Utilise une messagerie asynchrone pour piloter les participants

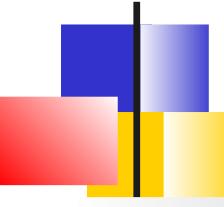
Chorégraphie





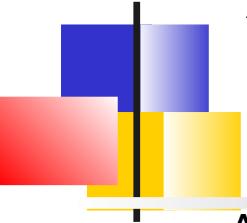
Séquence, si tout se passe bien

1. *OrderService* crée une commande dans l'état **APPROVAL_PENDING** et publie un événement **OrderCreated**.
2. *ConsumerService* utilise l'événement *OrderCreated*, vérifie que le consommateur peut passer la commande et publie un événement **ConsumerVerified**.
3. *KitchenService* consomme l'événement *OrderCreated*, valide la commande , crée un ticket dans un état **CREATE_PENDING** et publie l'événement **TicketCreated**.
4. *AccountingService* consomme l'événement *OrderCreated* et crée une CreditCardAuthorization dans un état **PENDING**.
5. *AccountingService* consomme les événements *TicketCreated* et *ConsumerVerified*, débite la carte de crédit du consommateur et publie l'événement **CreditCardAuthorized**.
6. *KitchenService* consomme l'événement *CreditCardAuthorized* et change l'état du ticket en **AWAITING_ACCEPTANCE** .
7. *OrderService* reçoit les événements *CreditCardAuthorized*, modifie l'état de la commande en **APPROVED** et publie un événement **OrderApproved**.



Séquence si cela se mal

1. OrderService crée une commande dans l'état **APPROVAL_PENDING** et publie un événement **OrderCreated**.
2. ConsumerService utilise l'événement *OrderCreated*, vérifie que le consommateur peut passer la commande et publie un événement **ConsumerVerified**.
3. KitchenService consomme l'événement *OrderCreated*, valide la commande , crée un ticket dans un état **CREATE_PENDING** et publie l'événement **TicketCreated**.
- 4 AccountingService consomme l'événement *OrderCreated* et crée une *CreditCardAuthorization* dans un état **PENDING**.
5. AccountingService consomme les événements *TicketCreated* et *ConsumerVerified*, débite la carte de crédit du consommateur et publie un événement **CreditCardAuthorizationFailed**.
6. KitchenService consomme l'événement *CreditCardAuthorizationFailed* et change l'état du billet en **REJECTED** .
7. OrderService utilise l'événement *CreditCardAuthorizationFailed* et modifie l'état de la commande en **REJECTED** .



Avantages / Inconvénients de la chorégraphie

Avantages

Simplicité :

les services publient des événements lorsqu'ils créent, mettent à jour ou suppriment des objets métier.

Couplage lâche :

Les participants s'abonnent à des événements et ne se connaissent pas directement.

Inconvénients

Plus difficile à comprendre :

contrairement à l'orchestration le code est décentralisé.

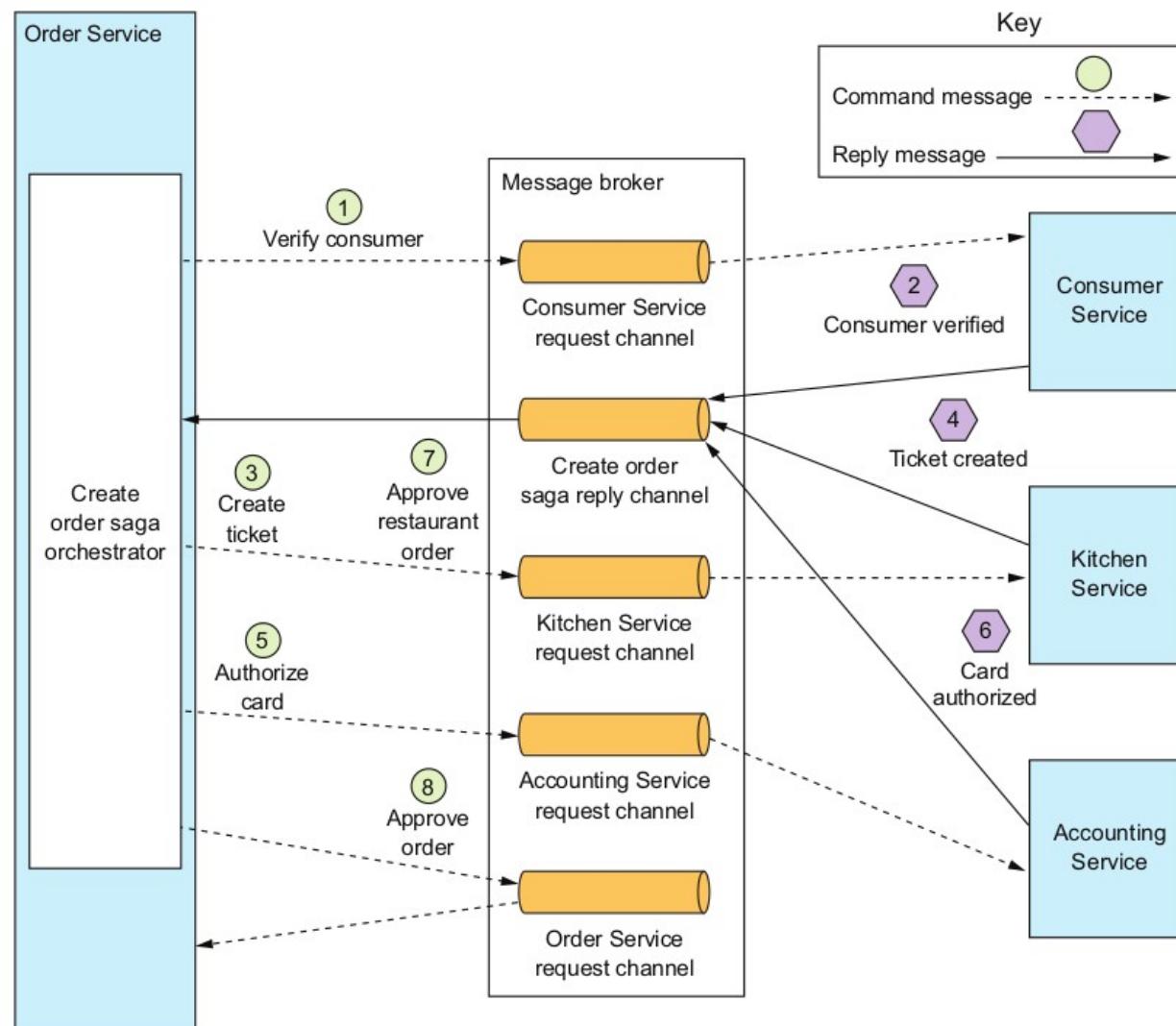
Dépendances cycliques entre les services :

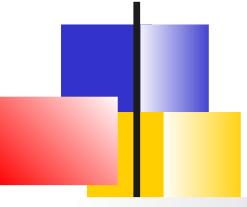
les participants à la saga s'abonnent aux événements des autres, ce qui crée souvent des dépendances cycliques.

Risque de couplage étroit :

Chaque participant à la saga doit s'abonner à tous les événements qui le concernent. Il existe un risque que les participants doivent être mis à jour si le cycle de vie l'objet métier (la commande dans l'exemple) change.

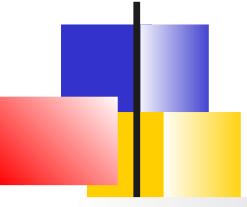
Orchestration





Quand tout se passe bien

1. L'orchestrateur de saga envoie une commande **VerifyConsumer** à *ConsumerService*.
2. *ConsumerService* répond par un message **Verified**
3. L'orchestrateur de saga envoie une commande **CreateTicket** à *KitchenService* .
4. *KitchenService* répond avec un message **TicketCreated**
5. L'orchestrateur de saga envoie un message **AuthorizeCard** à *AccountingService*.
6. *AccountingService* répond avec un message **CardAuthorized**.
7. L'orchestrateur de saga envoie une commande **ApproveTicket** à *KitchenService* .
8. L'orchestrateur de saga envoie une commande **ApproveOrder** à *OrderService*.



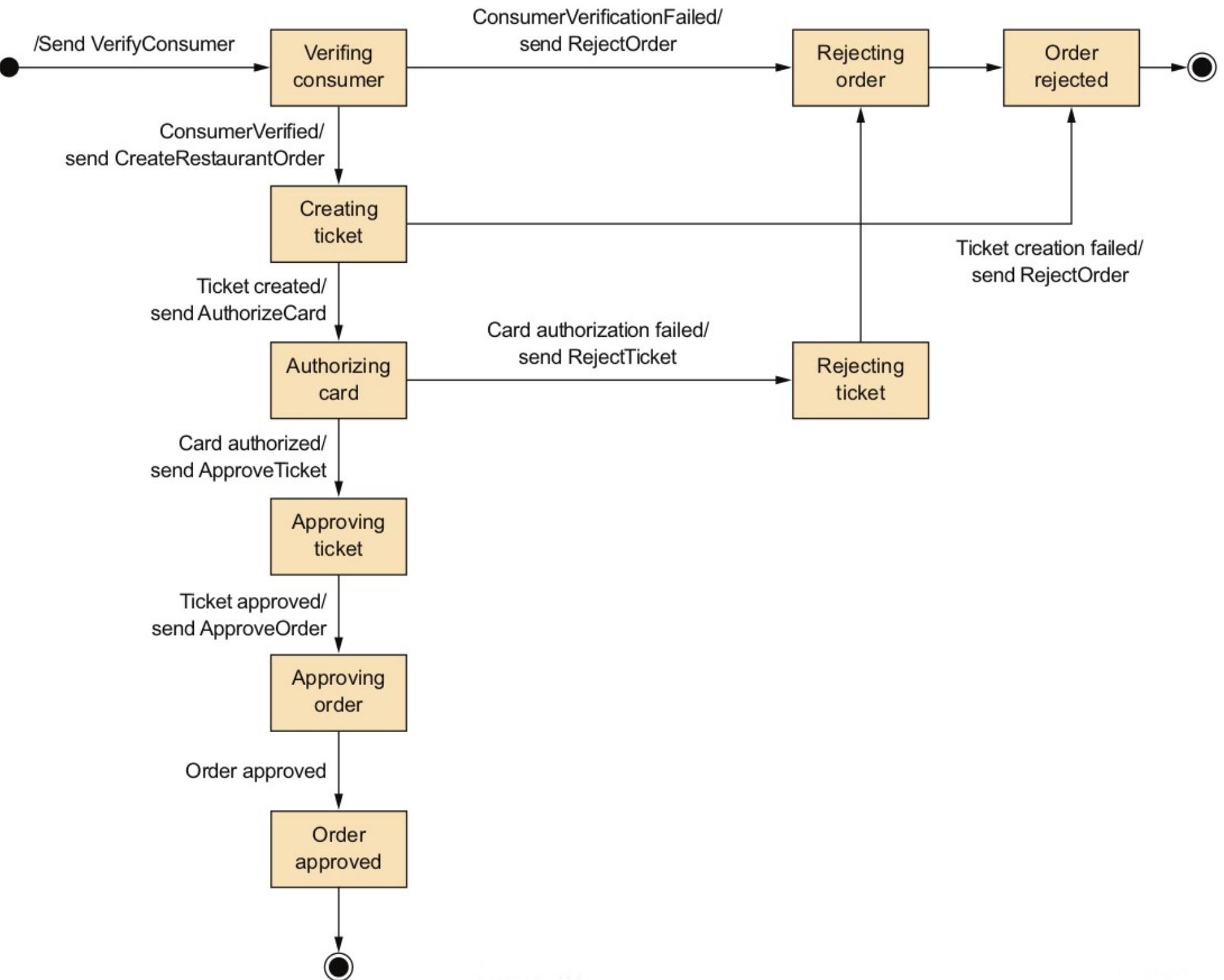
Modélisation d'une saga

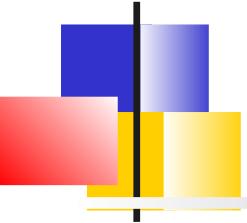
Une saga a généralement de nombreux scénarios

Il est alors utile de la modéliser via une machine à états

Les états de l'exemple :

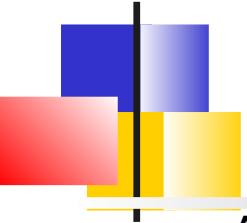
- Vérification du consommateur : l'état initial. Dans cet état, la saga attend que le Service Consommateur vérifie que le consommateur peut passer la commande.
- Création de ticket : la saga attend une réponse à la commande Créer un ticket.
- Autorisation carte : En attente du service de comptabilité pour autoriser la carte de crédit du consommateur.
- Commande approuvée : état final indiquant que la saga s'est terminée avec succès.
- Commande rejetée : état final indiquant que la commande a été rejetée par l'un des participants.





Orchestration et message transactionnel

Chaque étape d'une saga basée sur l'orchestration consiste en un service mettant à jour une base de données et publiant un message



Avantages et Inconvénients

Avantages :

Dépendances plus simples : Pas de dépendances cycliques.

L'orchestrateur dépend des participants mais pas l'inverse

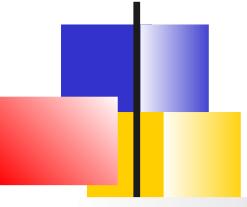
Moins de couplage : chaque service implémente une API invoquée par l'orchestrateur, il n'a donc pas besoin de connaître les événements publiés par les participants à la saga.

Améliore la séparation des préoccupations et simplifie la logique métier : la logique de coordination de la saga est localisée dans l'orchestrateur de la saga. Les objets de domaine sont plus simples et n'ont aucune connaissance des sagas auxquelles ils participent.

Inconvénients :

le risque de centraliser trop de logique métier dans l'orchestrateur.

=> L'orchestrateur ne doit être responsables du séquençage et ne pas contenir d'autre logique métier.



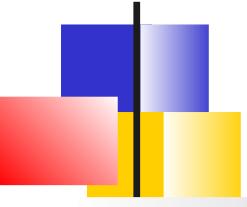
SAGA et ACD

Le challenge majeur avec SAGA est qu'il n'a pas la propriété d'isolation des transactions locales *ACID*

Les mises à jour apportées par chacune des transactions locales d'une saga sont immédiatement visibles pour les autres sagas concurrentes, donc avant que celle-ci soit terminée.

Ce comportement peut provoquer 3 problèmes :

- **Perte de mise à jour** : Une saga écrase sans lire les modifications apportées par une autre saga.
- **Dirty reads** : Une transaction ou une saga lit les mises à jour effectuées par une saga qui n'a pas encore terminé ces mises à jour.
- **Lectures floues/non répétables** : deux étapes différentes d'une saga lisent les mêmes données et obtenir des résultats différents car une autre saga a fait des mises à jour



Contre-mesures

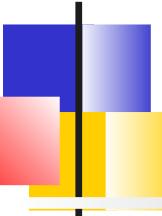
Le développeur doit écrire les sagas de manière à prévenir ces anomalies ou minimiser les risques.

De nombreuses techniques sont possibles¹ :

- **Verrou sémantique** : un verrou au niveau de l'application.
- **Mises à jour commutatives** : Les opérations de mise à jour sont exécutables dans n'importe quel ordre.
- **Vue pessimiste** : réorganisation des étapes d'une saga pour minimiser les risques.
- **Selecture** : Relire les données pour vérifier qu'elles sont inchangées avant une mise à jour. *Optimistic Offline Lock Pattern*²
- **Fichier de version** : enregistrez les mises à jour afin qu'elles puissent être réorganisées.
- **Par valeur** : Évaluer le mécanisme à utiliser à chaque requête

1. <https://dl.acm.org/citation.cfm?id=284472.284478>

2. <https://martinfowler.com/eaaCatalog/optimisticOfflineLock.html>



Exemple : Verrou sémantique

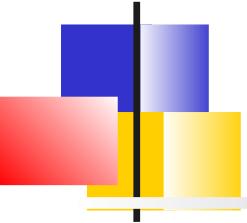
Les transactions compensables d'une sage la transaction positionne un flag dans l'enregistrement qu'elle mette à jour.

Par statut : *PENDING

L'indicateur indique que l'enregistrement n'est pas validé et peut potentiellement changer.

L'indicateur peut être traité par les autres transactions

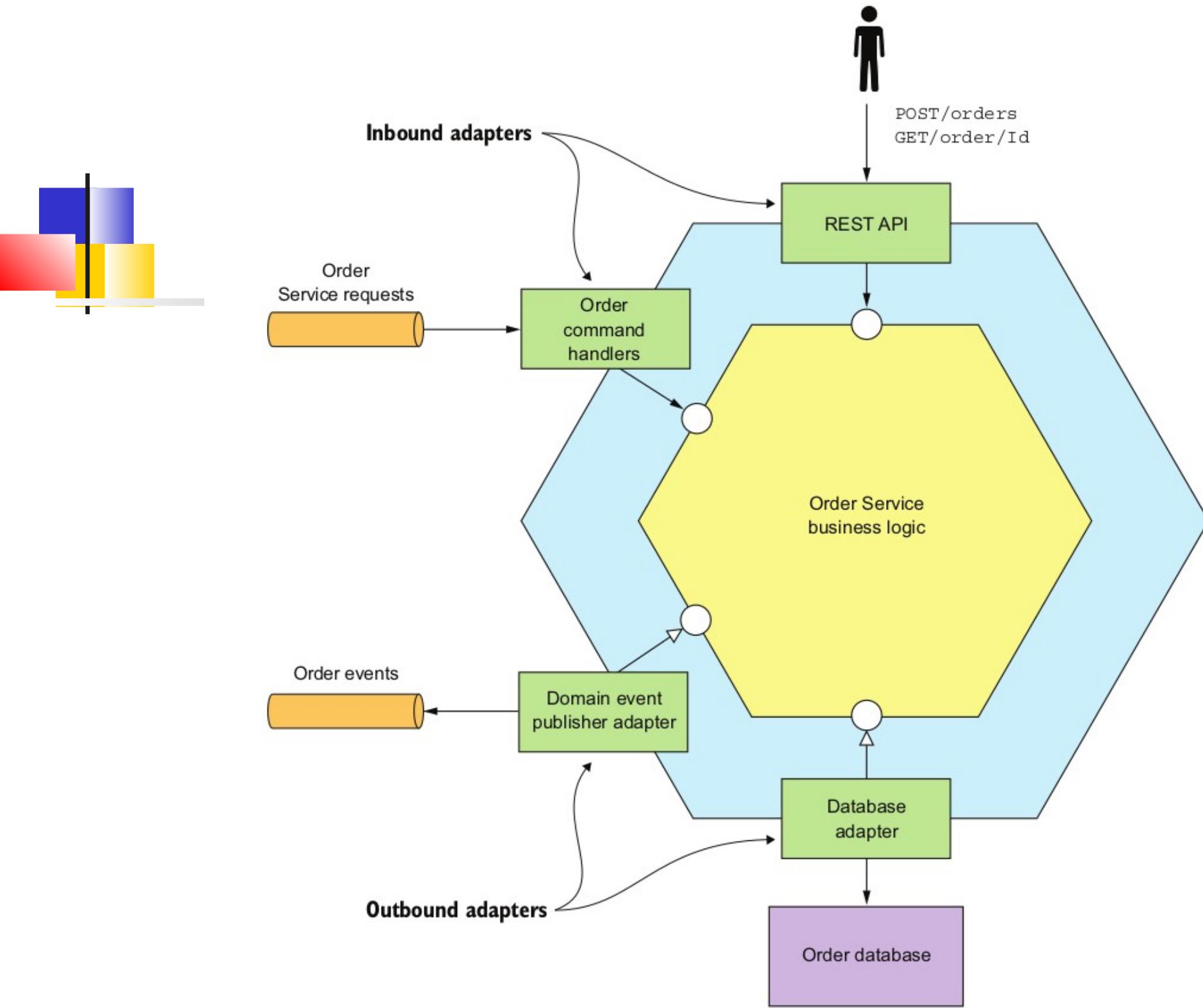
- soit comme un verrou. Elles s'interdisent d'accéder à la donnée
- soit un avertissement, éventuellement remonter vers l'utilisateur. .

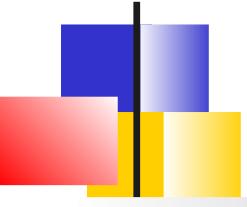


Design Logique métier

Introduction

Transaction Script Pattern
Patterns Orienté objet
Event Sourcing Pattern





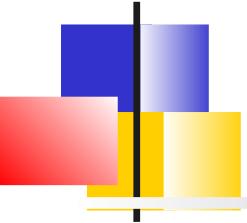
Introduction

La logique métier est généralement la partie la plus complexe du service.

L'organisation des classes doit être la plus appropriée à l'application

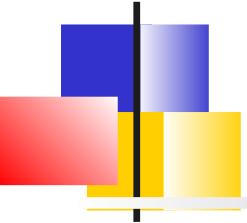
Même avec une technologie objet, il y a 2 principaux patterns pour organiser la logique métier d'un service :

- Procédurale : Transaction script pattern,
- Orienté objet : Domain model pattern.



Design Logique métier

Introduction
Transaction Script Pattern
Patterns Orienté objet
Event Sourcing Pattern



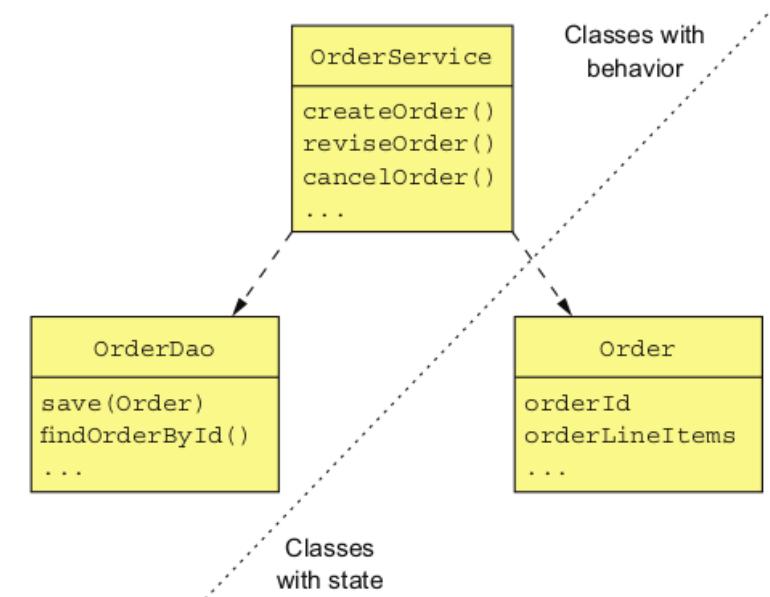
Transaction Script Pattern

Transaction Script Pattern¹ est le plus simple des patterns pour la logique métier.

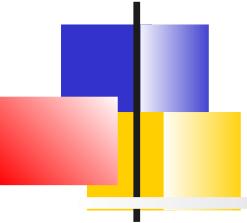
La logique métier est organisée en procédure. Chaque procédure correspond à une requête possible du système

Les scripts sont typiquement présents dans les classes services

Inconvénient : Si la logique est complexe, on tend vers un code spaghetti

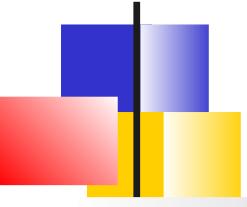


1. <https://martinfowler.com/eaaCatalog/transactionScript.html>



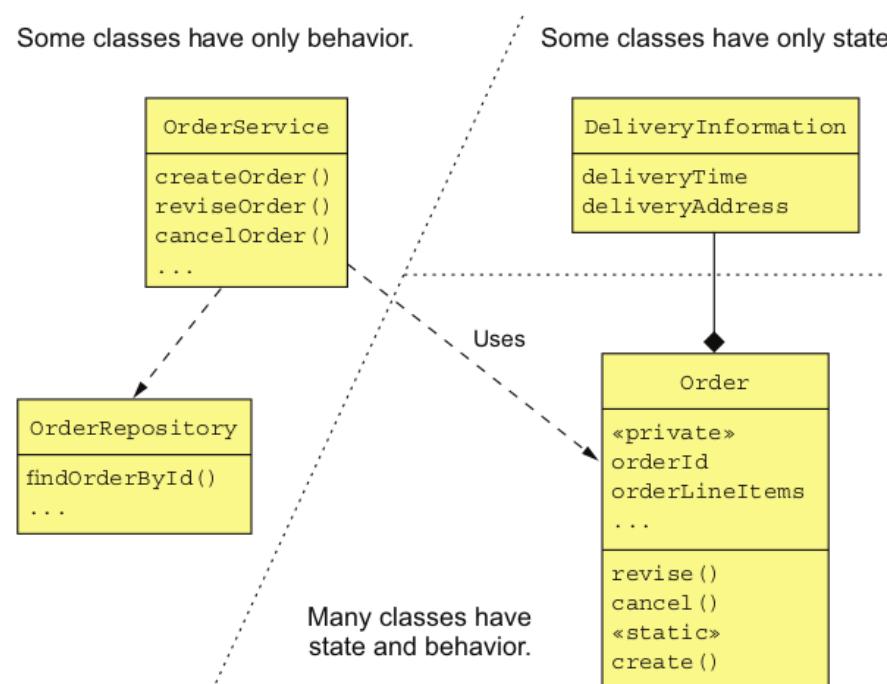
Design Logique métier

Introduction
Transaction Script Pattern
Patterns Orienté objet
Event Sourcing Pattern

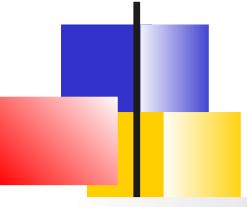


Domain Model Pattern

Domain model Pattern¹ organise la logique métier dans un modèle Objet dont les classes contiennent un état et un comportement



1. <https://martinfowler.com/eaaCatalog/domainModel.html>



Avantages du Domain Model pattern

Avec ce pattern, les méthodes de la classe service sont généralement simples.

- Elle délègue presque toujours aux objets de domaine qui contiennent l'essentiel de la logique métier.

Le design est facile à comprendre et à maintenir

- Au lieu de se composer d'une grande classe qui fait tout, il se compose d'un certain nombre de petites classes qui ont chacune un petit nombre de responsabilités.

Le design est plus facile à tester

- Chaque classe peut être testée indépendamment

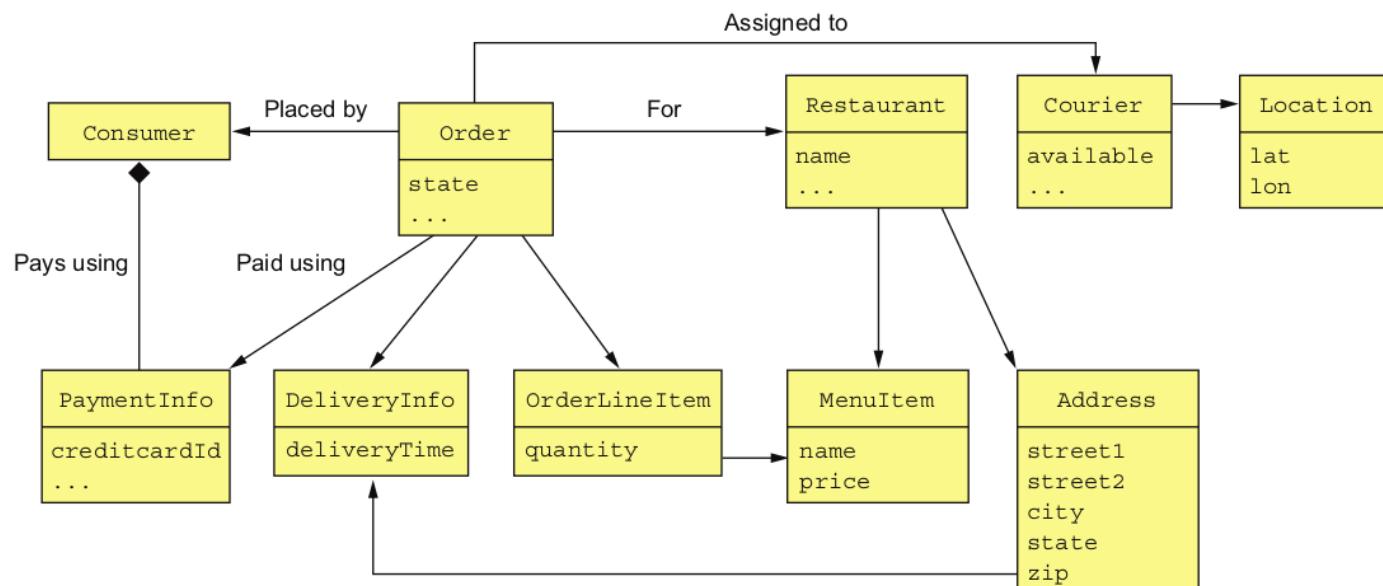
Le design est plus extensible

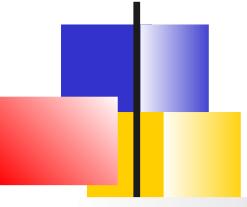
- On peut appliquer les patterns *Strategy* ou *Template* sur les classes du domaine

Limitation du Domain Model

Dans la conception traditionnelle orientée objet, un modèle de domaine est un ensemble de classes et de relations entre les classes. Les classes sont généralement organisées en packages.

Mais quelles classes font partie de l'objet *Order* ?



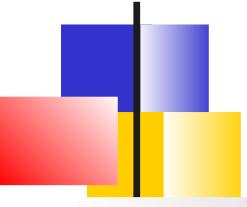


Frontières floues

L'absence de frontières dans le modèle objet peut créer des problèmes.

Exemple :

- *Order* a un invariant : Montant minimal
- 2 transactions travaillent sur le même *Order*
 - Chacune supprime des items dans l'*order*
 - Pour chacune des transactions, l'invariant est respecté. La transaction peut se valider
 - Mais au final, on se retrouve avec un *Order* ne respectant plus l'invariant



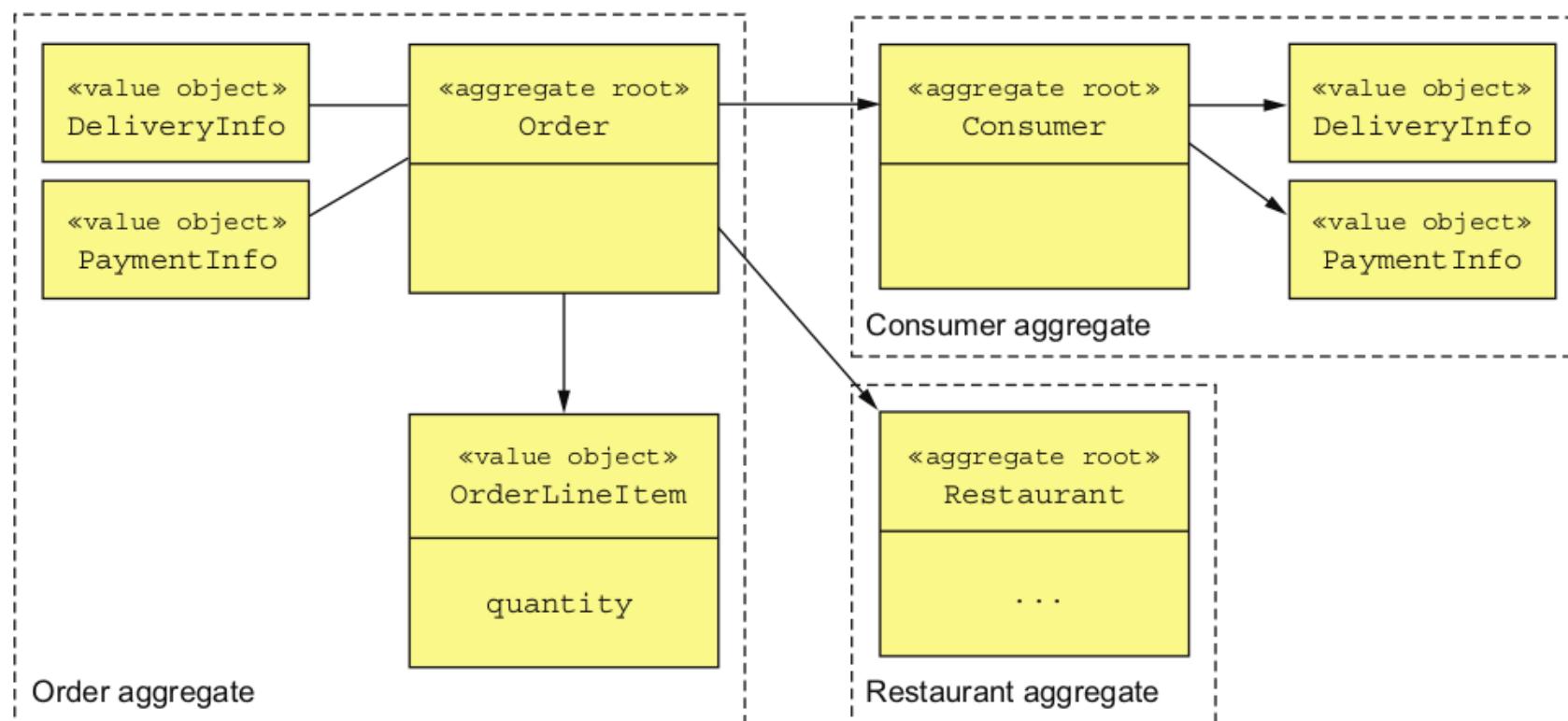
Aggregate Pattern

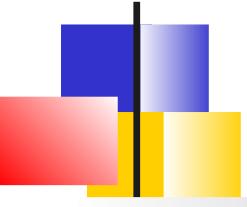
Aggregate Pattern¹ : Organise le modèle du domaine comme une collection d'agrégats, i.e. un graphe d'objets traité unitairement

- Les agrégats décomposent un modèle en morceaux, qui sont individuellement plus faciles à comprendre.
- Ils clarifient également la portée des opérations telles que le chargement, la mise à jour et la suppression. Ces opérations agissent sur l'ensemble de l'agrégat.
- La suppression d'un agrégat supprime tous ses objets d'une base de données.

1. https://martinfowler.com/bliki/DDD_Aggregate.html

Exemple





Avantages

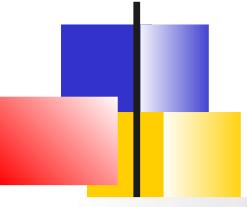
La mise à jour d'un agrégat entier plutôt que de ses parties résout les problèmes de cohérence

- Les opérations de mise à jour sont appelées sur l'agrégat racine qui applique les invariants.

La concurrence est gérée en verrouillant l'agrégat racine en utilisant, par exemple, un numéro de version ou un verrou au niveau de la base de données

=> Dans DDD, un élément clé de la conception d'un modèle de domaine consiste à identifier les agrégats, leurs limites et leurs racines.

Leur granularité aura des impacts sur la scalabilité de l'application et également la décomposition en micro-service



Règles sur les agrégats

Les agrégats doivent cependant respecter certaines règles :

- 1. Référencer seulement l'agrégat racine**

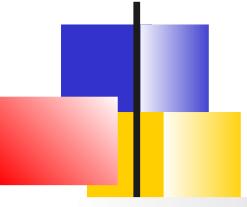
Cela garantit que l'agrégat puisse toujours vérifier ses invariants.

- 2. Les références inter-agrégats utilisent les clés primaires.**

L'utilisation d'identité plutôt que de références d'objet signifie que les agrégats sont faiblement couplés

- 3. UNE transaction créé ou met à jour UN agrégat.**

Les opérations devant mettre à jour plusieurs agrégats utilisent SAGA

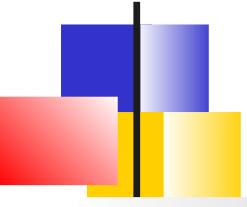


Événements métier

D'autres parties, (les utilisateurs, les autres services ou les autres composants) sont souvent intéressé par les changements d'états des agrégats.

Par exemple :

- Maintenir la cohérence des données entre les services à l'aide de sagas
- Notifier un service qui gère un réplica que les données source ont changé.
- Notifier une autre application afin de déclencher la prochaine étape d'un processus métier.
- Notifier un composant différent (Par exemple, mettre à jour l'index d'un moteur de recherche)
- Envoi de notifications (messages texte ou e-mails) aux utilisateurs.
- Faire du monitoring ou de l'analyse d'usage



Domain Event Pattern

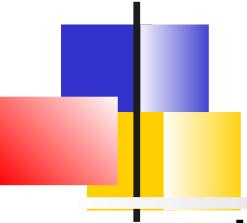
Domain event Pattern¹ : Un agrégat publie un événement domaine lorsqu'il est créé ou subit un changement significatif

Un événement domaine est nommé à l'aide d'un verbe au participe passé.

Il a en général

- Des propriétés qui caractérisent l'événement.
- Des méta-données comme un ID et un horodatage
- Quelque fois l'agrégat complet concerné

1. <https://microservices.io/patterns/data/domain-event.html>

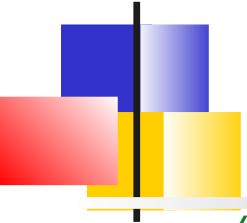


Génération et publication des évènements

Les événements doivent être créés puis typiquement publiés vers un message broker.

- La création de l'évènement concerne typiquement l'agrégat (*Domain Model Pattern*)
- Afin que l'agrégat n'est pas de dépendance sur l'API de messaging, il souhaitable que la classe Service qui peut profiter de l'injection de dépendance publie le message.

Tout cela fait partie d'une transaction locale qui implique la BD et le message Broker (*Transactional Outbox Pattern*)

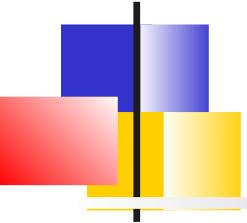


Exemple

```
// L'agrégat, instancie l'événement
public class Ticket {

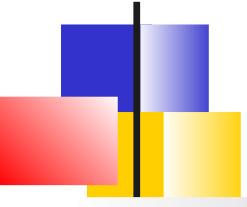
    public List<DomainEvent> accept(ZonedDateTime readyBy) {
        ...
        this.acceptTime = ZonedDateTime.now();
        this.readyBy = readyBy;
        return singletonList(new TicketAcceptedEvent(readyBy));
    }
}

// Le service connaît l'API de messaging
public class KitchenService {
    @Autowired
    private TicketRepository ticketRepository;
    @Autowired
    private DomainEventPublisher domainEventPublisher;
    public void accept(long ticketId, ZonedDateTime readyBy) {
        Ticket ticket = ticketRepository.findById(ticketId).orElseThrow(() ->
            new NotFoundEx(ticketId));
        List<DomainEvent> events = ticket.accept(readyBy);
        domainEventPublisher.publish(Ticket.class, ticketId, events);
    }
}
```



Design Logique métier

Introduction
Transaction Script Pattern
Patterns Orienté objet
Event Sourcing Pattern



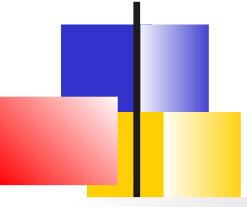
Introduction

Dans les exemples précédents, la logique de publication d'événements est imbriquée avec la logique métier.

La logique métier continue de fonctionner même si un développeur oublie de publier un événement.

– source de bugs ?

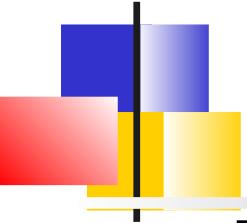
Pourrait-on avoir la garantie que dès qu'un agrégat est créé ou mis à jour, un événement est publié ?



Event sourcing Pattern

Event sourcing Pattern¹ : Persiste un agrégat comme une séquence d'événements du domaine représentant les changements d'état

=> Une application recrée l'état courant d'un agrégat en rejouant les événements



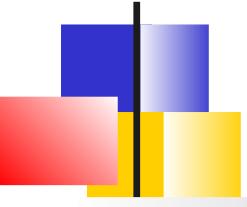
Bénéfices/Inconvénients

Bénéfices

- Préserve l'historique des agrégats, idéal pour l'audit
- Publie de façon sûre les événements métier, idéal pour les micro-services

Inconvénients

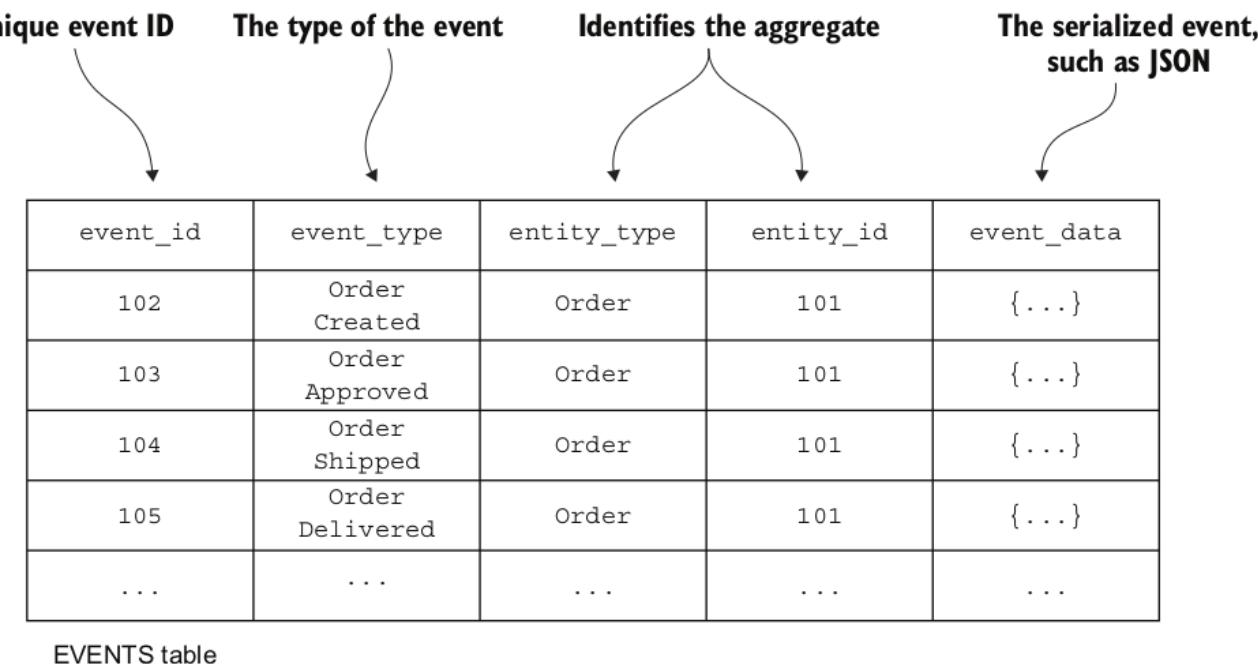
- Temps d'apprentissage long car approche très différente
- Les requêtes sur la base d'événements sont plus difficiles (Voir CQRS Pattern)



Event Store

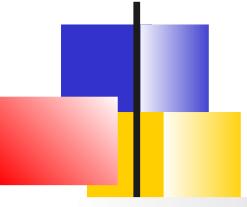
Au lieu de stocker, l'agrégat dans un schéma traditionnel classique.

L'agrégat est stocké sous forme d'événements dans un ***EventStore***



event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{...}
103	Order Approved	Order	101	{...}
104	Order Shipped	Order	101	{...}
105	Order Delivered	Order	101	{...}
...

EVENTS table

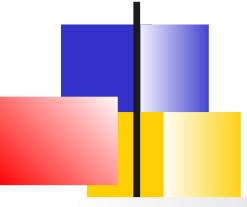


Chargement d'un agrégat

Le chargement d'un agrégat est constitué de plusieurs étapes :

- Charger les événements de l'agrégat
- Créer un agrégat avec le constructeur par défaut
- Rejouer les événements

En programmation fonctionnelle, c'est une fonction ***reduce***



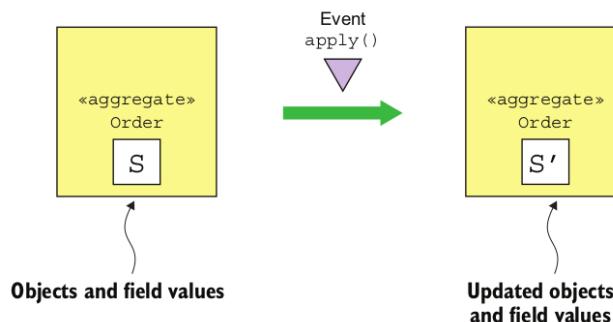
Contraintes sur les événements

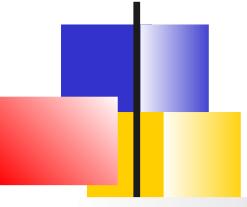
Les événements ne sont plus optionnels.

Toute modification ou création de l'agrégat doit se traduire en un événement métier, qu'il y ait ou pas des consommateurs

L'événement doit contenir toutes les données que l'agrégat nécessite pour faire sa transition d'état.

=> Certains événements auront peu de données (changement d'un statut) , d'autres beaucoup :
Ex. Création initiale de l'agrégat

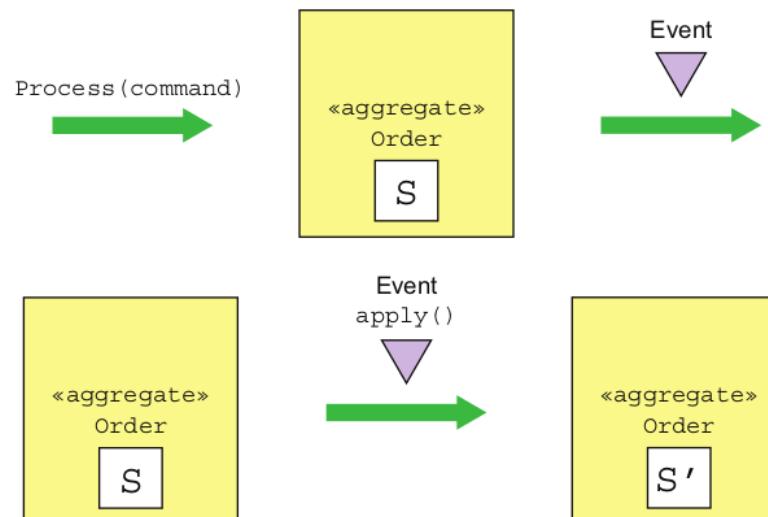


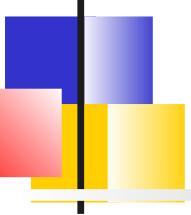


Traitement d'une requête

Le traitement métier des requêtes est également restructuré.

- Les méthodes de commande génèrent déjà l'événement à partir de la commande (ou génère une exception si les règles métier ne sont pas respectées)
process(command)
- L'événement est ensuite appliqué à l'agrégat (méthode *apply*)





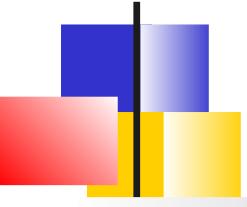
Exemple

```
public class Order {  
  
    public List<Event> process(ReviseOrder command) {  
        OrderRevision orderRevision = command.getOrderRevision();  
        switch (state) {  
            case AUTHORIZED:  
                LineItemQuantityChange change =  
                    orderLineItems.lineItemQuantityChange(orderRevision);  
                if (change.newOrderTotal.isGreaterThanOrEqualTo(orderMinimum)) {  
                    throw new OrderMinimumNotMetException();  
                }  
                return singletonList(  
                    new OrderRevisionProposed(  
                        orderRevision, change.currentOrderTotal,  
                        change.newOrderTotal));  
  
            default:  
                throw new UnsupportedStateTransitionException(state);  
        }  
    }  
}
```

Returns events without updating the Order

```
public class Order {  
  
    public void apply(OrderRevisionProposed event) {  
        this.state = REVISION_PENDING; //-----  
    }  
}
```

Applies events to update the Order



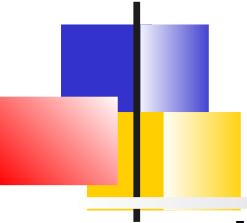
Séquences

Création :

- 1) Instancier l'agrégat via constructeur par défaut.
- 2) Appeler *process()* pour générer les événements de création.
- 3) Mettre à jour l'agrégat en itérant sur les événements et en appelant leur méthode *apply()* correspondantes.
- 4) Sauvegarder les événements dans l'*event store*.

Mise à jour :

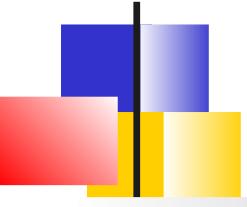
- 1) Charger les événements de l'agrégat à partir de l'*event store*.
- 2) Instancier l'agrégat via constructeur par défaut
- 3) Itérer sur les événements chargés et appeler leurs méthodes *apply()*.
- 4) Invoquer la méthode *process()* pour générer de nouveaux événements
- 5) Mettre à jour l'agrégat en itérant sur les nouveaux événements et en appelant leur méthode *apply()* correspondante.
- 6) Sauvegarder les nouveaux événements dans l'*event store*.



Optimistic-Locking

Un *event store* peut également utiliser les techniques d'***optimistic locking*** pour gérer des mises à jour concurrentes.

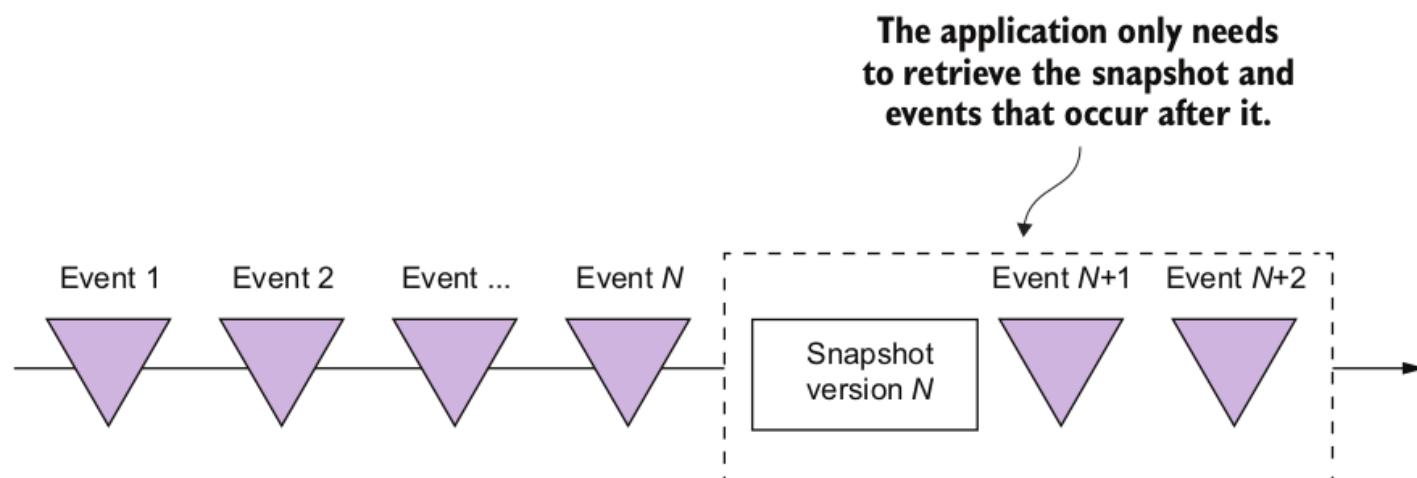
- Lors de l'ajout d'évènement, le store vérifie que l'agrégat n'a pas été modifié entre temps via un n° de version.
- Le n° de version peut tout simplement le nombre d'événements liés à l'agrégat.

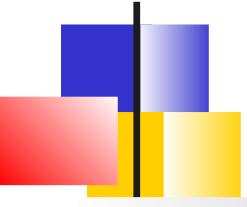


Snapshots

Certains agrégats peuvent avoir de longue durée de vie, et donc un grand nombre d'événements associés.

Afin de pas à avoir à recharger tout l'historique pour reconstruire l'état d'un agrégat, une solution commune est de stocker périodiquement des **snapshots**.





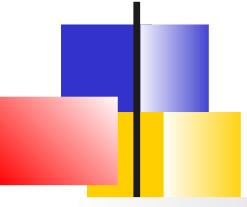
Evolutivité, *upcaster*

L'*event-sourcing* stocke les événements pour toujours, cependant la structure des événements change avec les évolutions de l'application.

=> Certains changements sont compatibles (ajout d'une nouvelle colonne) d'autres non (suppression ou renommage)

Afin de gérer, les changements incompatibles, un composant, appelé ***upcaster***, met à jour des événements individuels d'une ancienne version vers une version plus récente.

L'*event store* lui n'est pas migré à la différence d'une BD traditionnelle



Bénéfices

Publie de façon sûre les événements métier et fournit un audit log fiable.

La persistance et la notification ne font plus qu'un

Préserve l'historique des agrégats.

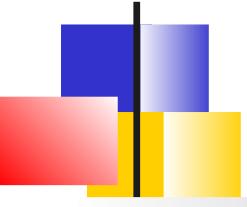
On peut se repositionner sur un état passé.

Évite le problème du passage entre BD et Objet
(O/R impedance mismatch problem).

Les événements sont de simples Objets JSON

Fournit aux développeurs une machine à remonter le temps

Correction de bugs à posteriori !!



Inconvénients

Modèle de programmation différent avec une courbe d'apprentissage pas évidente

Complexité d'une application basée sur la messagerie.

Attention au doublons (At-least-once des message brokers) !!

Idempotence

L'évolution des événements peut être délicate.

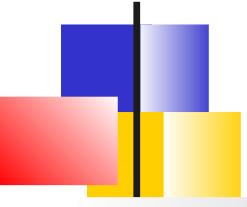
Services dépendants doivent rapidement s'adapter avec des *upcasters*

La suppression de données est délicate.

Comment être en accord avec le droit européen¹ donnant à tous citoyen le droit d'effacer ses données. L'email de la personne est peut être dans plein d'évènements de l'EventStore

Le requêtage de l'event-store est difficile.

Obliger d'utiliser l'approche CQRS

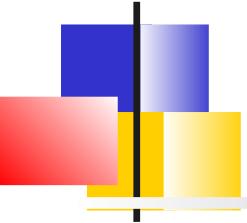


Sagas et Event-Sourcing

Les services distribués ont souvent besoin d'initier et de participer à des sagas pour maintenir la cohérence des données entre les services.

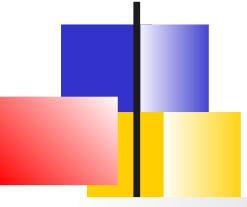
L'*event-sourcing* facilite l'utilisation des sagas basées sur la ***chorégraphie***.

- Les participants échangent les événements de domaine émis par leurs agrégats.
- Les agrégats de chaque participant gèrent les événements en traitant des commandes et en émettant de nouveaux événements



Requête

API Composition Pattern
CQRS Pattern



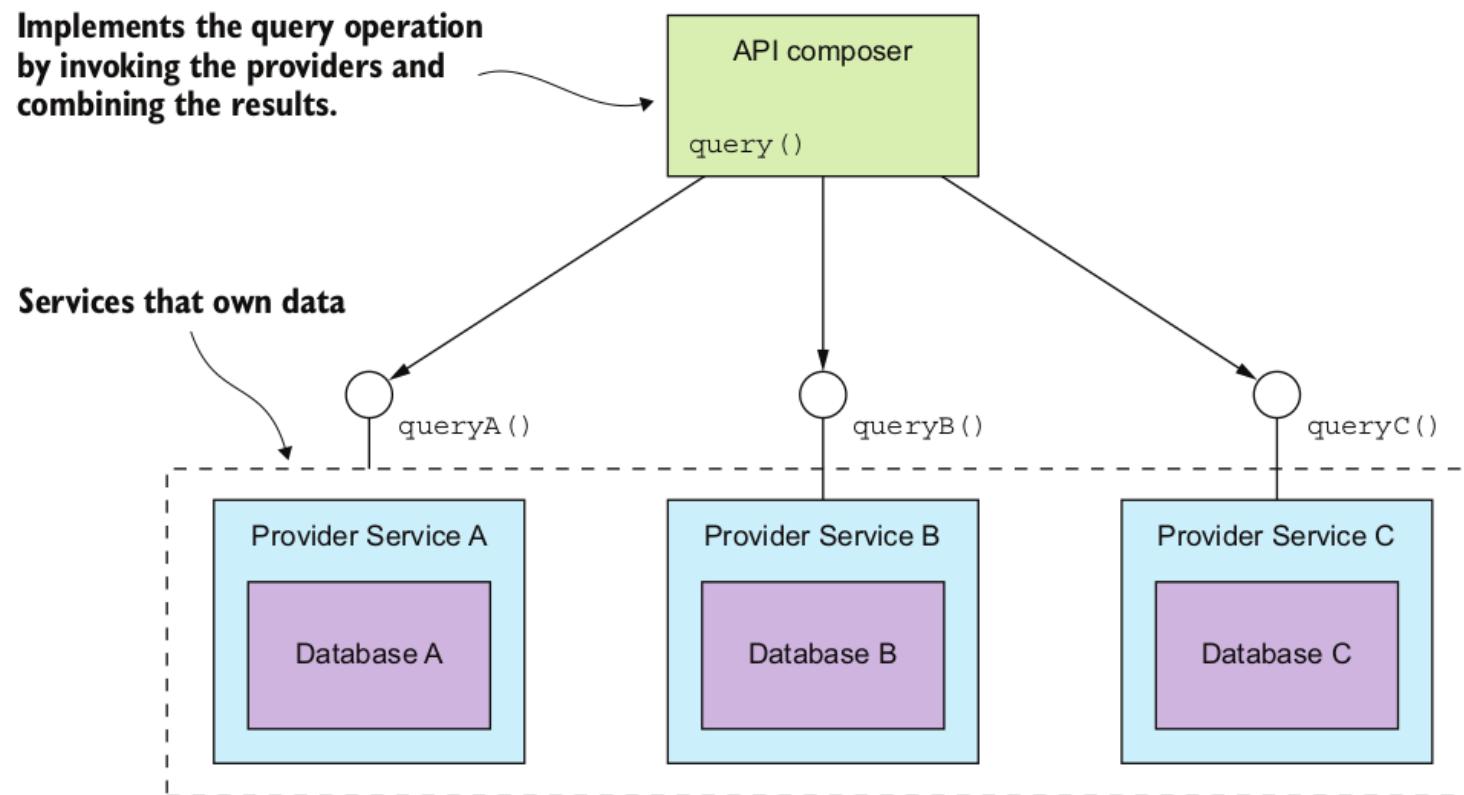
Introduction

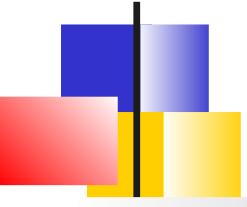
Les requêtes doivent souvent récupérer des données dispersées parmi les bases de données appartenant à plusieurs services.

2 patterns principaux :

- ***API composition pattern*** : La plus simple des approches consiste à développer un service combinant plusieurs serviceshe results.
- ***Command query responsibility segregation (CQRS) pattern*** : Plus puissant mais plus complexe, il maintient des vues dédiées aux requêtes

API Composition Pattern





API Composition

C'est le pattern le plus simple à placer et qu'il faut privilégier.

Attention cependant :

- Il se peut que l'agrégateur soit obligé d'effectuer une jointure mémoire sur de large volumes de données
- Certaines opérations de requêtage ne peuvent pas être implémentées par ce pattern

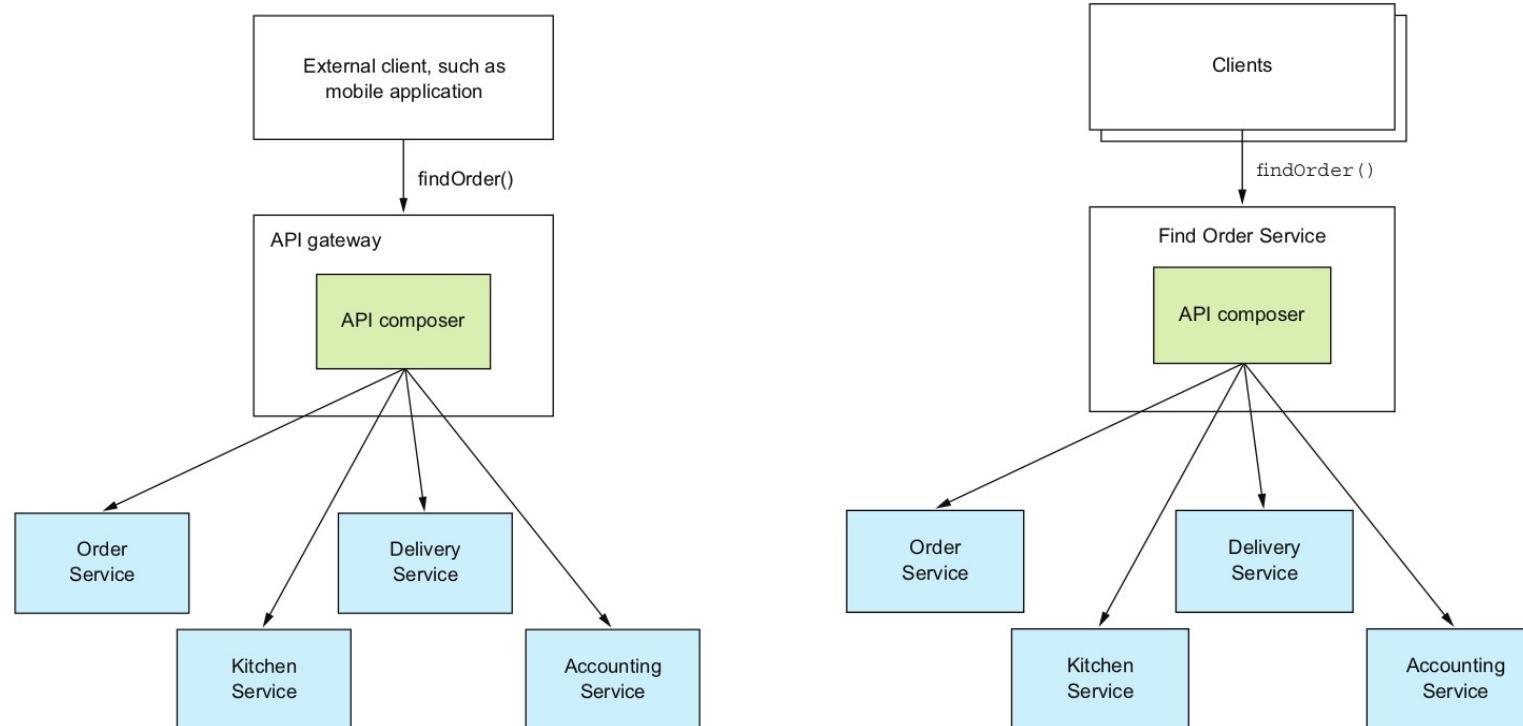
Composer

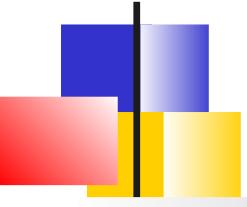
Où placer le Composer ?

Généralement dans une Gateway ou un service dédié

Le modèle de programmation ?

Modèle réactif bien sûr





Inconvénients

Avantages de la simplicité mais :

Surcharge des traitements

A la différence des monolithiques, la requête est composées de plusieurs requêtes et de la composition mémoire

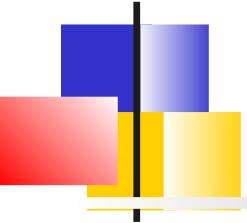
Réduction de la disponibilité :

Plus de services impliqués, plus de services susceptibles d'être en panne.

=> Utilisation de cache ?

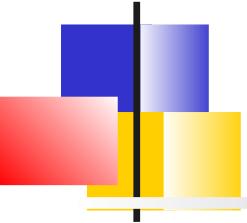
Manque de cohérence de données :

Au moment de la requête, certains services sont peut être en cours d'une SAGA et peuvent avoir des données incohérentes



Requête

API Composition Pattern
CQRS Pattern



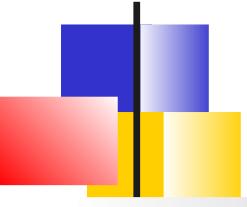
CQRS

Command query responsibility segregation Pattern

segregation Pattern¹ : Implémenter une requête qui a besoin des données de plusieurs services en utilisant des événements pour conserver une vue en lecture seule qui réplique les données des services.

CQRS maintient une ou plusieurs vues des bases de données qui implémentent une ou plusieurs requêtes de l'application

1. <http://microservices.io/patterns/data/cqrs.html>



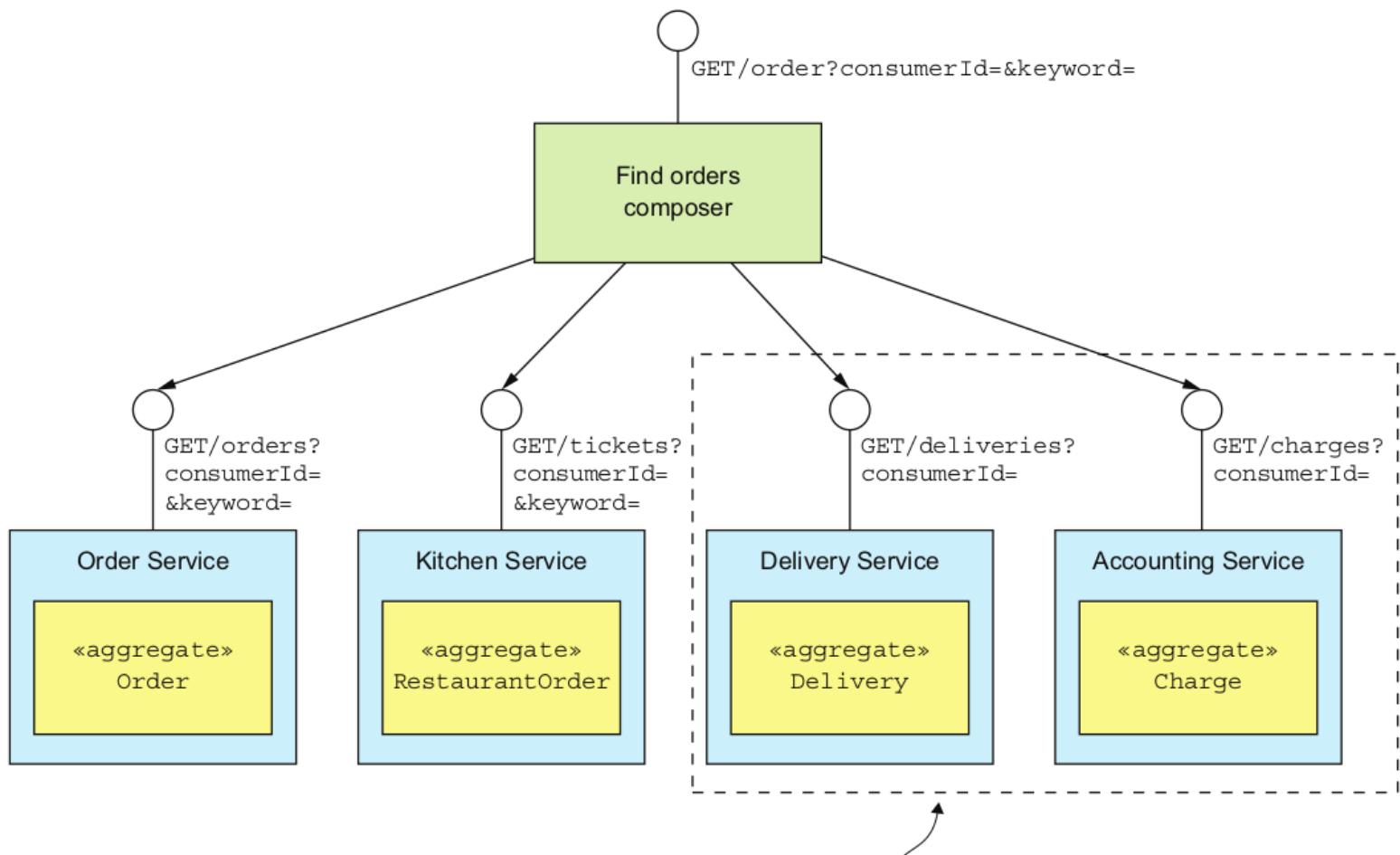
Motivation pour CQRS

Imaginer une requête qui prend un filtre dont les attributs ne concernent pas tous les services.

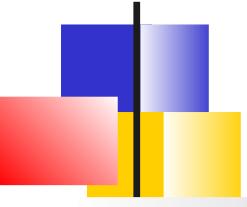
- Par ex : Rechercher les commandes depuis une certaine date dont le restaurant associé ou les entrées du menu correspondent au mot clé «mafé »

Ce type de requête est très difficile à implémenter via le pattern API composition, il va en tout cas l'obliger à parcourir un très large jeu de données

Pb de l'API Composition Pattern



These services don't store the data needed for a keyword search, so will return all of a consumer's orders.



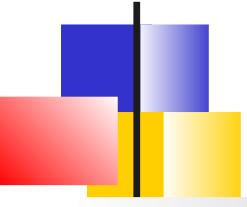
Motivation pour CQRS

Même si la requête ne concerne qu'un seul service, elle peut être difficile à implémenter avec une simple BD :

- Recherche full-texte
- Recherche géo-spatial

Dans ce cas, le service impliqué doit stocker les informations dans d'autres support de persistance que la BD et s'assurer que les 2 supports restent cohérents

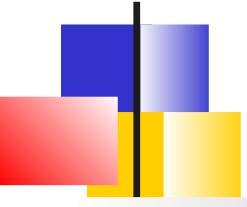
Cela revient à maintenir la cohérence entre des données répliquées => CQRS



Motivation pour CQRS

Dernière motivation pour CQRS : la séparation des responsabilités (Separation Of Concerns)

- Soit 2 services *OrderService* et *RestaurantService* s'occupant de leurs agrégats respectifs.
- *OrderService* aimerait avoir une requête recherchant les restaurants disponibles pour un *Order* spécifique.
- A priori, ce n'est pas à *RestaurantService* d'implémenter la logique de requête demandé par *OrderService*

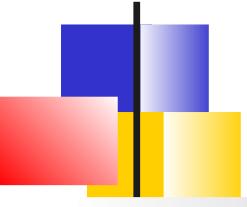


CQRS

CQRS, comme son nom l'indique, est une question de séparation de responsabilité: Il sépare les commandes (opération d'écriture) des requêtes (lecture)

Il divise le modèle de données en 2 :

- 1 pour les opérations CUD (Create, Update, Delete)
- 1 pour les opérations de requêtage (R)



Fonctionnement

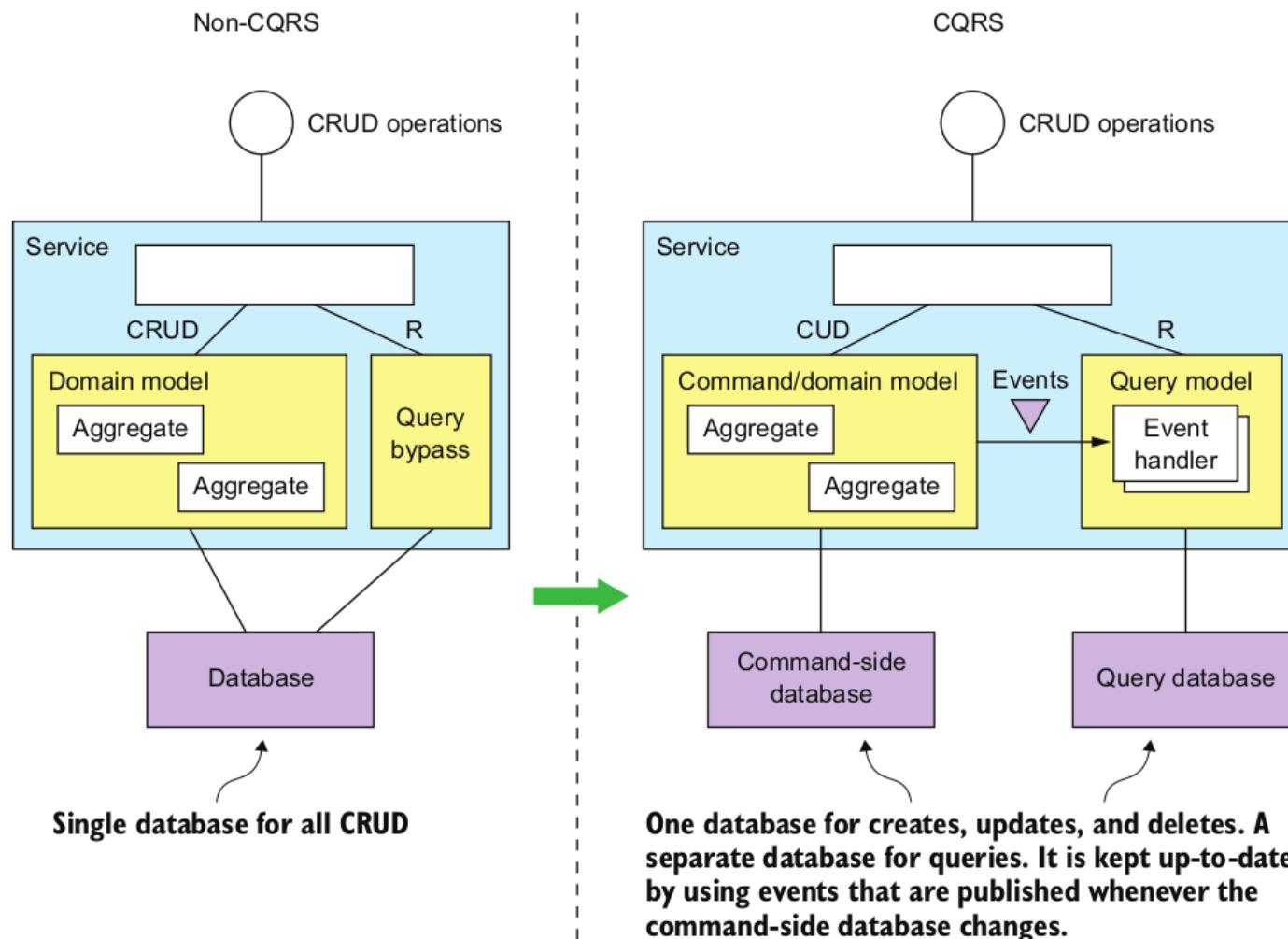
Dans un service basé sur CQRS, le modèle de domaine côté commande

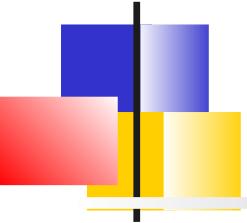
- gère les opérations CRUD et est mappé à sa propre base de données.
- gère des requêtes simples, telles que des requêtes basées sur une clé primaire sans jointure.
- publie des événements de domaine chaque fois que ses données changent

Le modèle de requête distinct :

- gère les requêtes non triviales.
- utilise une BD adaptées aux requêtes qu'il doit prendre en charge.
- comporte des gestionnaires d'événements qui s'abonnent aux événements de domaine et mettent à jour sa base de données.
- Plusieurs modèles de requête peuvent exister, un pour chaque type de requête.

Non-CQRS vs CQRS





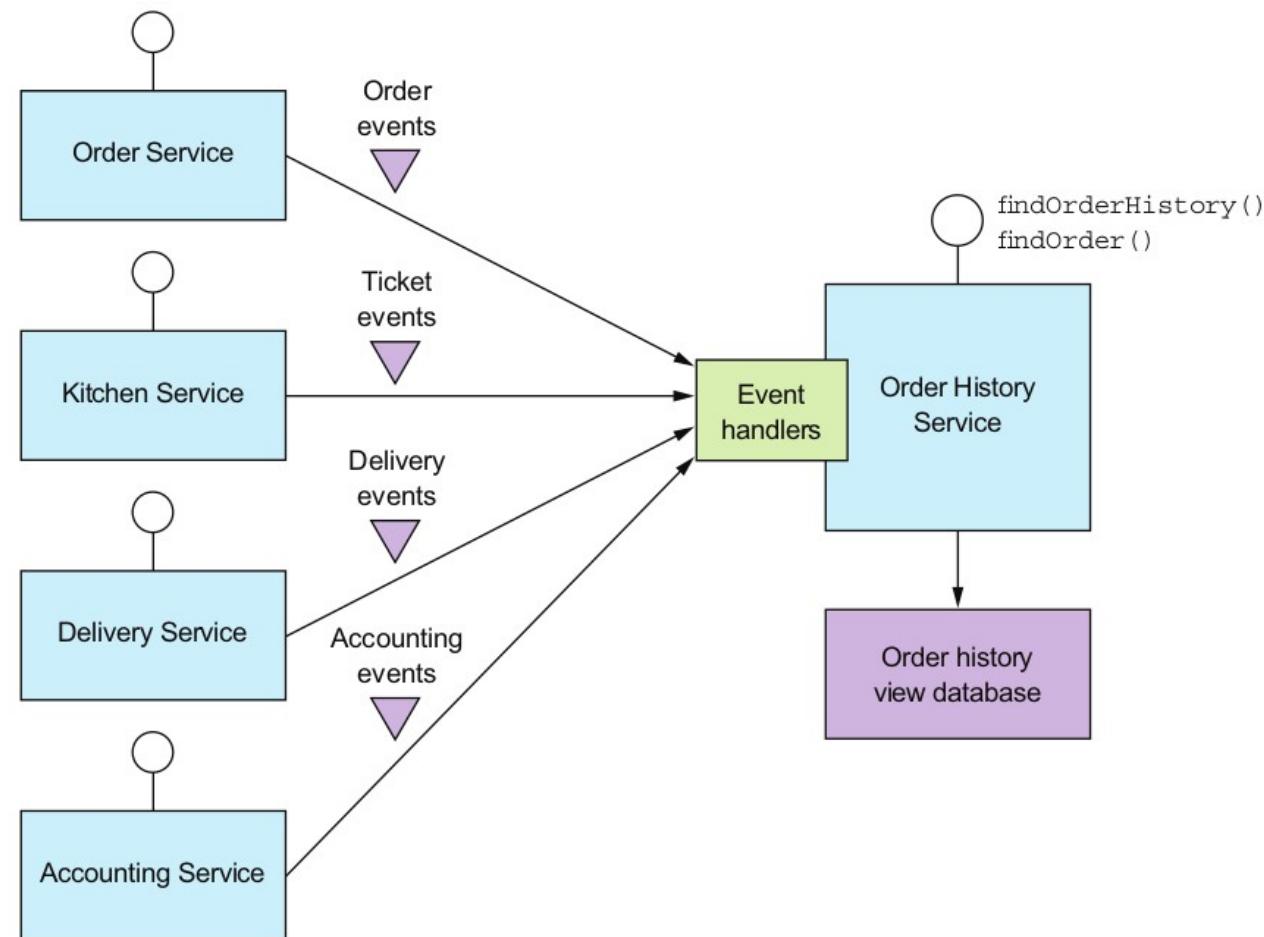
Service *Query-only*

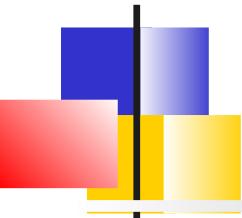
CQRS peut être appliqué au sein d'un service, mais également utiliser pour définir des services de requête.

Un service de requête a une API composée uniquement d'opérations de requête.

Il implémente des opérations d'interrogation en interrogeant une base de données qu'il tient à jour en s'abonnant aux événements publiés par un ou plusieurs autres services.

Exemple





Bénéfices de CQRS

Permet la mise en œuvre efficace des requêtes dans une architecture de microservices

Implémente efficacement des requêtes qui font intervenir plusieurs services

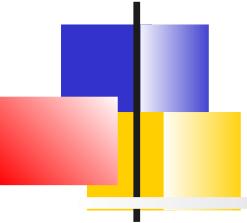
Permet la mise en œuvre efficace de diverses requêtes
Requêtes full-text, requête géo-spatial, IA

Rend l'interrogation possible dans une application basée sur EventSourcing

EventSourcing ne permet que les requêtes basées sur les clés-primaires, CQRS est alors une obligation

Améliore la séparation des responsabilités

Chaque micro-service se consacre exclusivement à ses propres responsabilités métier



Inconvénients de CQRS

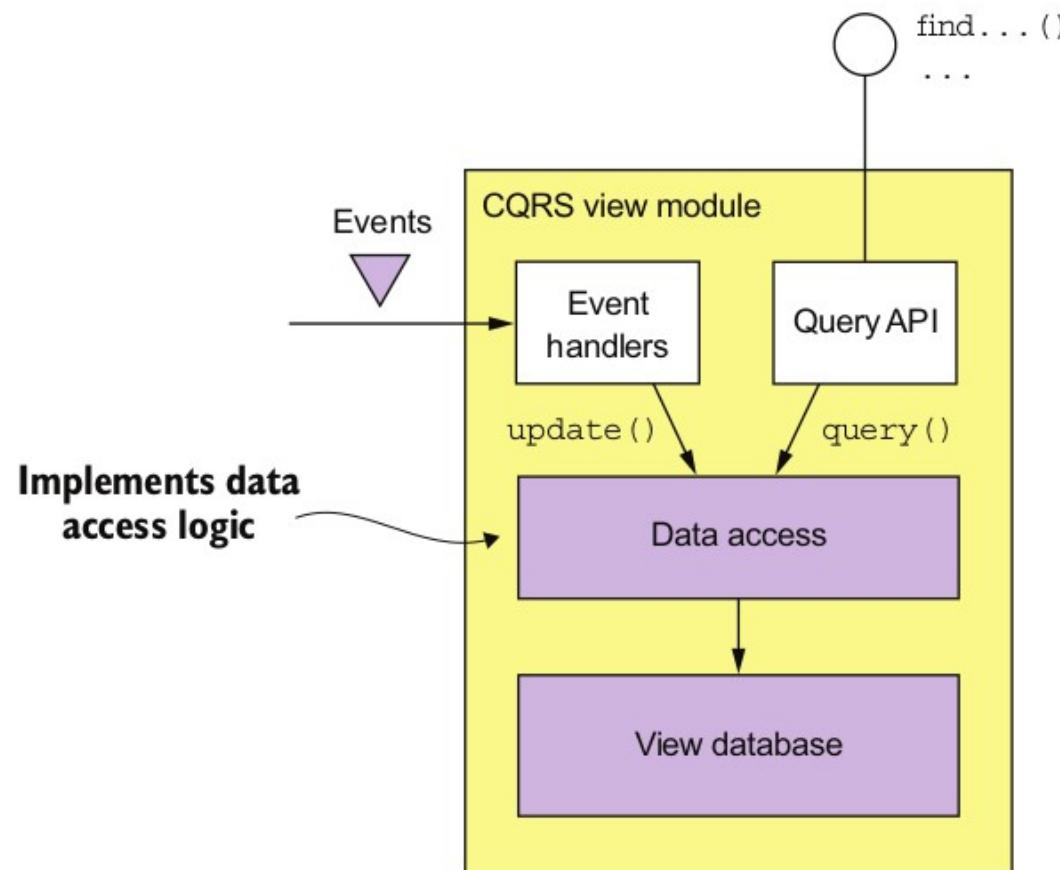
Architecture plus complexe

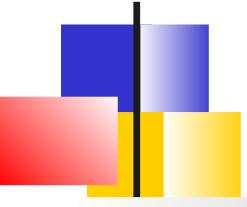
Code nécessaire pour mettre à jour les vues,
exploitation de plusieurs technologies de
stockage

Obliger de traiter avec le délai de réplication (replication lag)

Les opérations d'écriture sont visibles dans les
requêtes après le temps de traitement des
événements.

Design QueryService

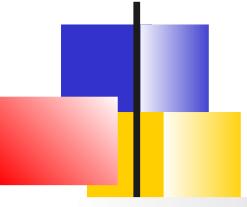




Choix de design

Lors de la mise en place des services de requêtage plusieurs choix doivent être fait :

- Choix de la base : En fonction des types de requêtes que l'on doit supporter : NoSQL, Full-Text, SQL
- Module DAO : Gestion de la concurrence des mises à jour, Idempotence des gestionnaires d'événements ou détection de doublons, Gestion du lag



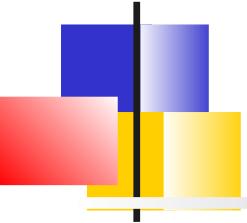
Evolutivité

De nouvelles vues ou des mises à jour de vue apparaîtront lors des évolutions de l'application.

A priori, le nouveau schéma de vue doit être recréé à partir de la base des événements. Le problème c'est que le message broker n'est pas là pour stocker indéfiniment tous les événements¹ et que ce traitement peut devenir très long

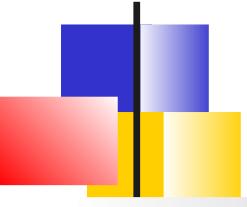
- Utilisation de base d'événements archivés BigData
- Construction incrémentale des vues avec des snapshots

1. *Même Kafka a généralement une période de rétention limitée*



API Externes

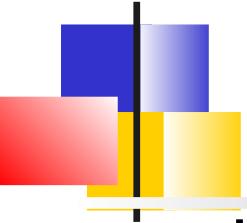
API gateway pattern



Introduction

L'architecture micro-service a potentiellement de nombreux clients différents (Application Mobile, Web, Applications de partenaires, ...) qui nécessitent différentes données.

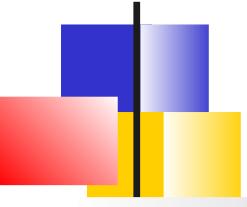
Il est donc difficile d'offrir une API unique qui conviennent à tous les clients



Inconvénients des appels directs

L'invocation directe des services par les clients a de nombreux inconvénients :

- Les API des services ayant une granularité fine, cela oblige les clients à faire plusieurs requêtes pour récupérer les données dont ils ont besoin
- Le manque d'encapsulation du fait que les clients connaissant chaque service et son API rend difficile le changement d'architecture et d'API.
- Les services peuvent utiliser des mécanismes IPC qui ne sont pas pratiques pour des clients à l'extérieur du firewall



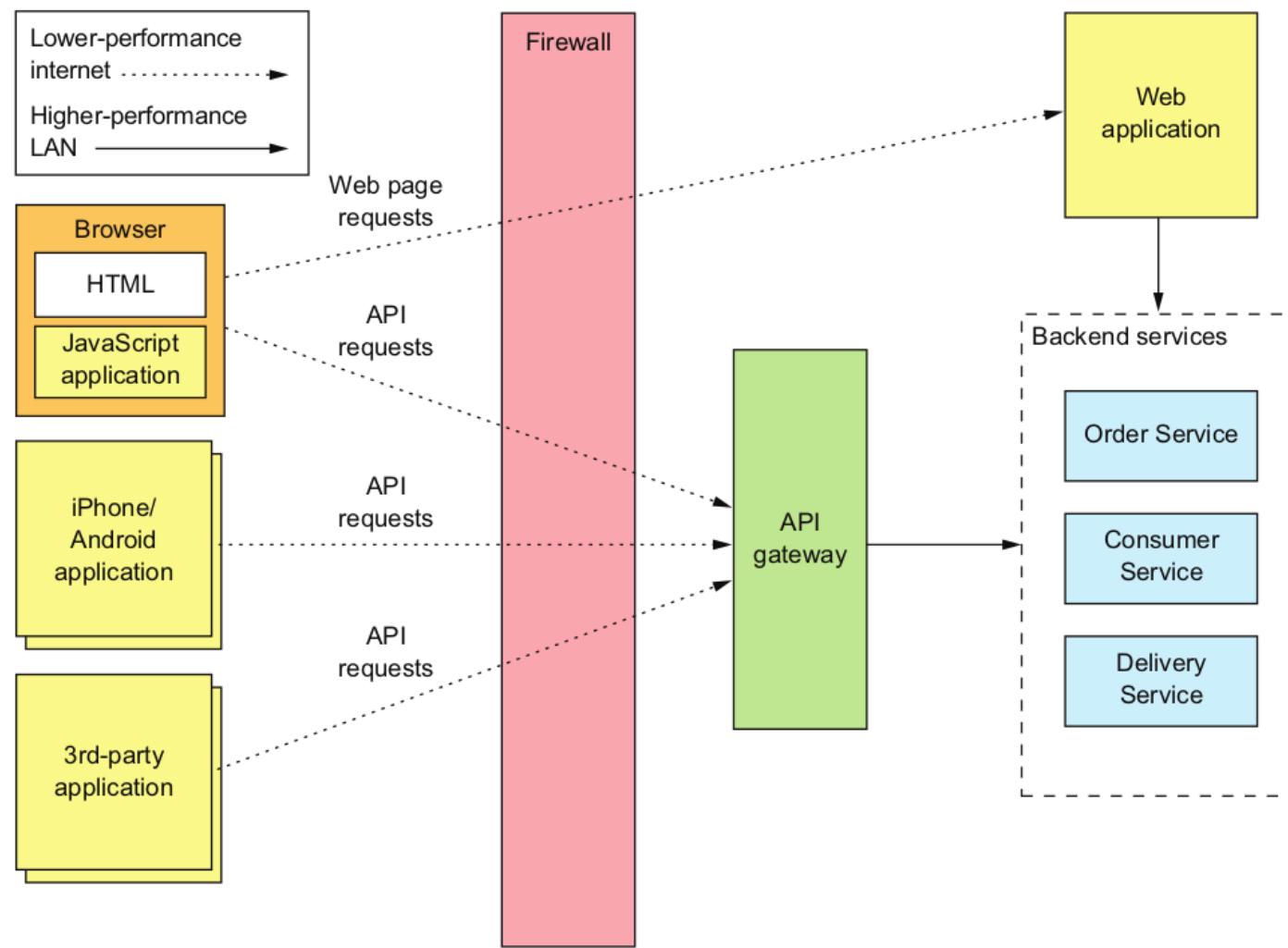
API Gateway Pattern

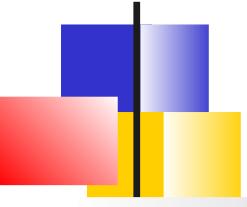
API gateway Pattern¹ : Implémente un service qui est le point d'entrée de l'application micro-service pour les clients externes

Le service API Gateway est alors responsable du routage des requêtes, de la composition d'API, de la traduction de protocole, de l'authentification et d'autres fonctions

Similaire au pattern *Facade* en Objet

API Gateway





Fonctions de la Gateway

Routage : En fonction d'un table de routage, la gateway transfère les requêtes aux services backend. La table de routage peut s'appuyer sur tous les composants HTTP (URL, entêtes, paramètres de requête)

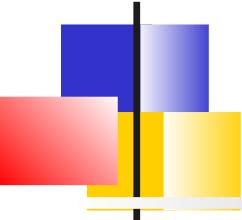
Composition d'API : Plusieurs appels vers les services backend sont alors agrégés

Traduction de protocoles : Traduction de requêtes GraphQL en requêtes REST par exemple

API spécifique par clients : La gateway n'offre pas la même API pour un mobile que pour les partenaires externes

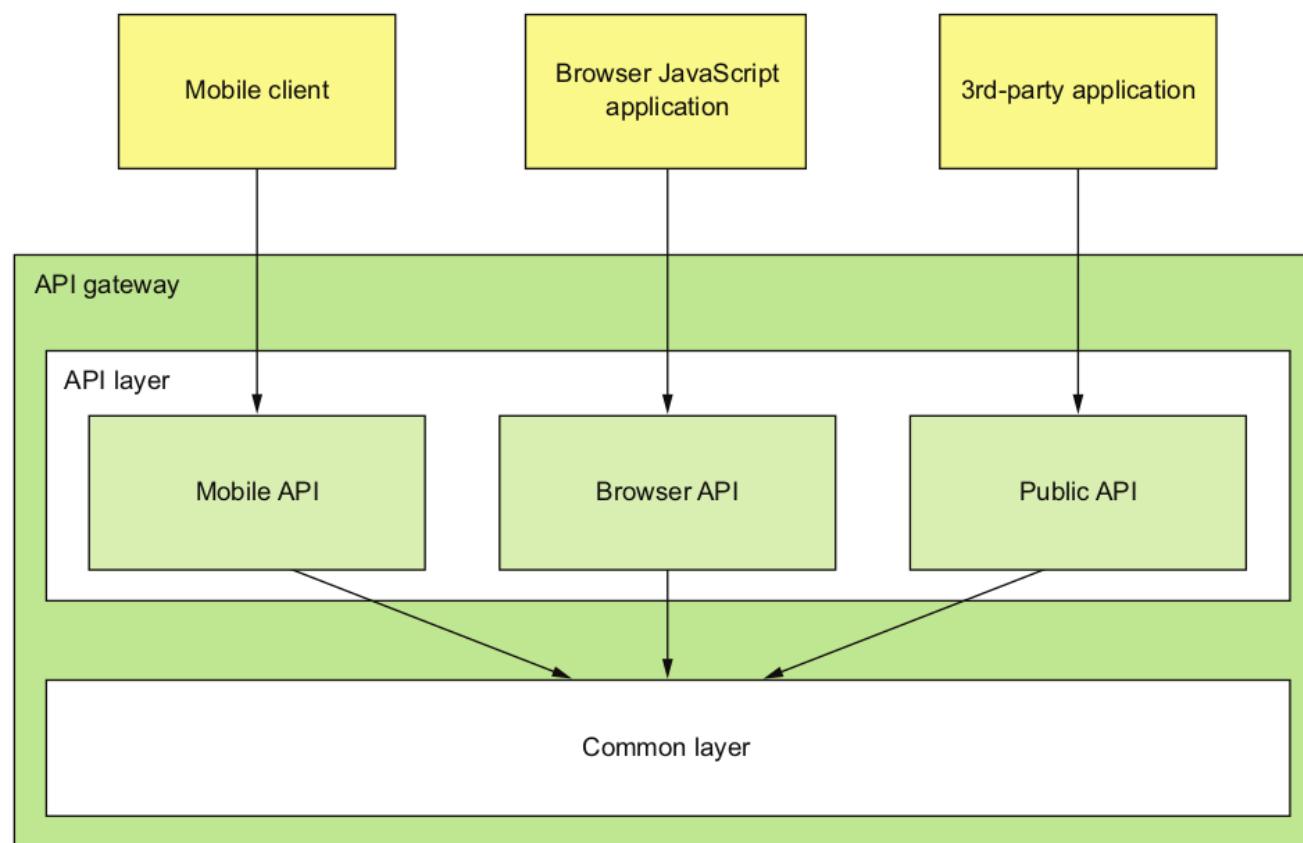
Fonctions transverses (edge functions)¹ : Implémentation de l'authentification, de l'autorisation, du cache, du lod de requête,

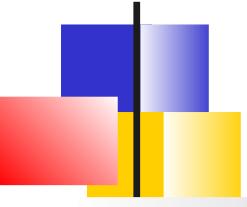
1. On peut également implémenter ces fonctions dans un service dédié en amont de la gateway



Architecture de la Gateway

Chaque opération de l'API est :
soit un simple routage
Soit une composition





Organisation

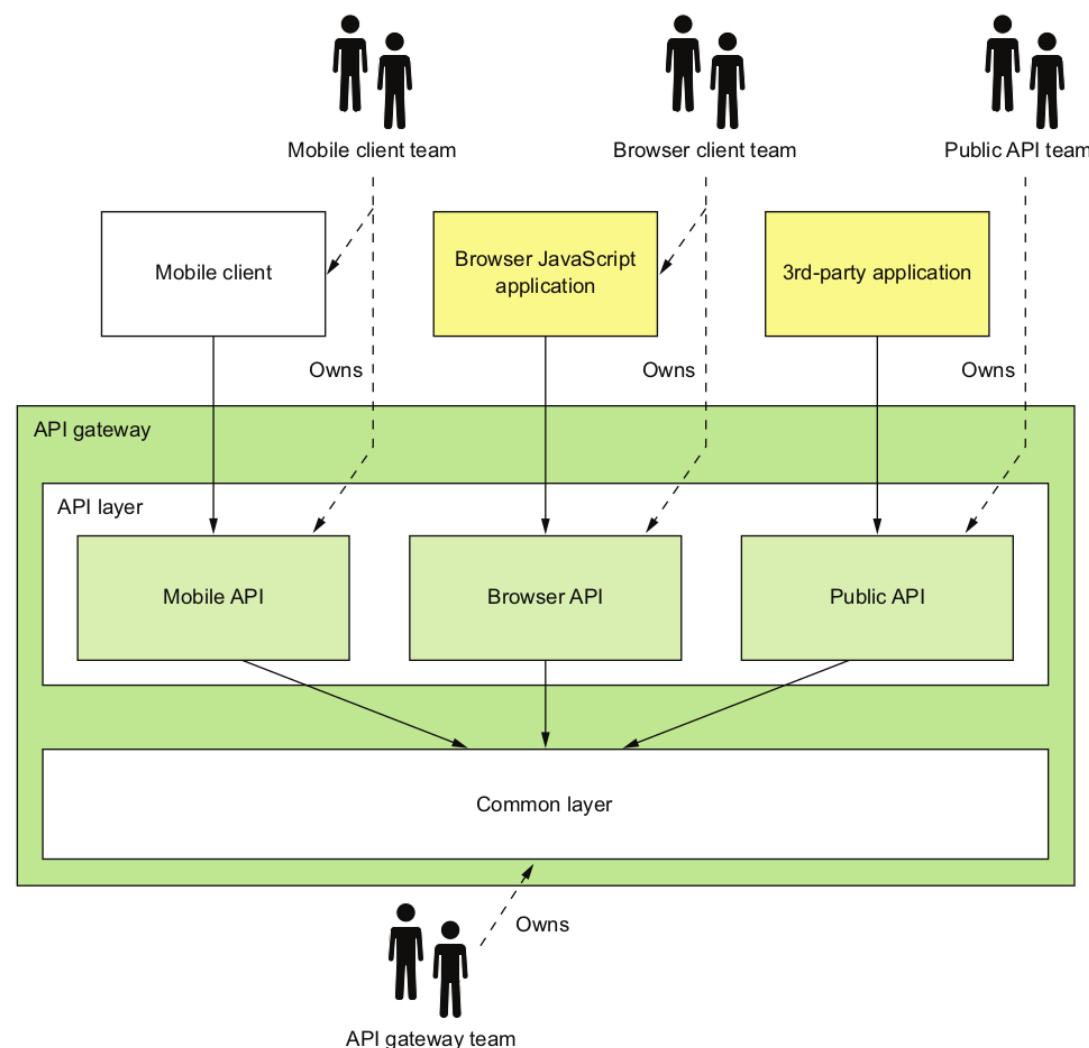
Qui est responsable du développement et de l'exploitation de la gateway ?

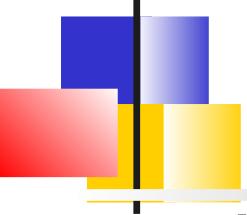
Une approche efficace, promue par Netflix, consiste de confier le module API d'un client à l'équipe responsable du code client (les équipes d'API mobiles, Web Javascript, ...).

Une équipe Gateway est responsable du module Commun et de l'exploitation de la gateway.

Lorsqu'une équipe cliente doit modifier son API, elle committe ses modifications dans le référentiel source de la Gateway et une pipeline de déploiement continue valide la cohérence.

Organisation





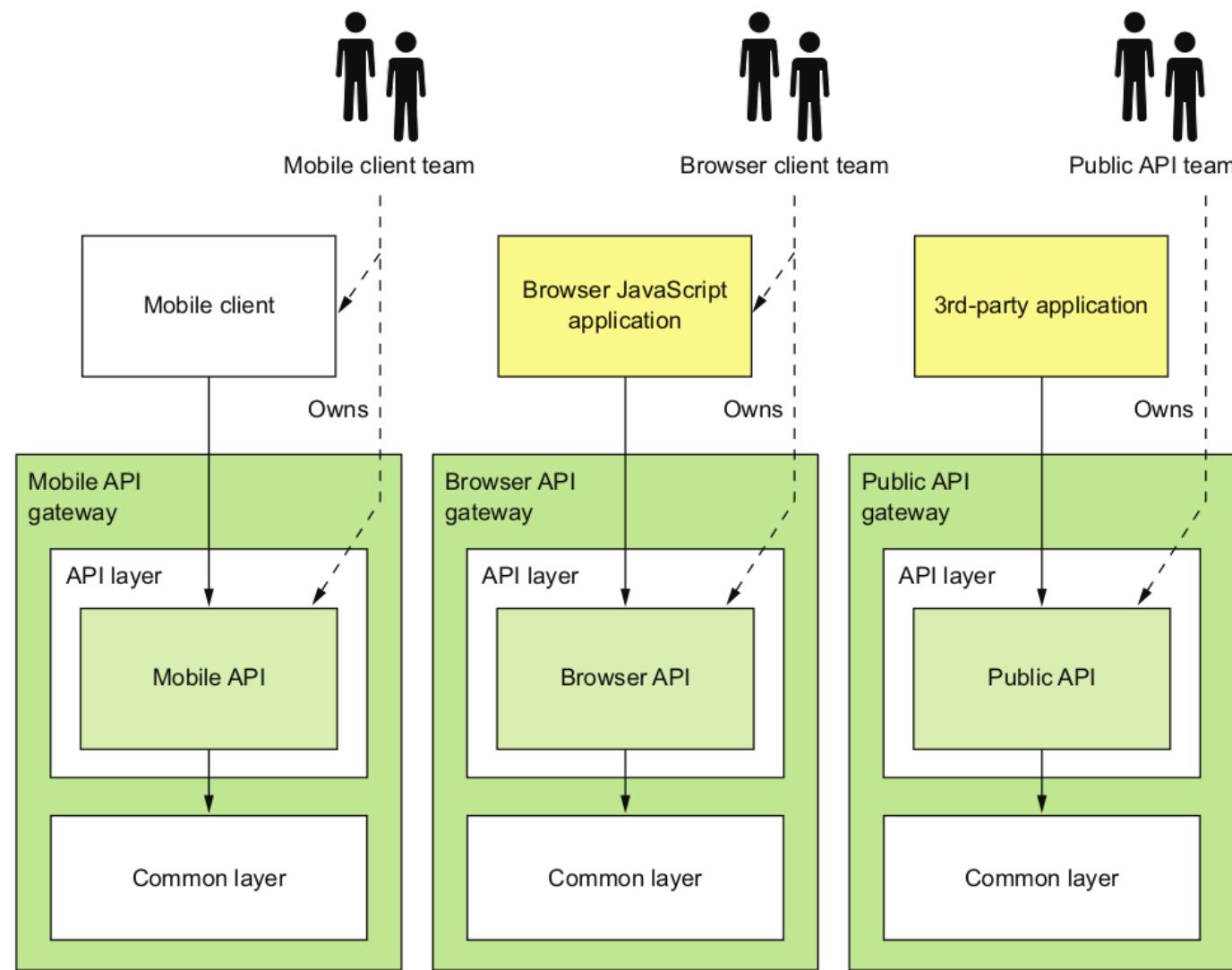
Backends for frontends Pattern

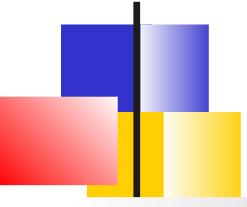
L'organisation précédente est contraire aux principes des micro-services.
Chaque équipe doit être indépendante.

=>

Backends for frontends Pattern¹ :
Implémenter une gateway différente pour chaque type de client.

1. <http://microservices.io/patterns/apigateway.html>





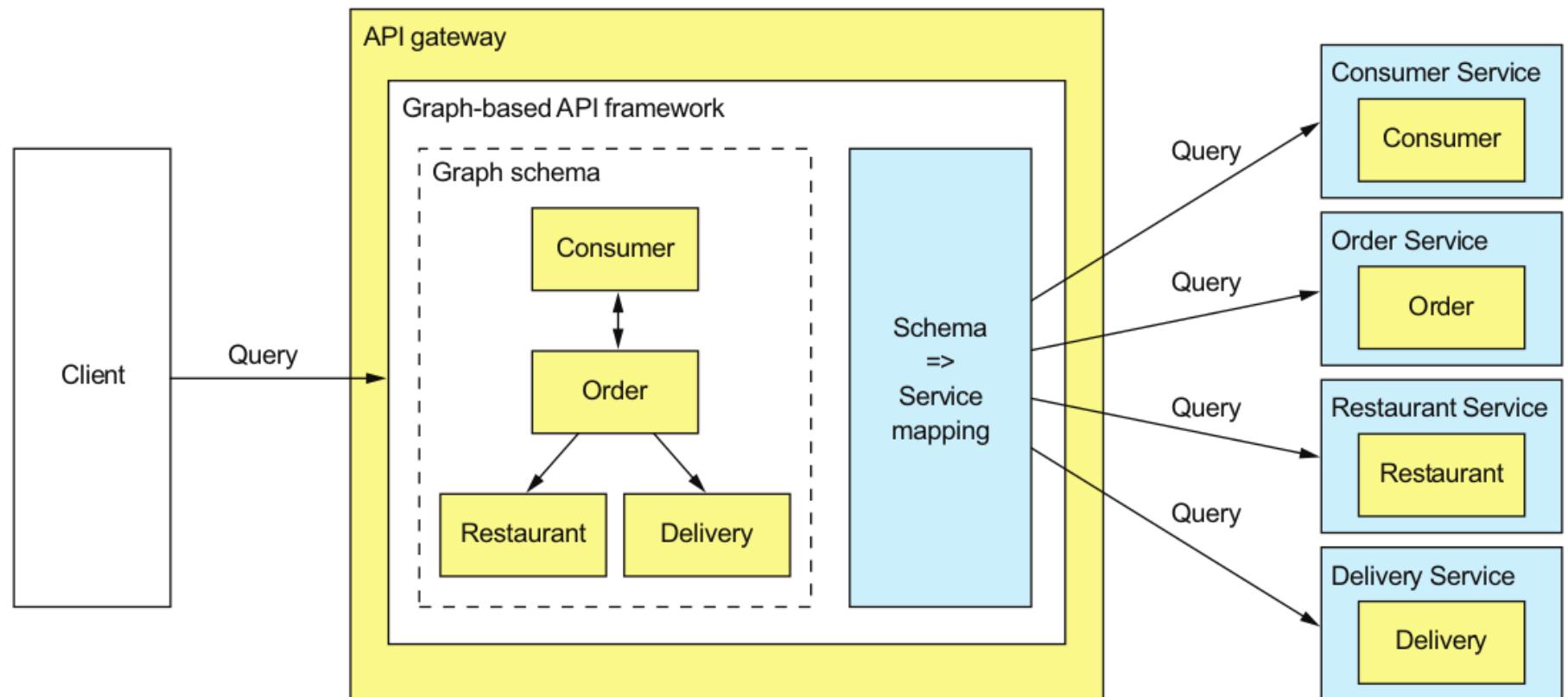
Alternatives à REST

Un autre alternative afin que chaque client puisse avoir sa propre API est d'utiliser des technologies comme GraphQL ou Netflix Falcor.

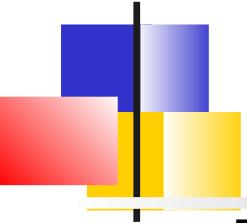
Ces technologies sont une alternative au modèle REST.

Le client peut demander le graphe de données qu'il veut obtenir dans sa requête

Exemple GraphQL Gateway



<https://medium.com/swlh/building-graphql-gateway-with-springboot-framework-251f92cdc99e>
<https://github.com/joaquin-alfaro/graphql-gateway>



Bénéfices / Inconvénients

Bénéfices

Encapsule la structure interne de l'application

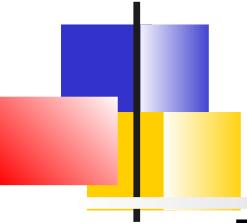
Réduit les aller-retours entre le client et
l'application

Inconvénients

Un autre service hautement disponible à exploiter

Risque de goulot d'étranglement

Une mise à jour des APIs backend nécessite une
mise à jour des APIs de la gateway



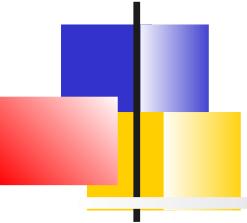
Design issues

Performance et scalabilité : Modèle réactif plus approprié

Composition API : Effectuer si possible les requêtes en parallèle. Encore une fois modèle réactif

Traiter les dysfonctionnements : Répliquer la gateway, utiliser le circuit breaker pattern

Doit respecter les mêmes principes que les services backend : Se baser sur un service de discovery et offrir des points d'observabilité



Tests

Introduction
Tests unitaires
Tests d'intégration
Tests Composant
Test End To End

Pyramide des tests

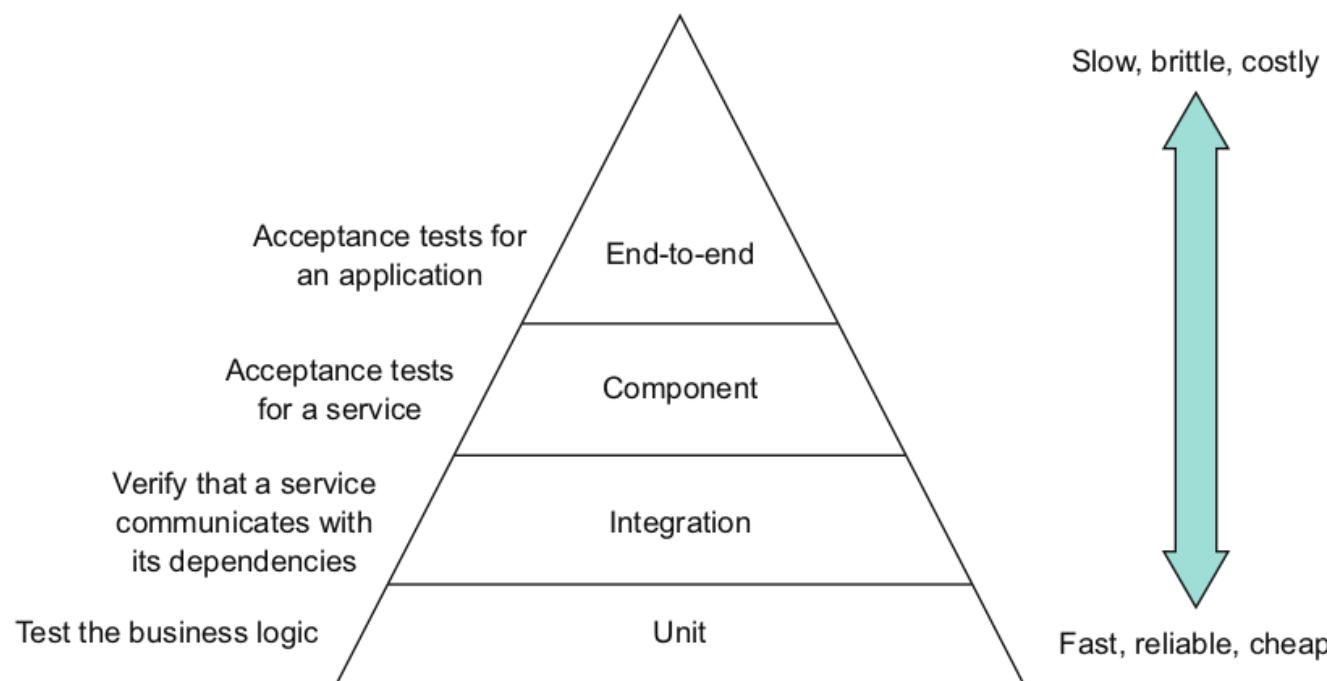
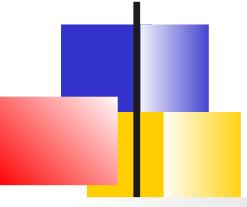


Figure 9.5 The test pyramid describes the relative proportions of each type of test that you need to write. As you move up the pyramid, you should write fewer and fewer tests.

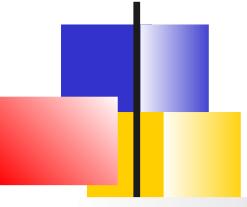


Tests micro-services

Les micro-services ont beaucoup d'interactions entre eux

Les tests end-2-end sont longs et coûteux

Besoin de tester les services en isolation
=> Consumer-driven contract testing



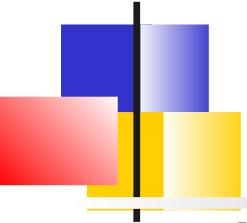
Consumer Driven Contract Test

Consumer-driven contract test Pattern¹ : Vérifier que la « forme » de l'API d'un fournisseur répond aux attentes du consommateur.

Dans le cas REST, le test de contrat vérifie que le fournisseur implémente un point de terminaison qui :

- A la méthode et le chemin HTTP attendus
- Accepte les entêtes attendus
- Accepte un corps de requête, le cas échéant
- Renvoie une réponse avec le code d'état, les entêtes et le corps attendus

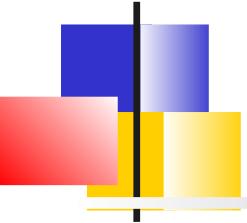
1. <http://microservices.io/patterns/testing/service-integration-contract-test.html>



Spécification par l'exemple

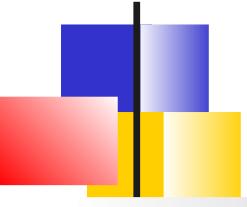
Le Consumer-driven contract définit les interactions entre un fournisseur et consommateur via des exemples, i.e les contrats

Chaque contrat consiste en des exemples des messages échangés durant une interaction



Tests

Introduction
Tests unitaires
Tests d'intégration
Tests Composant
Test End To End



Tests unitaires

Il existe deux types de tests unitaires¹ :

- Test unitaire solitaire : teste une classe en isolation avec des mock-objects
- Test unitaire sociable : teste une classe avec ses dépendances

Les responsabilités de la classe et son rôle dans l'architecture du service déterminent quel type de test à utiliser.

1. <https://martinfowler.com/bliki/UnitTest.html>

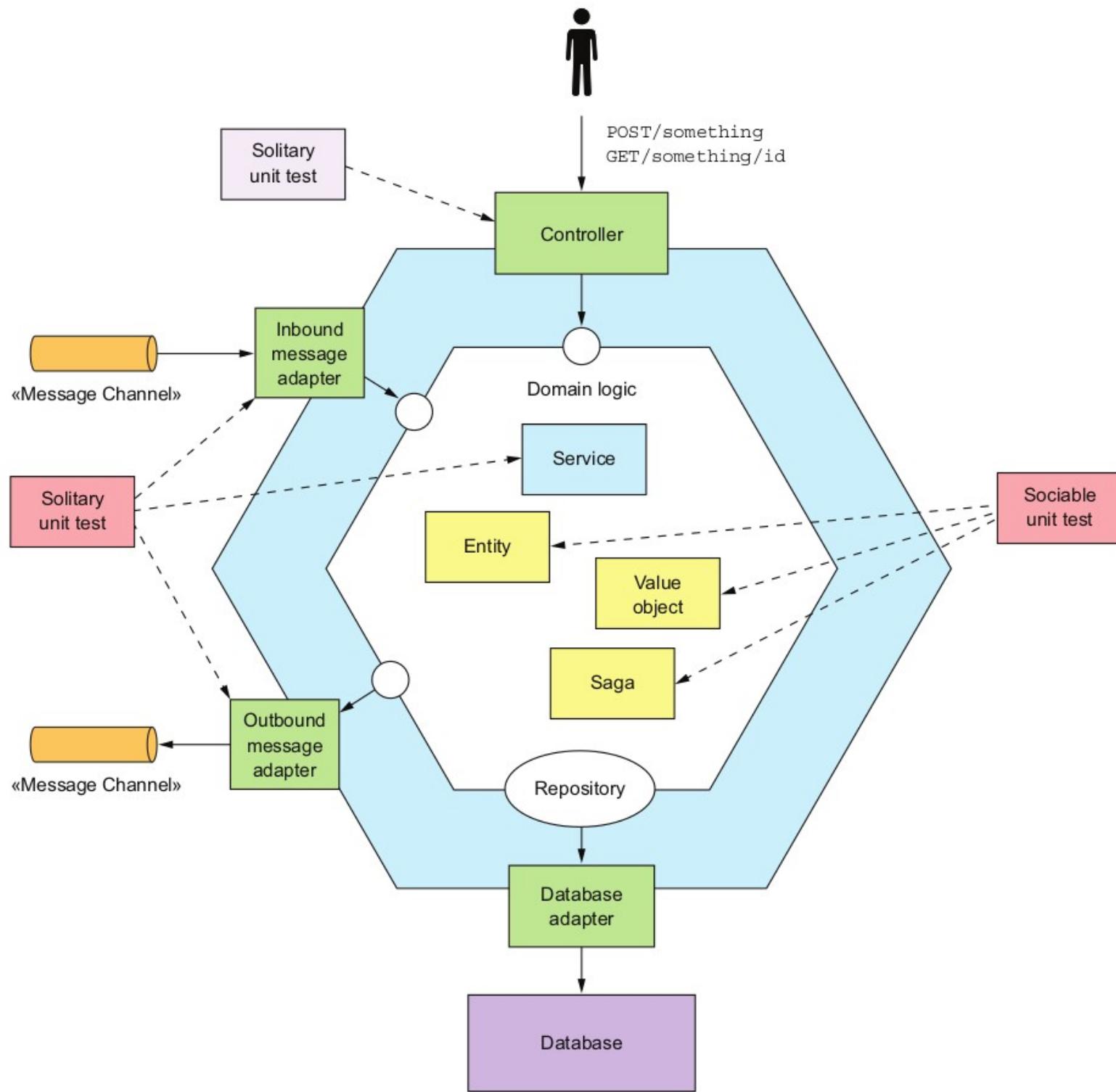
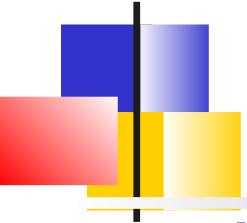


Figure 9.11 The responsibilities of a class determine whether to use a solitary or sociable unit test.

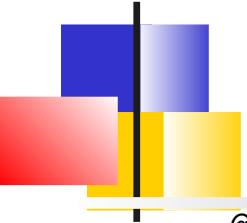


Tests des sagas

Un test une classe saga vérifie la séquence de messages attendue est bien envoyés aux participants de la saga.

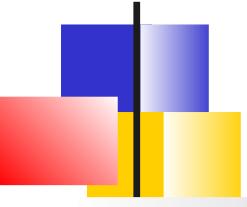
Les tests doit couvrir les différents scénarios.

- Le scénario ou la SAGA est validée
- Les scénarios de rollback.



Exemple Saga

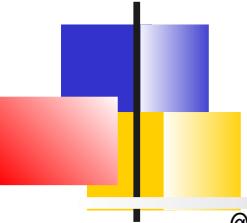
```
@Test
public void shouldCreateOrder() {
given()
    .saga(new CreateOrderSaga(kitchenServiceProxy),
          new CreateOrderSagaState(ORDER_ID, CHICKEN_VINDALOO_ORDER_DETAILS))
    .expect() // Vérification de l'envoi vers ConsumerService
        .command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                                              CHICKEN_VINDALOO_ORDER_TOTAL))
        .to(ConsumerServiceChannels.consumerServiceChannel)
    .andGiven()
        .successReply() // Consumer Service a bien répondu
    .expect() // Vérification du bon message à KitchenService
        .command(new CreateTicket(AJANTA_ID, ORDER_ID, null))
        .to(KitchenServiceChannels.kitchenServiceChannel);
}
```



Exemple Service (1)

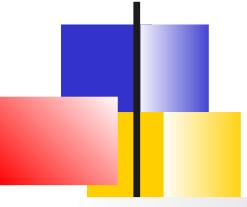
```
// Créer des mocks pour les dépendances et initialiser le service

@Before
public void setup() {
    orderRepository = mock(OrderRepository.class);
    eventPublisher = mock(DomainEventPublisher.class);
    restaurantRepository = mock(RestaurantRepository.class);
    createOrderSagaManager = mock(SagaManager.class);
    cancelOrderSagaManager = mock(SagaManager.class);
    reviseOrderSagaManager = mock(SagaManager.class);
    orderService = new OrderService(orderRepository, eventPublisher,
                                    restaurantRepository, createOrderSagaManager,
                                    cancelOrderSagaManager, reviseOrderSagaManager);
}
```



Exemple Service (2)

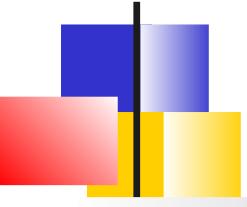
```
@Test
public void shouldCreateOrder() {
    // Configuration des mocks
    when(restaurantRepository.findById(AJANTA_ID)).thenReturn(Optional.of(AJANTA_RESTAURANT_));
    when(orderRepository.save(any(Order.class))).then(invocation -> {
        Order order = (Order) invocation.getArguments()[0];
        order.setId(ORDER_ID); return order;});
    // Appel de la méthode
    Order order = orderService.createOrder(CONSUMER_ID, AJANTA_ID,
        CHICKEN_VINDALOO_MENU_ITEMS_AND_QUANTITIES);
    // Vérification des appels vers les dépendances
    verify(orderRepository).save(same(order));
    verify(eventPublisher).publish(Order.class, ORDER_ID, singletonList(new
        OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)));
    verify(createOrderSagaManager).create(new
        CreateOrderSagaState(ORDER_ID, CHICKEN_VINDALOO_ORDER_DETAILS),
        Order.class, ORDER_ID);
}
```



Exemple Contrôleur

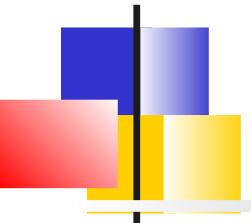
```
@Before
public void setUp() throws Exception {
    // Mocker les dépendances
    orderService = mock(OrderService.class);
    orderRepository = mock(OrderRepository.class);
    orderController = new OrderController(orderService, orderRepository);
}

@Test
public void shouldFindOrder() {
    // Configurer le mock
    when(orderRepository.findById(1L))
        .thenReturn(Optional.of(CHICKEN_VINDALOO_ORDER_));
    // Le Test
    given().
        standaloneSetup(configureControllers(new OrderController(orderService, orderRepository))).when().
        get("/orders/1").
    then(). // Vérification du statut et du corps de la réponse
        statusCode(200).
        body("orderId", equalTo(new Long(OrderDetailsMother.ORDER_ID).intValue()).
        body("state", equalTo(OrderDetailsMother.CHICKEN_VINDALOO_ORDER_STATE.name()).
        body("orderTotal", equalTo(CHICKEN_VINDALOO_ORDER_TOTALasString())))
```



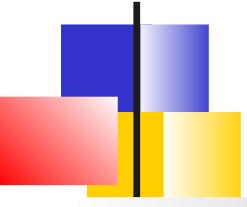
Exemple gestionnaire de message

```
@Before
public void setUp() throws Exception {
    // Mock des dépendances
    orderService = mock(OrderService.class);
    orderEventConsumer = new OrderEventConsumer(orderService);
}
@Test
public void shouldCreateMenu() {
given().
    eventHandlers(orderEventConsumer.domainEventHandlers()).
when(). // Publication d'un événement
    aggregate("ftgo.restaurantservice.domain.Restaurant", AJANTA_ID)
    .publishes(new RestaurantCreated(AJANTA_RESTAURANT_NAME, AJANTA_RESTAURANT_MENU))
then(). // Vérification que le handler a appelé la bonne méthode
verify(() -> {
    verify(orderService).createMenu(AJANTA_ID,
        new RestaurantMenu(RestaurantMother.AJANTA_RESTAURANT_MENU_ITEMS));
}) ;
}
```



Tests

Introduction
Tests unitaires
Tests d'intégration
Tests Composant
Test End To End

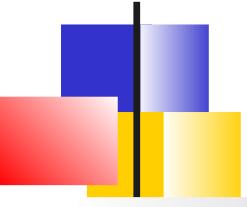


Tests d'intégration

Les tests d'intégration doivent vérifier qu'un service peut communiquer avec ses clients et ses dépendances (BD, Message Broker).

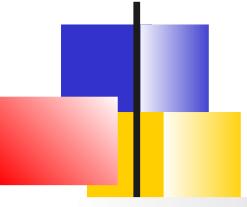
Mais plutôt que de tester des services entiers, la stratégie consiste à tester les classes d'adaptateur implémentant la communication individuellement

- Tester les classes Repository
- Tester si les événements sont correctement publiés
- Tester si les interactions entre services sont correctes



Exemple Repository

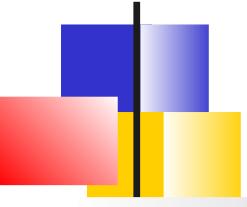
```
// La base doit être provisionnée,  
// par exemple via Docker compose Gradle Plugin  
  
@Test  
public void shouldSaveAndLoadOrder() {  
    Long orderId = transactionTemplate.execute((ts) -> {  
        Order order = new Order(CONSUMER_ID, AJANTA_ID,  
CHICKEN_VINDALOO_LINE_ITEMS);  
        orderRepository.save(order);  
        return order.getId();  
    });  
    transactionTemplate.execute((ts) -> {  
        Order order = orderRepository.findById(orderId).get();  
        assertEquals(OrderState.APPROVAL_PENDING, order.getState());  
        assertEquals(AJANTA_ID, order.getRestaurantId());  
        assertEquals(CONSUMER_ID, order.getConsumerId().longValue());  
        assertEquals(CHICKEN_VINDALOO_LINE_ITEMS, order.getLineItems());  
        return null;  
    });  
}
```



Différents contrats

La structure d'un contrat dépend du type d'interaction entre les services

- REST : Des exemples de requêtes HTTP et les réponses attendues
- Publish/subscribe : Les événements du domaine
- Requêtes /réponses asynchrones : Message de commande et de réponse



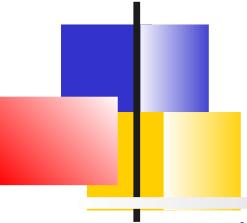
Spring Cloud Contract

Spring Cloud Contract est un projet qui permet d'adopter une approche *Consumer Driven Contract*

A partir d'une spécification d'interaction entre un producteur/serveur et consommateur/client, cela permet

- De générer des tests d'acceptation côté producteur
- De créer des mocks serveur pour le client

SCC permet également les tests pour les APIs de messaging



Exemple Groovy

```
import org.springframework.cloud.contract.spec.Contract
Contract.make {
    description "should return even when number input is even"
    request{
        method GET()
        url("/validate/prime-number") {
            queryParameters {
                parameter("number", "2")
            }
        }
    }
    response {
        body("Even")
        status 200
    }
}
```

Process

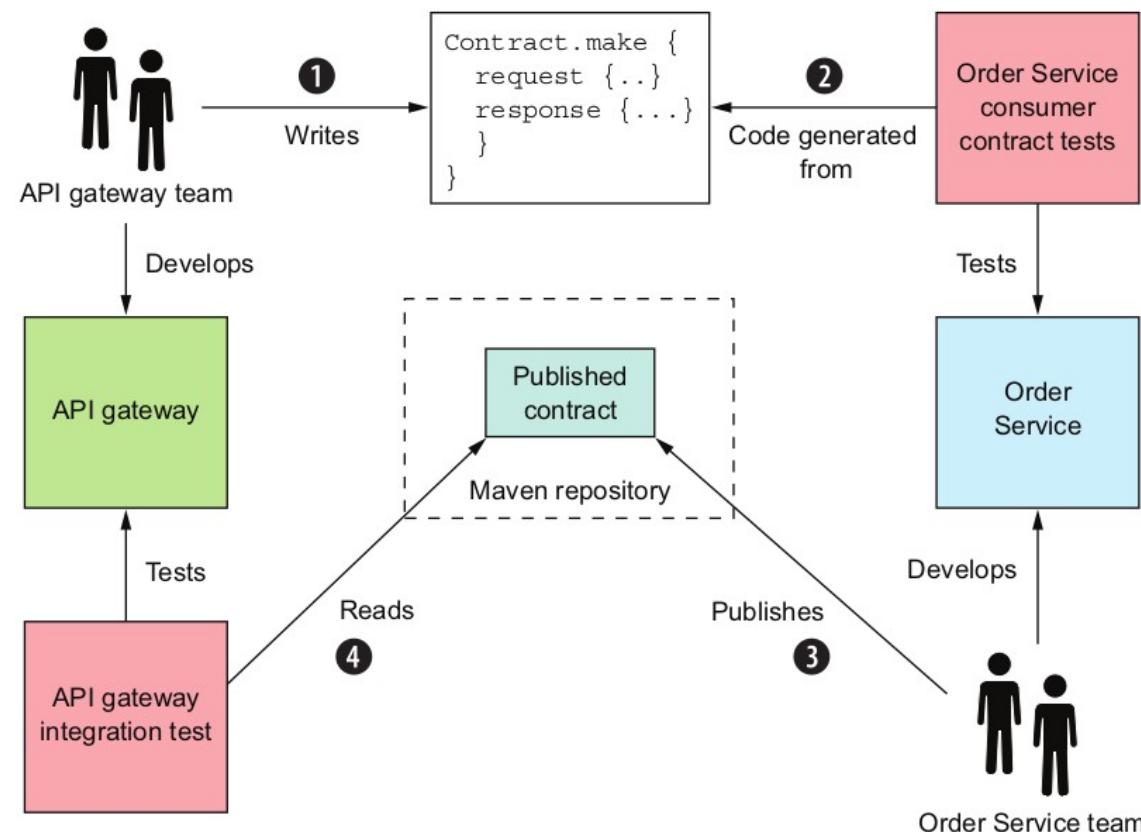
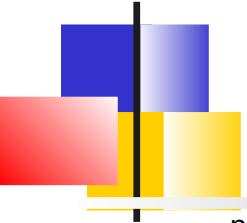
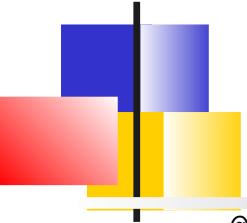


Figure 9.8 The API Gateway team writes the contracts. The Order Service team uses those contracts to test Order Service and publishes them to a repository. The API Gateway team uses the published contracts to test API Gateway.



Tests générés côté producteur

```
public class ContractVerifierTest extends BaseTestClass {  
  
    @Test  
    public void validate_shouldReturnEvenWhenRequestParamIsEven() throws Exception {  
        // given:  
        MockMvcRequestSpecification request = given();  
  
        // when:  
        ResponseOptions response = given().spec(request)  
            .queryParam("number", "2")  
            .get("/validate/prime-number");  
  
        // then:  
        assertThat(response.statusCode()).isEqualTo(200);  
  
        // and:  
        String responseBody = response.getBody().asString();  
        assertThat(responseBody).isEqualTo("Even");  
    }  
}
```



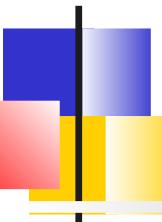
Tests Consommateur

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@AutoConfigureMockMvc
@AutoConfigureJsonTesters
@AutoConfigureStubRunner(
    stubsMode = StubRunnerProperties.StubsMode.LOCAL,
    ids = "com.baeldung.spring.cloud:spring-cloud-contract-producer:+:stubs:8090")
public class BasicMathControllerIntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven()
        throws Exception {

        mockMvc.perform(MockMvcRequestBuilders.get("/calculate?number=2")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string("Even"));
    }
}
```



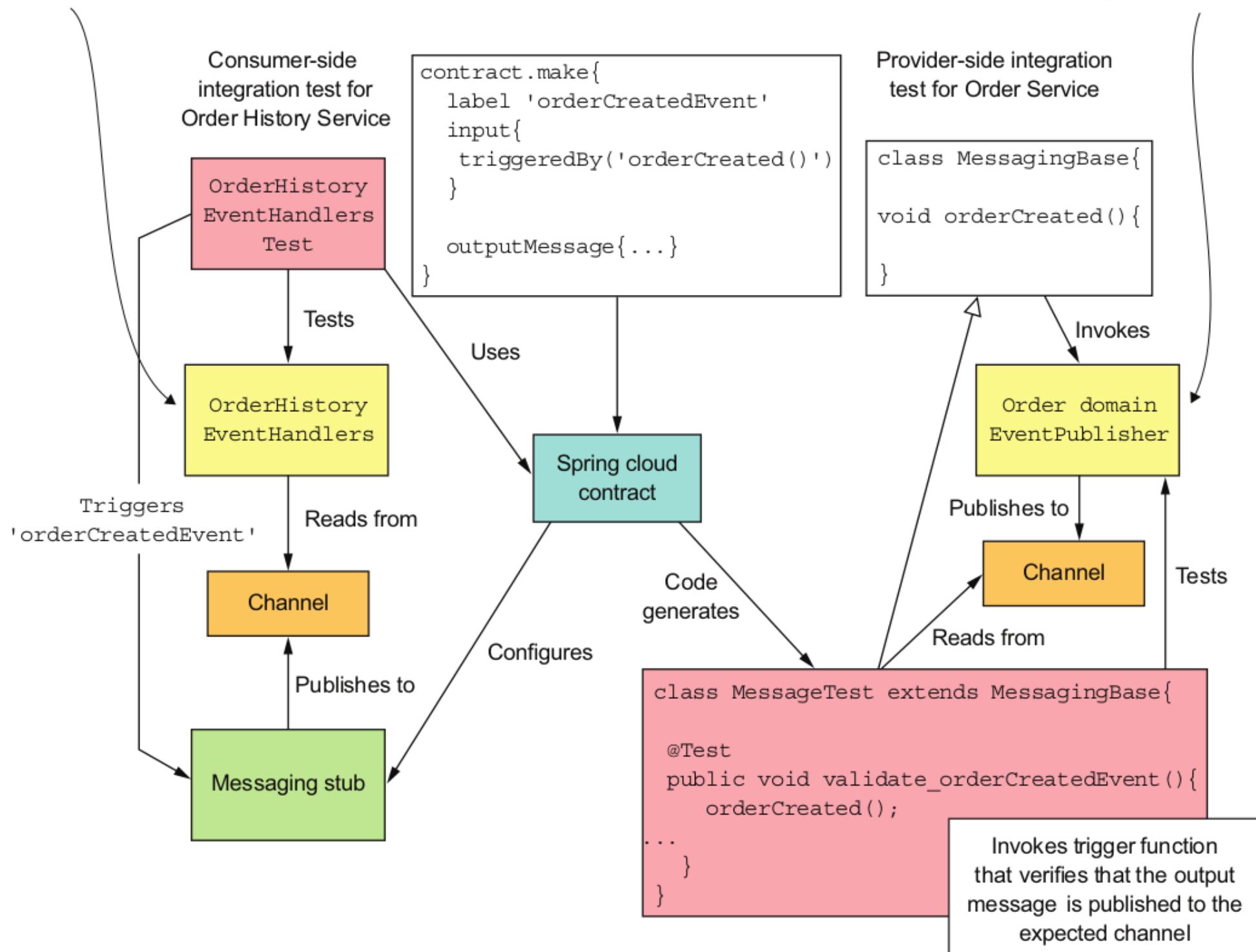
Interactions Publish/Subscribe

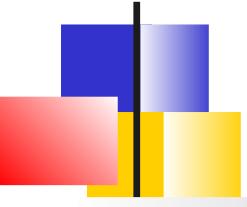
Les tests d'intégration doivent alors vérifier que le producteur et ses consommateurs sont d'accord sur le canal de message et la structure des événements du domaine.

L'approche est similaire aux tests d'interactions REST, un contrat spécifie des exemples d'événements du domaine

Class under test

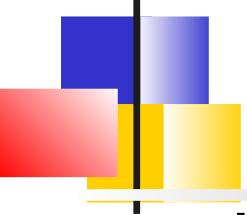
Class under test





Contrat

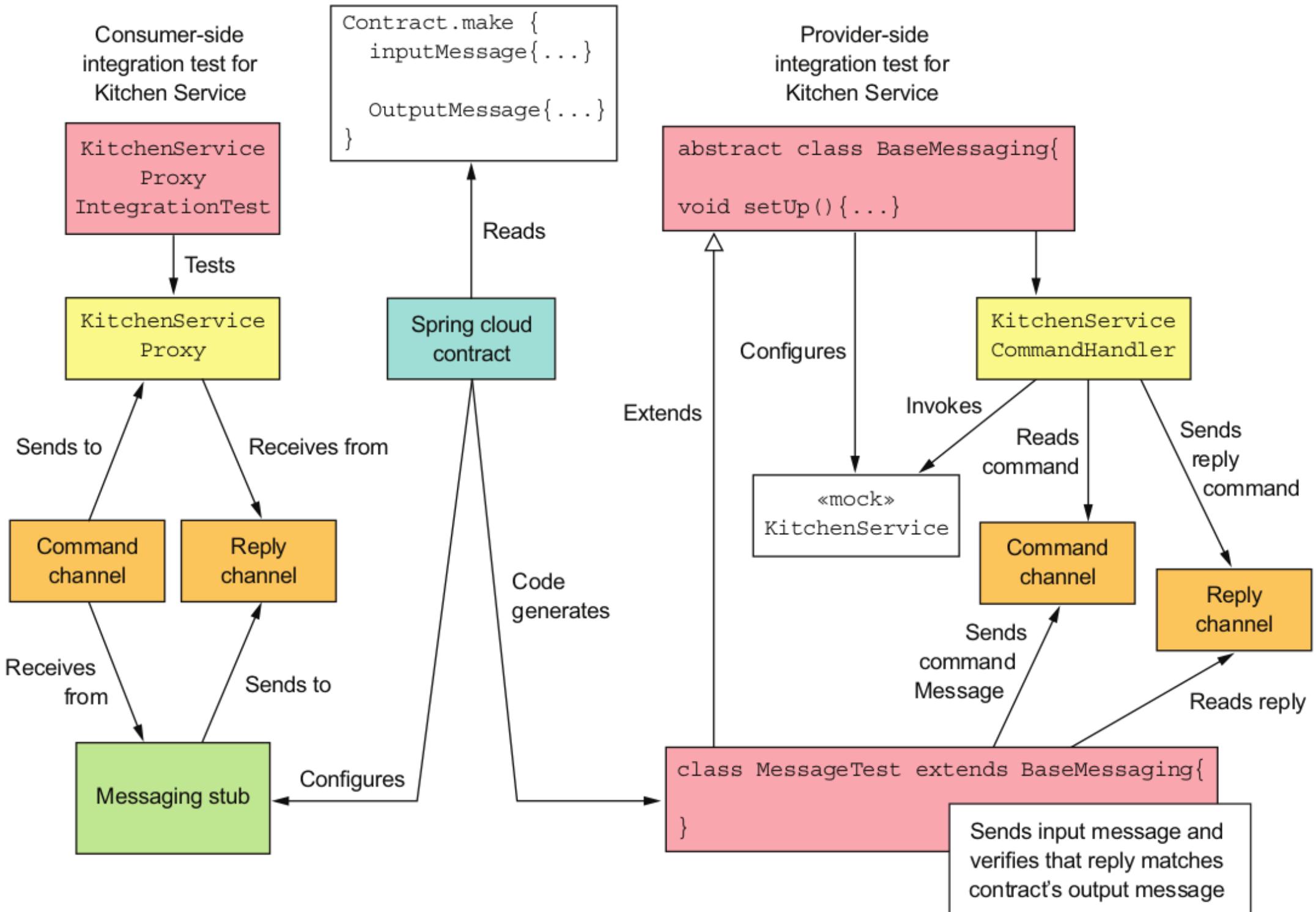
```
org.springframework.cloud.contract.spec.Contract.make {  
    label 'orderCreatedEvent'  
    input {  
        triggeredBy('orderCreated())')  
    }  
    outputMessage {  
        sentTo('net.chrisrichardson.ftgo.orderservice.domain.Order')  
        body('''{ "orderDetails":{ "lineItems": [ { "quantity":5, "menuItemId": "1",  
            "name": "Chicken  
Vindaloo", "price": "12.34", "total": "61.70" } ], "orderTotal": "61.70", "restaurantId": 1,  
            "consumerId": 1511300065921 }, "orderState": "APPROVAL_PENDING" } ''')  
        headers {  
            header('event-aggregate-type',  
                ' net.chrisrichardson.ftgo.orderservice.domain.Order')  
            header('event-aggregate-id', '1')  
        }  
    }  
}
```

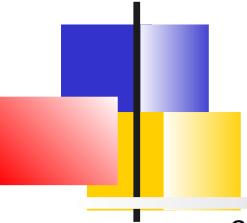


Requête/Réponse asynchrones

Les deux parties dans une interaction demande/réponse asynchrone sont le demandeur, qui est le service qui envoie la commande, et le répondeur, qui est le service qui traite la commande et renvoie une réponse.

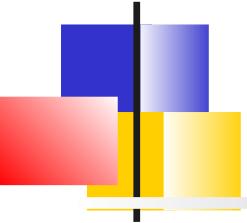
Ils doivent se mettre d'accord sur le nom du canal de message de commande et la structure des messages de commande et de réponse.





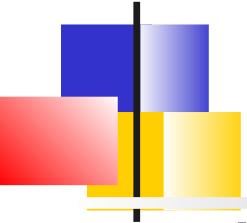
Contrat

```
org.springframework.cloud.contract.spec.Contract.make {  
    label 'createTicket'  
    input {  
        messageFrom('kitchenService')  
        messageBody('''{ "orderId":1,"restaurantId":1,"ticketDetails":{...}}'''')  
        messageHeaders {  
            header('command_type','net.chrisrichardson...CreateTicket')  
            header('command_saga_type','net.chrisrichardson...CreateOrderSaga')  
            header('command_saga_id',$(consumer(regex('[0-9a-f]{16}-[0-9a-f]{16}'))))  
            header('command_reply_to','net.chrisrichardson...CreateOrderSaga-Reply')  
        }  
    }  
    outputMessage {  
        sentTo('net.chrisrichardson...CreateOrderSaga-reply')  
        body([  
            ticketId: 1  
        ])  
        headers {  
            header('reply_type', 'net.chrisrichardson...CreateTicketReply')  
            header('reply_outcome-type', 'SUCCESS')  
        }  
    }  
}
```



Tests

Introduction
Tests unitaires
Tests d'intégration
Tests Composant
Test End To End

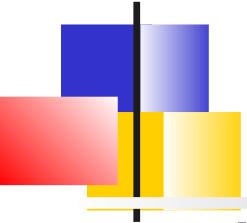


Tests de composant

Les tests de composants sont les tests d'acceptation d'un service.

Ce sont des tests black box qui vérifient le bon comportement d'un service à travers son API

Le test du service est cependant en isolation, les services dépendants sont mockés

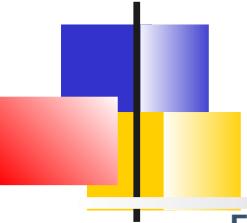


Tests d'acceptation

Les tests d'acceptation sont des tests métier.

Ils décrivent le comportement visible de l'extérieur souhaité du point de vue des clients du composant plutôt qu'en termes de mise en œuvre interne

Ils peuvent être exprimés via la syntaxe Gherkin



Exemple Gherkin

Feature: Place Order

As a consumer of the Order Service

I should be able to place an order

Scenario: Order authorized

Given a valid consumer

Given using a valid credit card

Given the restaurant is accepting orders

When I place an order for Chicken Vindaloo at Ajanta

Then the order should be APPROVED

And an OrderAuthorized event should be published

Scenario: Order rejected due to expired credit card

Given a valid consumer

Given using an expired credit card

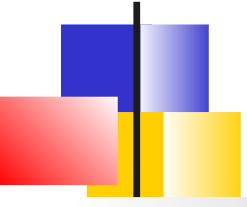
Given the restaurant is accepting orders

When I place an order for Chicken Vindaloo at Ajanta

Then the order should be REJECTED

And an OrderRejected event should be published

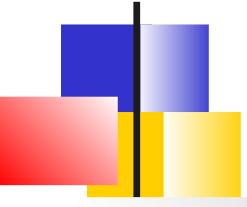
...



Cucumber

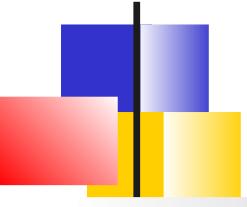
Cucumber est un framework de test automatisé qui exécute des tests écrits en Gherkin. Disponible dans une variété de langages, dont Java.

- Avec Java, les développeurs écrivent des classes de définitions d'étape,
- Les classes sont composées de méthodes qui définissent la signification de chaque étape du test.
- Les méthodes sont annotées avec **@Given** , **@When** , **@Then** ou **@And** plus un attribut qui est généralement une expression régulière, que Cucumber match avec la formulation *Gherkin*



Exemple

```
public class StepDefinitions ...  
{  
    ...  
    @Given("A valid consumer")  
    public void useConsumer() { ... }  
  
    @Given("using a(.*?) (.*) credit card")  
    public void useCreditCard(String ignore, String creditCard) { ... }  
  
    @When("I place an order for Chicken Vindaloo at Ajanta")  
    public void placeOrder() { ... }  
  
    @Then("the order should be (.*)")  
    public void theOrderShouldBe(String desiredOrderState) { ... }  
  
    @And("an (.*) event should be published")  
    public void verifyEventPublished(String expectedEventClass)  
    { ... }  
}
```



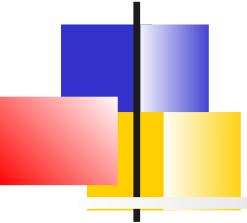
Exécution des tests de composants

Pour exécuter les tests de composants, il faut disposer des infrastructures (BD, Message Broker) et de mock des services

Pour les infrastructures, les alternatives sont :

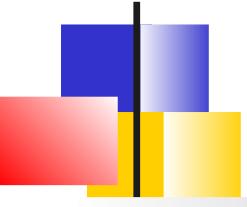
- Utiliser des BD/Message Broker in-memory
Attention on n'est plus dans les conditions de production
- Utiliser des images docker pour démarrer les services nécessaires
Plus lourd à mettre en place

Pour mocker les services dépendants, on peut utiliser des solutions comme *WireMock*



Tests

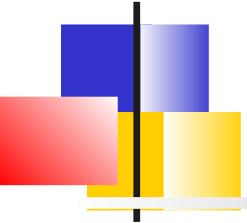
Introduction
Tests unitaires
Tests d'intégration
Tests Composant
Test End To End



Tests end-2-end

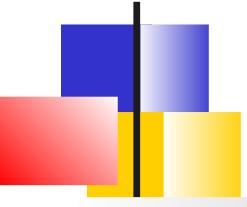
Les tests de bout en bout doivent exécuter l'intégralité de l'application, y compris tous les services d'infrastructure requis
=> *docker-compose* est alors intéressant

Ce sont des tests black-box et peuvent également utiliser la syntaxe *Gherkin* et *Cucumber*



Vers la production

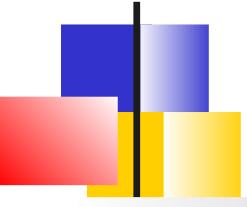
Préparation pour la production
Infrastructure de déploiement
Kubernetes et Istio



Introduction

Avant d'être prêt à être déployer en production, un service doit s'assurer de critères critiques :

- La sécurité
- La configurabilité
- L'observabilité



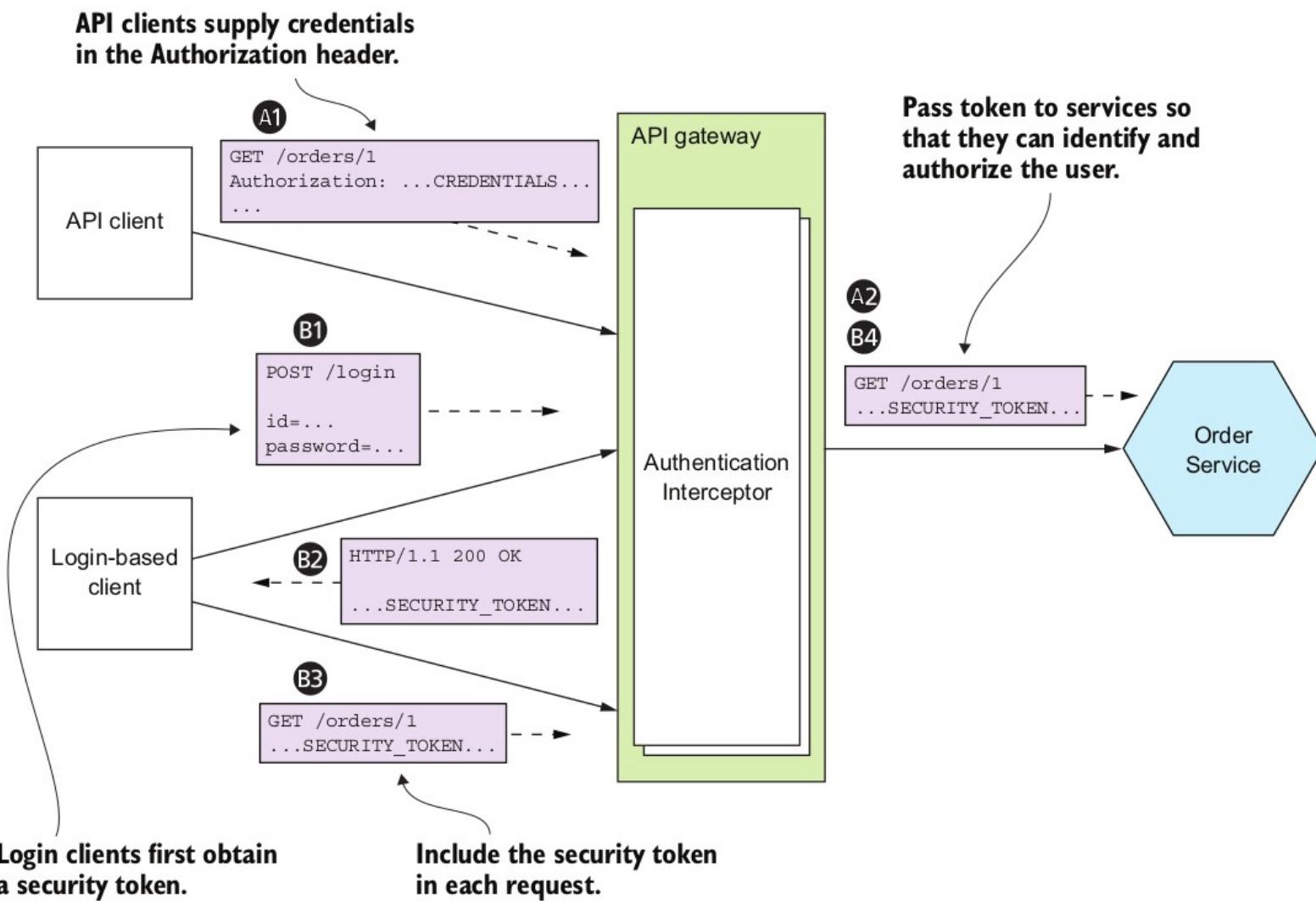
Sécurité

Plusieurs approches pour sécuriser une architecture micro-services :

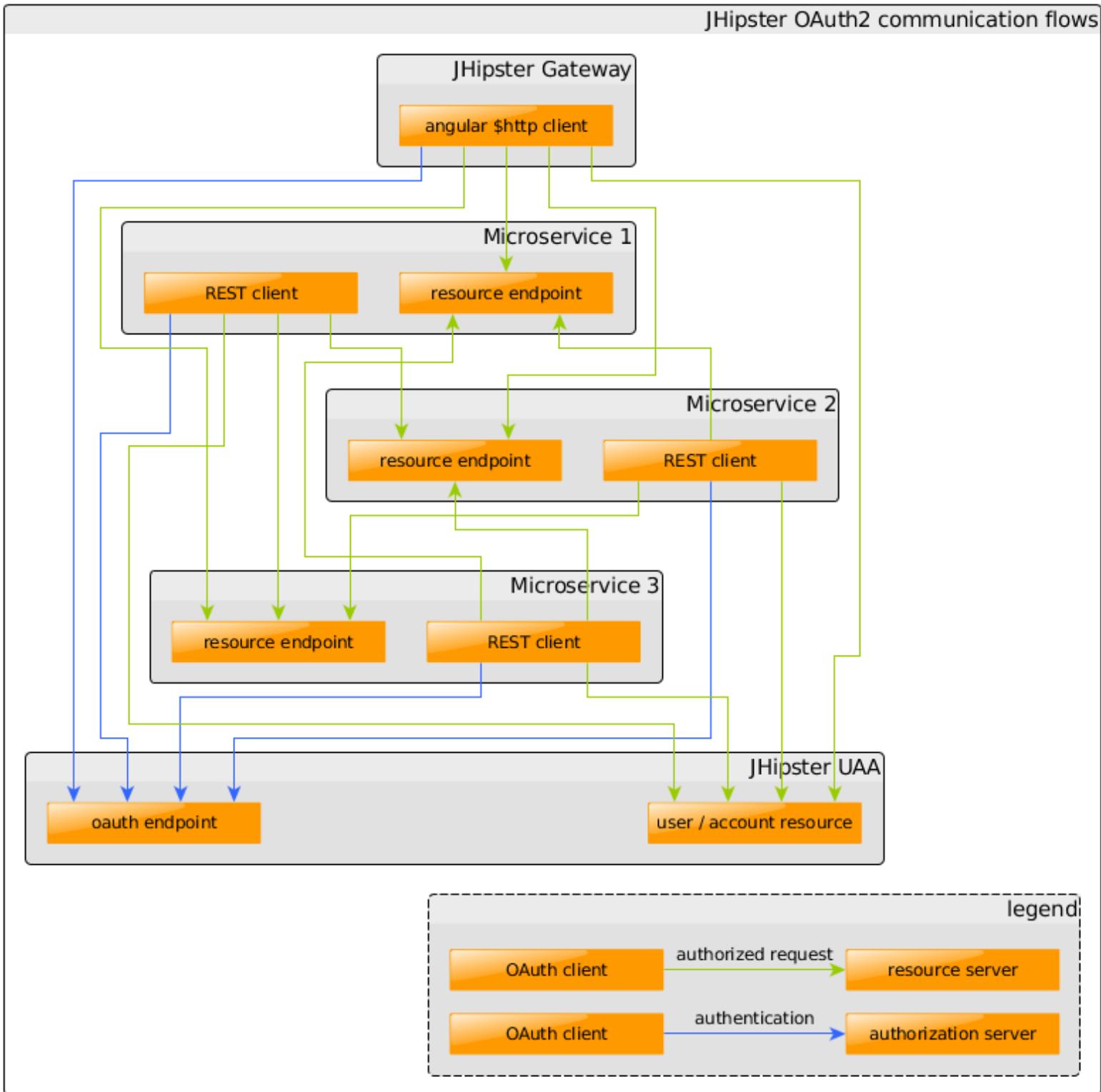
- N'implémenter la sécurité qu'au niveau de la gateway proxy. Les micro-services back-end ne sont protégés que par le firewall
- **Access token Pattern¹** : La gateway passe un jeton contenant l'information sur le user (identité et rôles)
Chaque micro-service a alors sa propre politique de sécurité.
- Chaque micro-service a sa propre politique de sécurité et chaque micro-service demande son propre jeton pour effectuer ses appels REST vers ses dépendances

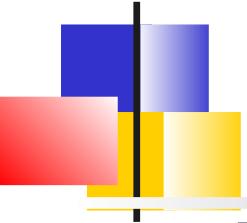
1. <http://microservices.io/patterns/security/access-token.html>

Access Token Pattern



JHipster OAuth2 communication flows



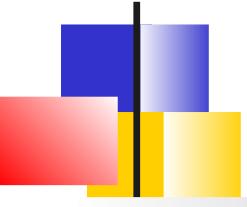


Technologies

Protocole **oAuth2** avec souvent des
Grant Type = Password

Format **JWT** permettant de valider un
jeton sans requête supplémentaire
vers le serveur d'autorisation d'origine

Transport du jeton via l'entête HTTP
Authorization-Header



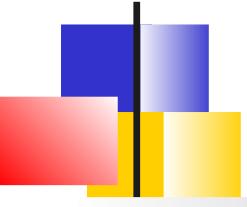
Configuration

Externalized configuration Pattern¹ : Fournir les propriétés de configuration comme les crédentiels BD et les emplacement réseau au service au moment de l'exécution

2 approches :

- Push : L'infrastructure de déploiement passe les propriétés de configuration à l'aide de variables d'environnement ou de fichier de configuration
- Pull : Le service lit ses propriétés de configuration à partir d'un serveur de configuration

1. <http://microservices.io/patterns/externalized-configuration.html>



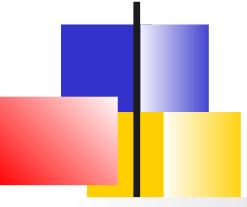
Services Observable

Health check API Pattern¹ : Un service expose un endpoint tel que GET /health, qui retourne la santé du service. Ex : Actuator

Log aggregation Pattern² : Agréger les traces de tous les services dans une base centralisée qui supporte la recherche et la notification d'alerte.
Ex : ElasticStack

Distributed tracing Pattern³ : Affecter à chaque requête externe un ID et le conserver à travers son parcourt dans tous les services du système. Agréger les traces dans un serveur centralisé qui fournit de la visualisation et l'analyse.
Ex : Sleuth, Zipkin

1. <http://microservices.io/patterns/observability/health-check-api.html>
2. <http://microservices.io/patterns/observability/application-logging.html>
3. <http://microservices.io/patterns/observability/distributed-tracing.html>



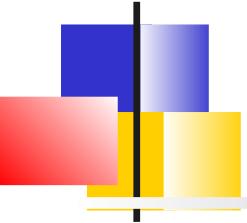
Services Observable (2)

Application metrics Pattern : Les services envoient des métriques vers un serveur central qui fournit de l'agrégation, de la visualisation et des alertes

Exception tracking Pattern¹ : Les services rapportent les exceptions vers un service central qui dédoublonne les exceptions, génère des alertes et gère la résolution d'exception

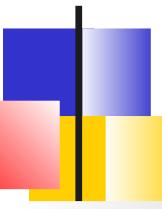
Audit logging Pattern² : Enregistre les actions utilisateur dans une base afin d'aider le support client, d'assurer la conformité et détecter des comportements suspects

1. <http://microservices.io/patterns/observability/audit-logging.html>
2. <http://microservices.io/patterns/observability/audit-logging.html>



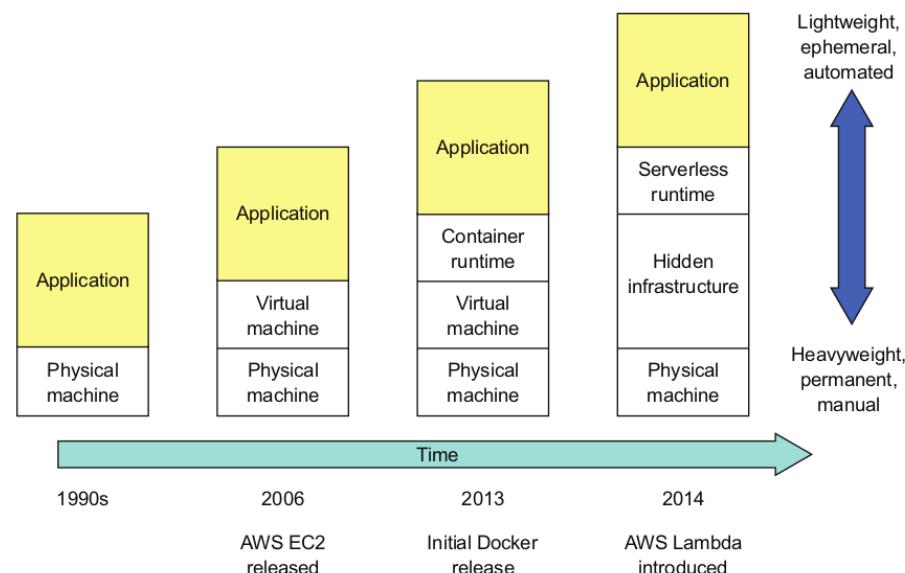
Vers la production

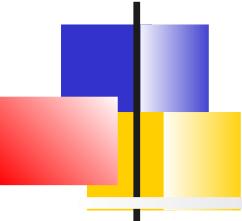
Préparation pour la production
Infrastructure de déploiement
Kubernetes et Istio



Infrastructure de déploiement

Même si plusieurs alternatives peuvent être envisagées, l'utilisation des technologies de container et d'orchestrateur de container sont à privilégier.



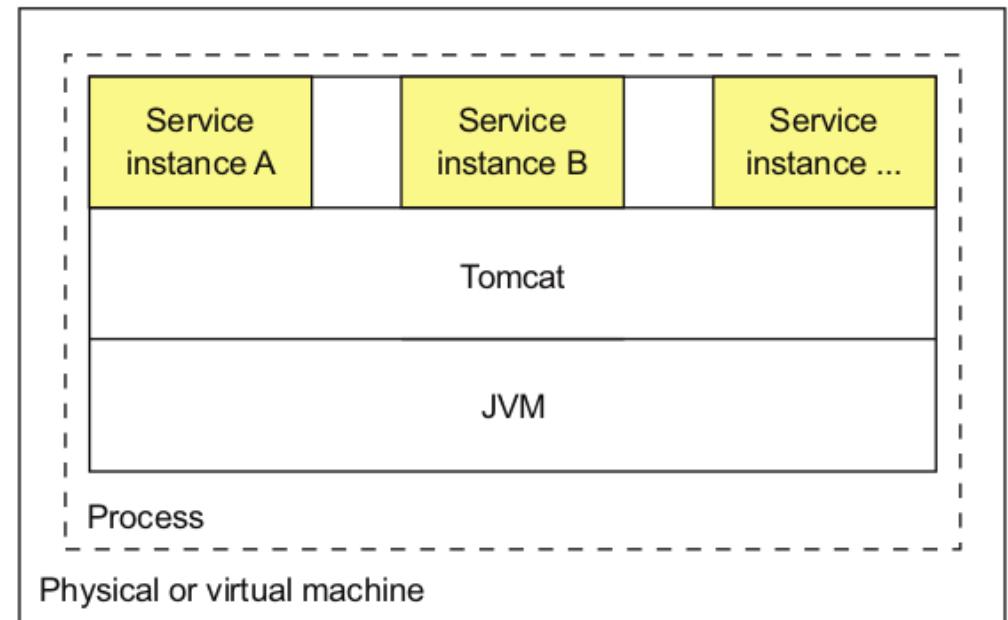
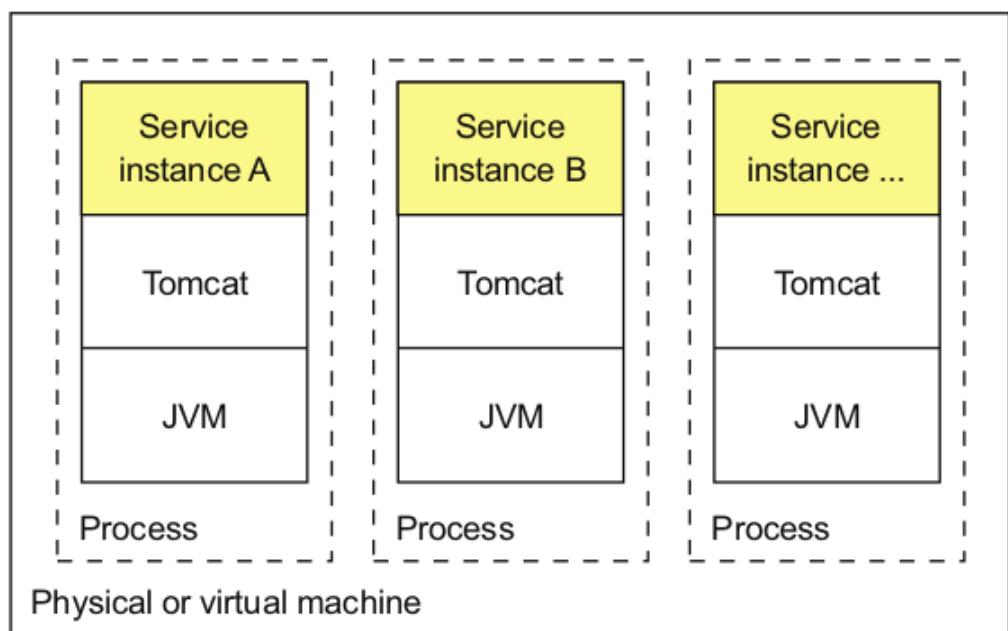


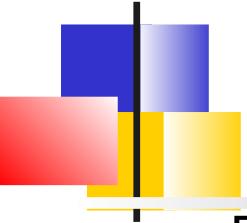
Format de packaging spécifique au langage

Une alternative est de packager les services dans un format spécifique au langage (e : .war) et de le déployer sur une machine provisionnée (JDK + Tomcat)

- Soit chaque service est dans un processus distinct (a son propre Tomcat)
- Soit plusieurs services sont dans le même processus (plusieurs services déployés sur le même Tomcat)

Exemples





Bénéfices / Inconvénients

Bénéfices

Déploiement rapide

Utilisation efficace des ressources surtout lorsque l'on exécute plusieurs instances sur la même machine ou le même processus

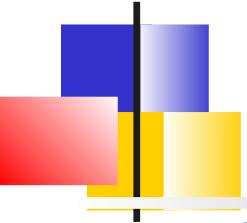
Inconvénients

Pas d'encapsulation de la pile technologique. Déploiements mutables.

Pas de possibilité pour limiter les ressources consommées par une instance de service.

Manque d'isolement lors de l'exécution de plusieurs instances de service sur la même machine.

Il est difficile de déterminer automatiquement où placer les instances de service.



Images VM

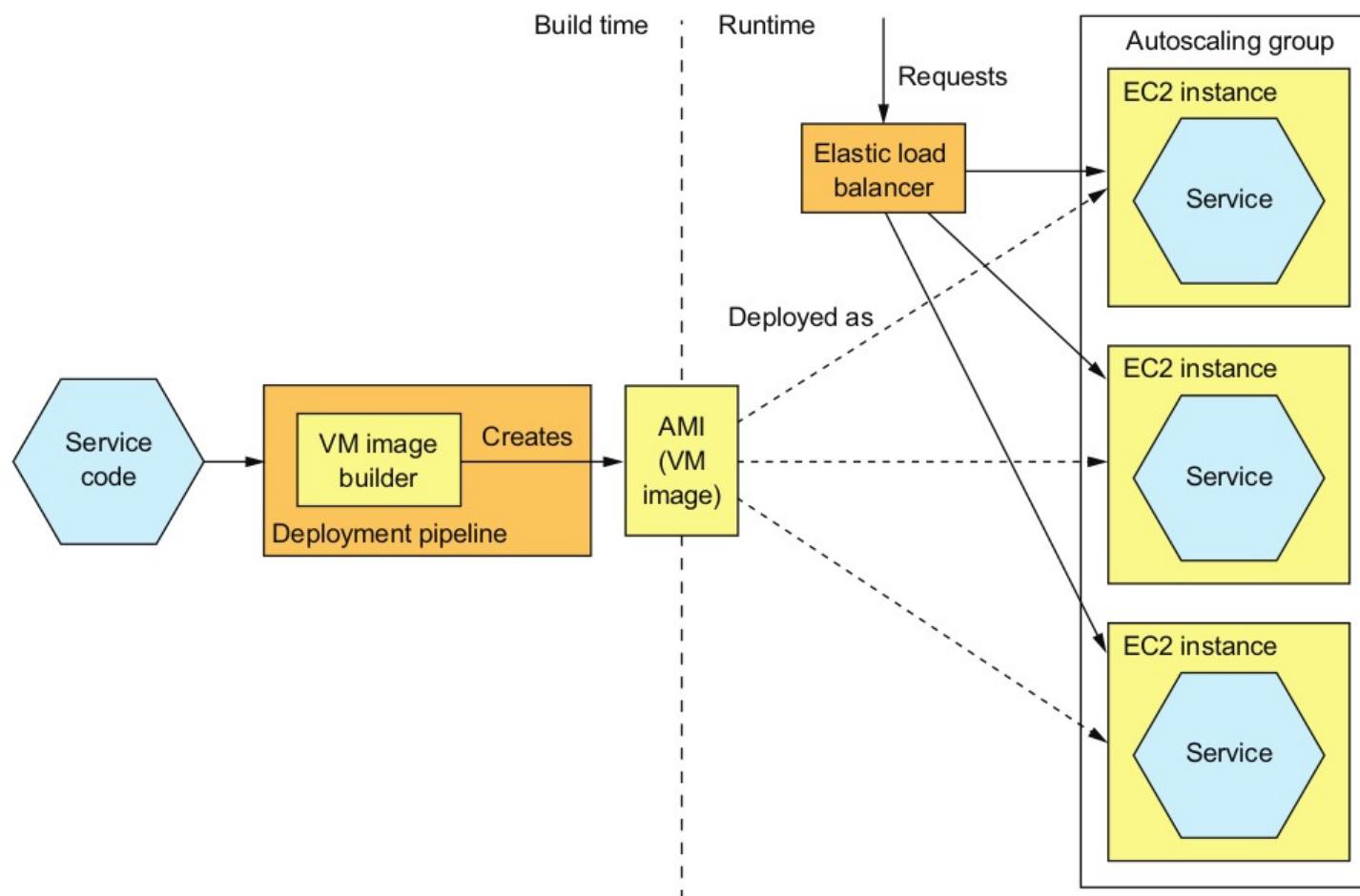
Deploy a service as a VM Pattern¹ :

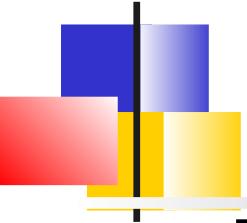
Déployer les services packagés comme des images VM. Chaque service est une VM

Le packaging peut s'automatiser dans les pipelines de build.

1. <http://microservices.io/patterns/deployment/service-per-vm.html>

Exemple





Bénéfices / Inconvénients

Bénéfices :

L'image VM encapsule la pile technologique =>
Déploiement immuable

Instances de service isolées.

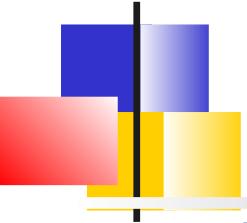
Utilise une infrastructure cloud mature.

Inconvénients :

Utilisation peu efficace des ressources

Déploiements relativement lents

Surcharge d'administration du système



Service comme Container

Deploy a service as a container

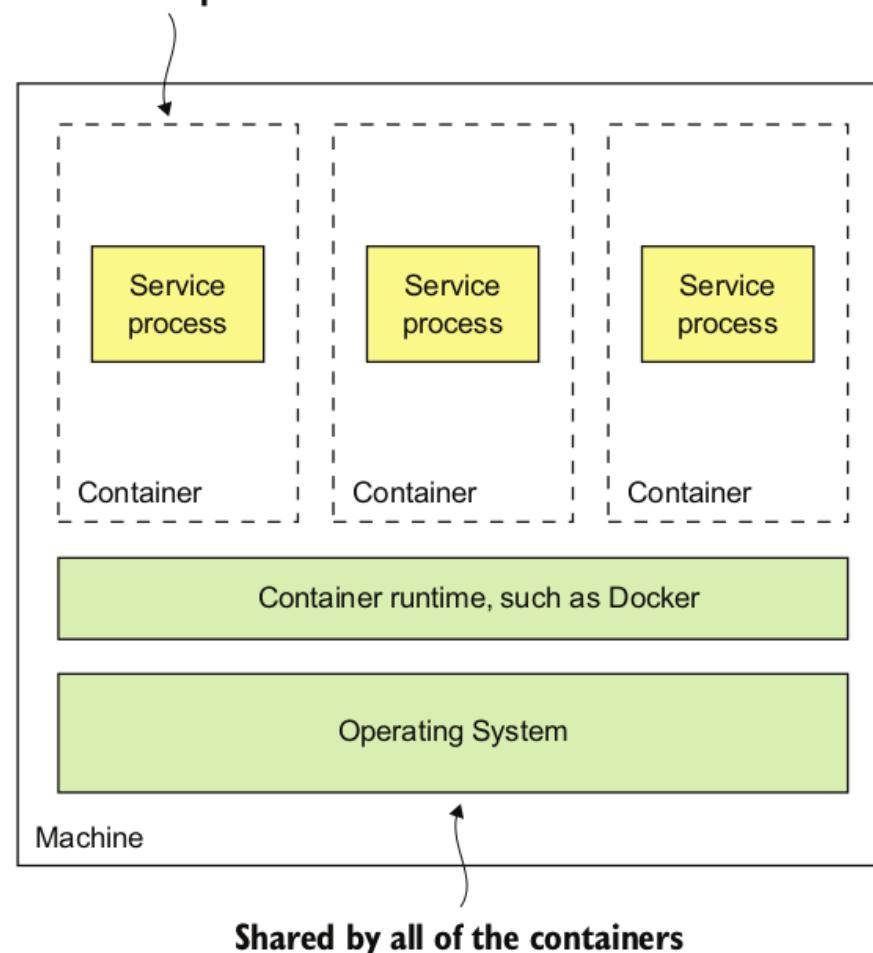
Pattern¹ : Déployé les services packagé comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

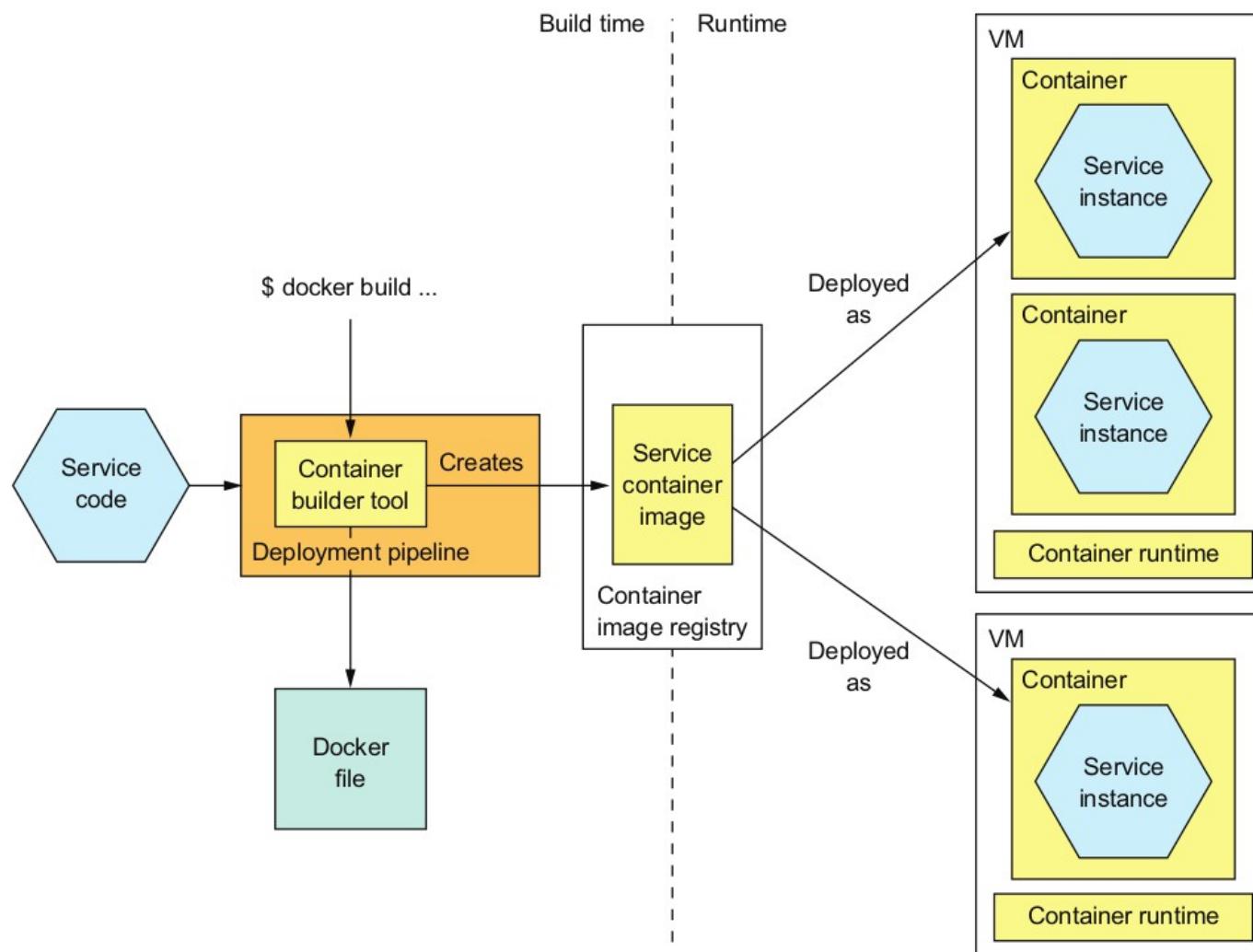
1. <http://microservices.io/patterns/deployment/service-per-container.html>

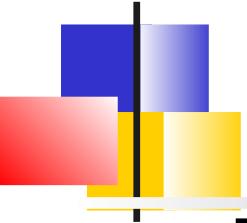
Execution

Each container is a sandbox
that isolates the processes.



Déploiement





Bénéfices / Inconvénients

Bénéfices

Encapsulation de la pile technologique. Déploiements immuables

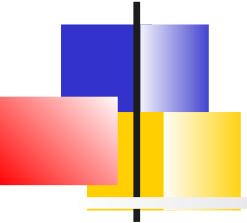
Les instances de service sont isolées.

Les ressources des instances de service sont limitées.

Inconvénients

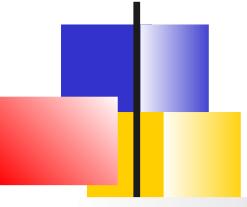
Équipe DevOps responsable de l'administration des images du conteneur. (Patches de l'OS par exemple)

Administrer l'infrastructure du conteneur-runtime et éventuellement des VMs associés



Vers la production

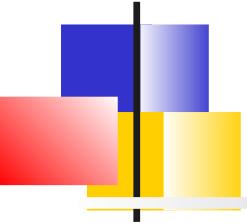
Préparation pour la production
Infrastructure de déploiement
Kubernetes et Istio



Auto-correctif

Kubernetes va TOUJOURS essayer de diriger le cluster vers son état désiré.

- **Moi**: «Je veux que 3 instances de Redis toujours en fonctionnement.»
- **Kubernetes**: «OK, je vais m'assurer qu'il y a toujours 3 instances en cours d'exécution. »
- **Kubernetes**: «Oh regarde, il y en a un qui est mort. Je vais essayer d'en créer un nouveau. »

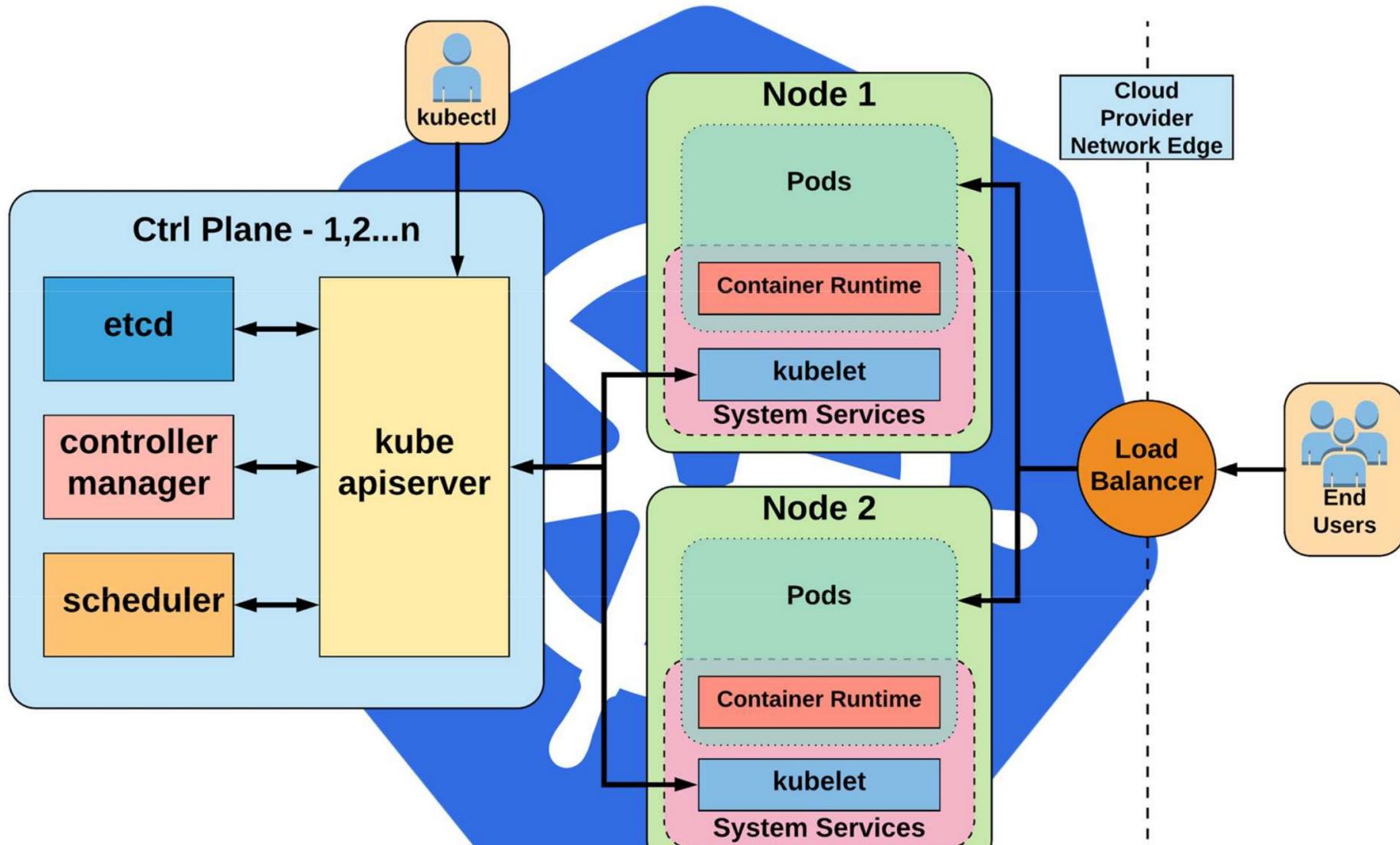


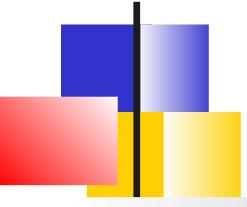
Fonctionnalités applicatives

- Scaling automatique
- Déploiements Blue/Green
- Démarrage de jobs planifiés
- Gestion d'application Stateless et Stateful
- Méthodes natives pour la découverte de services
- Intégration et support d'applications fournies par des tiers (*Helm*)

pod = 1 ou plusieurs conteneurs co-localisés

Architecture cluster





API

L'interaction se fait par une API Rest très riche.

L'API est très cohérente et tous les appels suivent le même format

Format:

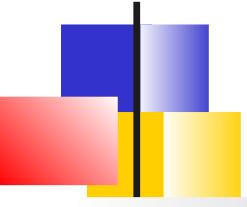
`/apis/<group>/<version>/<resource>`

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`

L'outil **kubectl** et le format **yaml** sont les plus appropriés pour effectuer les requêtes REST

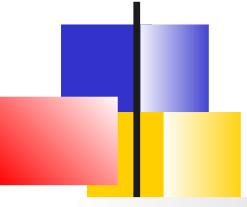


Principes

L'API est une API Rest, elle permet principalement des opérations CRUD sur des **ressources**

En particulier, le client *kubectl* propose les commandes :

- **create** : Créer une ressource
- **get** : Récupérer une ressource
- **edit/set** : Mise à jour d'une ressources
- **delete** : Suppression d'une ressource



Ressources applicatives

Les principales ressources d'une application sont :

- **deployment** : Un déploiement, les déploiements font référence à des *ReplicaSet*, ils peuvent être historisés
- **replicaSet** : Ils définissent le nombre d'instances maximales pour une image de conteneur applicative
- **pod** : Ce sont des conteneurs qui s'exécutent, ils sont distribués sur les nœuds par le scheduler de *Kubernetes*
- **service** : Ce sont des points d'accès stables à un service applicatif

pod

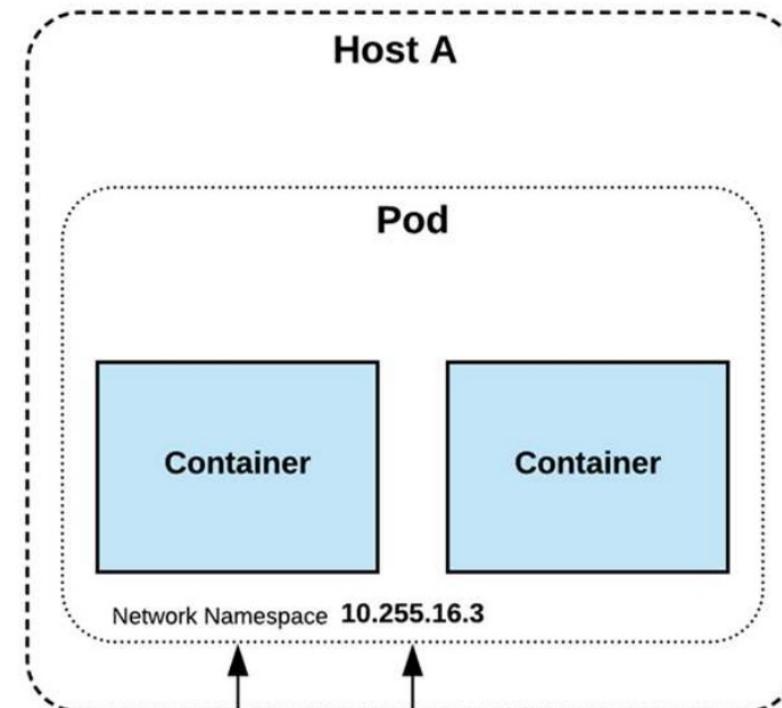
Un **pod** est la plus petite unité de travail

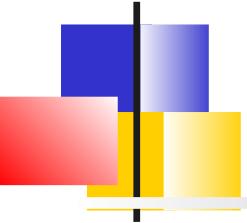
Un *pod* regroupe un ou plusieurs conteneurs qui partagent :

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

- Sont terminés
- Ont échoués
- Sont expulsés par manque de ressources





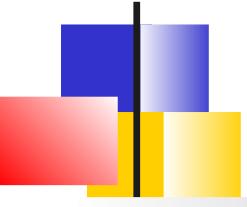
Ressource *deployment*

Exemple description d'un déploiement:

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 1
  spec:
    containers:
      - image: dthibau/annuaire
        name: annuaire
```

A partir de ce type de fichier *.yml*, on peut créer la ressource via :

kubectl create -f ./my-manifest.yaml



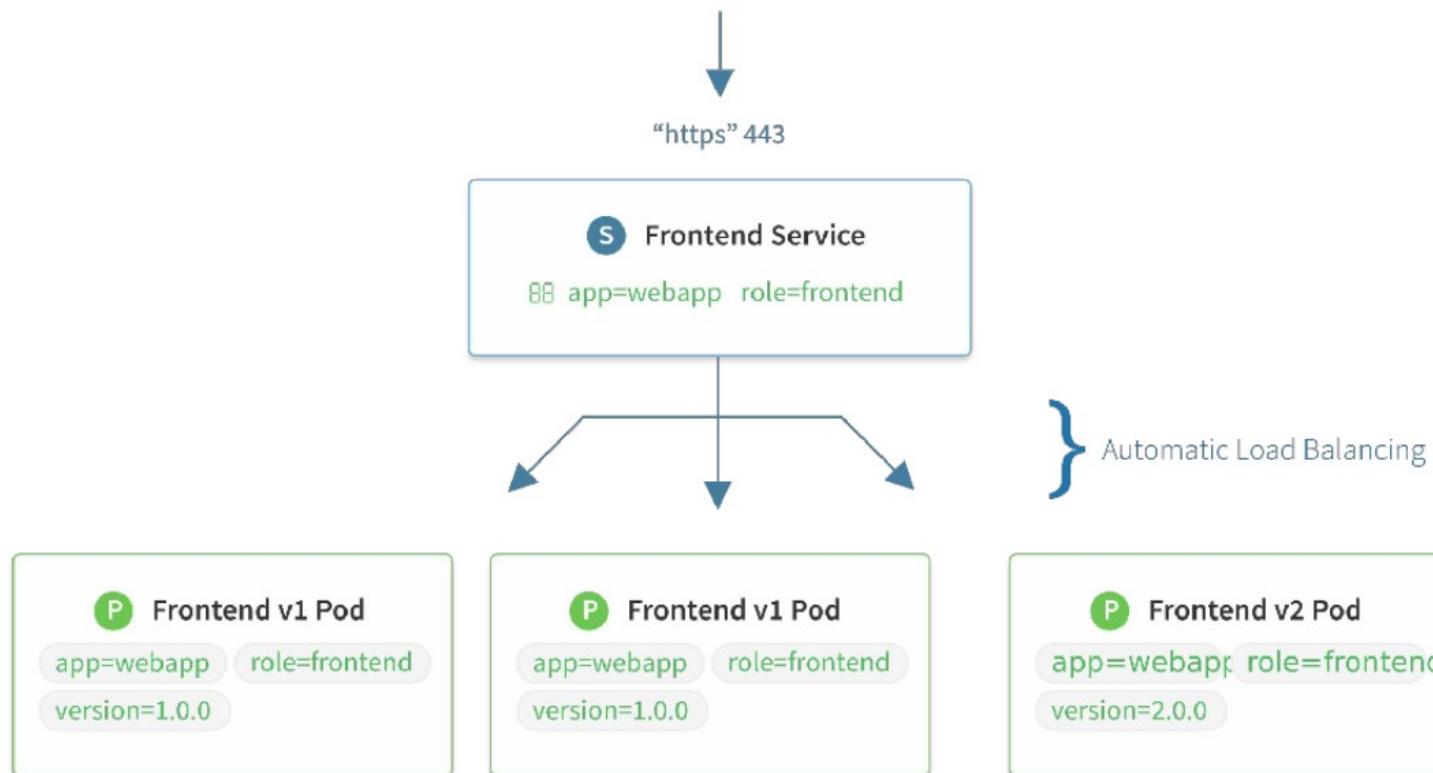
Services

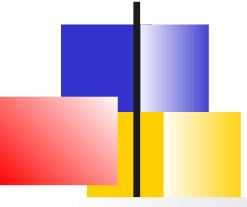
Un **service** est une méthode unifiée d'accès aux charges de travail exposées des *pods*.

Ressource durable. Les services ne sont pas éphémères :

- IP statique du cluster
- Nom DNS statique (unique à l'intérieur d'un espace de nom)

Service

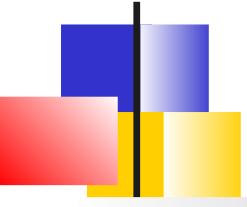




Exemple service

Un service nommé *my-service* qui représente tous les pods ayant le **label app=MyApp** et qui mappe son port 80 vers le port 80 des pods

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

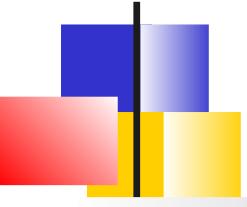


La commande *apply*

Dans la pratique, la commande ***apply*** avec en paramètre un fichier *.yml* décrivant la ressource est la plus adaptée pour des déploiements via *kubectl* :

- Elle peut créer ou modifier la ressource
- Les fichiers *.yml* décrivant les ressources à déployer sont committé, versionnés dans le dépôt des sources

kubectl apply -f ./my-manifest.yaml

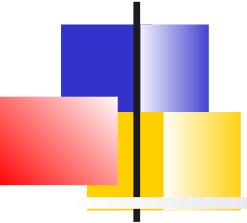


Déploiement

La ressource **deployment** permet de manipuler un ensemble de *Replicaset* (*ensemble de conteneurs répliqués*)

Les principales actions que l'on peut faire sur un déploiement sont :

- Le **rollout**: Création/Mise à jour entraînant la création des pods en arrière-plan
- Le **rollback**: Permet de revenir à une ancienne version des *ReplicaSets*
- La **scalabilité** horizontale : Permet de mettre en échelle l'application horizontalement
- La mise en pause
- La suppression de vieilles versions



Autres ressources du cluster

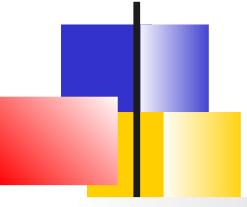
ClusterRole : Rôle avec permissions sur l'API

VolumePersistant : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

Secrets : Stockage de crédentiels



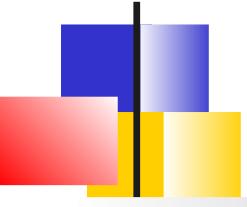
Namespace

Kubernetes prend en charge plusieurs clusters virtuels soutenus par le même cluster physique.

Ces clusters virtuels sont appelés **espaces de noms**.

- Les noms des ressources doivent être uniques dans un espace de noms, mais pas entre les espaces de noms.
- Chaque ressource Kubernetes ne peut être que dans un seul *namespace*

Les *namespaces* sont généralement utilisés dans des clusters utilisés par différentes équipes



Labels et sélecteurs

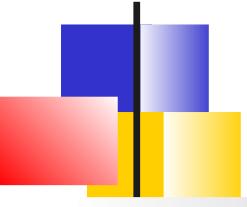
Les **labels** sont des paires clé / valeur attachées à des objets, tels que des pods, des services, des déploiements

Ils sont utilisés pour organiser et sélectionner des sous-ensembles d'objets.

Les **sélecteurs** permettent de rechercher des objets ayant des labels spécifiques.

Il y a 2 types de sélecteurs: égalité ou ensemble.

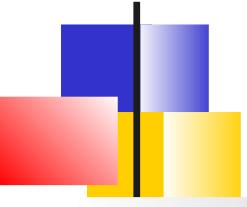
- Ils sont utilisés par les opérations *LIST* et *WATCH* de l'API
- Les services et les ReplicaSet utilisent les labels et les sélecteurs pour sélectionner les pods



Annotations

Les **annotations** (*metadata*) permettent d'attacher des métadonnées arbitraires non identifia**bles** à des objets.

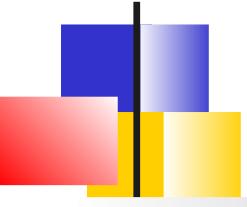
- Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.



Écosystème *Kubernetes*

De nombreux outils peuvent compléter une installation cœur de Kubernetes :

- **CoreDNS** : Permet de déclarer dans un DNS interne les services (qui deviennent accessibles via leur nom)
- **Helm** : Système de gestion de package permettant d'automatiser l'installation d'autres outils (ressources Kubernetes)
- **Prometheus** : Monitoring du cluster, généralement associé à Grafana
- **Ingress** : Permettant d'exposer les services à l'extérieur du cluster
- **Istio** : Maillage de service (services mesh), gère les communications inter-pods



Distributions Kubernetes

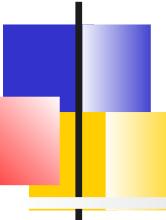
Kubernetes est disponible en OpenSource mais une installation nécessite encore beaucoup d'expertise ... et beaucoup de ressources

Kubernetes est donc proposé par les acteurs du cloud

- Amazon Elastic Container Service for Kubernetes,
Azure Kubernetes Services, Google Kubernetes
Engine, Digital Ocean, ...

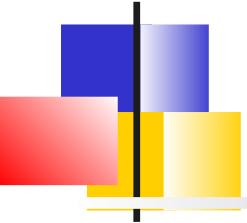
Il est également disponible en version « dev » mono-nœud : *microk8s*, *minikube*

L'outil *Rancher* permet de gérer graphiquement plusieurs installation



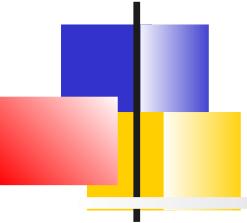
Exemple Deployment .yml (1)

```
apiVersion: extensions/v1beta1
kind: Deployment      // Type de resource Kubernetes
metadata:
  name: ftgo-restaurant-service // Le nom du déploiement
spec:
  replicas: 2 // Nombre de pods répliqués
  template:
    metadata:
      labels:
        app: ftgo-restaurant-service // Tous les pods ont ce label
    spec: // Spécification du pod
      containers:
        - name: ftgo-restaurant-service
          image: msapatterns/ftgo-restaurant-service:latest
          imagePullPolicy: Always
      ports:
        - containerPort: 8080      // Le port ouvert par le pod
          name: httpport
```



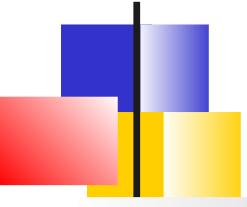
Exemple .yml (1)

```
env:
  - name: SPRING_DATASOURCE_URL    // Variables d'environnement surchargeant la configuration
    value: jdbc:mysql://ftgo-mysql/eventuate
  - name: SPRING_DATASOURCE_USERNAME
    valueFrom:
      SecretKeyRef:           // Les crédentiels BD sont stockés dans un Secret Kubernetes
        name: ftgo-db-secret
        key: username
  - name: SPRING_DATASOURCE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: ftgo-db-secret
        key: password
livenessProbe:          // Probe permettant de savoir que le service est vivant
  httpGet:
    path: /actuator/health
    port: 8080
    initialDelaySeconds: 60
    periodSeconds: 20
readinessProbe:         // Probe : le service est prêt à recevoir des requêtes
  httpGet:
    path: /actuator/health
    port: 8080
    initialDelaySeconds: 60
    periodSeconds: 20
```



Exemple Service

```
apiVersion: v1
kind: Service // Ressources Kubernetes
metadata:
  name: ftgo-restaurant-service
spec:
  port
  - port: 8080           // Port du service
    targetPort: 8080      // Port du pod cible
  selector:
    app: ftgo-restaurant-service // Label de sélection des pods
```



Zero-downtime deployment

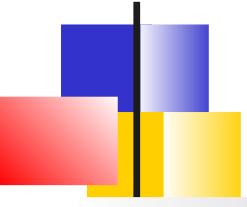
La mise à jour d'un service consiste à remplacer l'image associée au déploiement avec une nouvelle version

- Kubernetes nativement effectue un rollout progressif en remplaçant les pods exécutant la version n-1 par des pods exécutant la version n seulement lorsque ceux-ci sont prêts à recevoir des requêtes (Sonde readiness)

=> Il y a donc toujours un service qui répond, mais cela ne protège pas du déploiement d'une version buggée en production.

C'est pourquoi, il est conseillé de séparer :

- Le **déploiement** : Installer la nouvelle version sur l'infrastructure de production
- Du **releasing** : Ouvrir le nouveau service aux utilisateurs

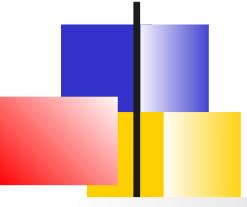


Etapes de déploiement

La séquence de mise en production devient alors :

- 1) Déployez la nouvelle version en production sans lui envoyer les demandes des utilisateurs finaux.
- 2) La tester dans l'environnement de production.
- 3) La diffuser à un petit nombre d'utilisateurs finaux.
- 4) La diffusez progressivement à un nombre de plus en plus important d'utilisateurs jusqu'à ce qu'il gère tout le trafic de production.
- 5) Si problème, revenir à l'ancienne version
- 6) Sinon supprimez l'ancienne version.

Ce type de processus peut être facilité par un service mesh

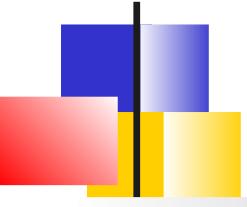


Service Mesh

Service Mesh Pattern¹ : Acheminer tout le trafic réseau entrant et sortant des services via une couche réseau qui implémentent diverses services techniques, comme les circuit-breaker, le traçage distribué, la découverte, l' équilibrage de charge et le routage du trafic basé sur des règles

Les services mesh permettent de déléguer à l'infrastructure des services techniques nécessaires aux micro-services

1. <http://microservices.io/patterns/deployment/service-mesh.html>



Istio

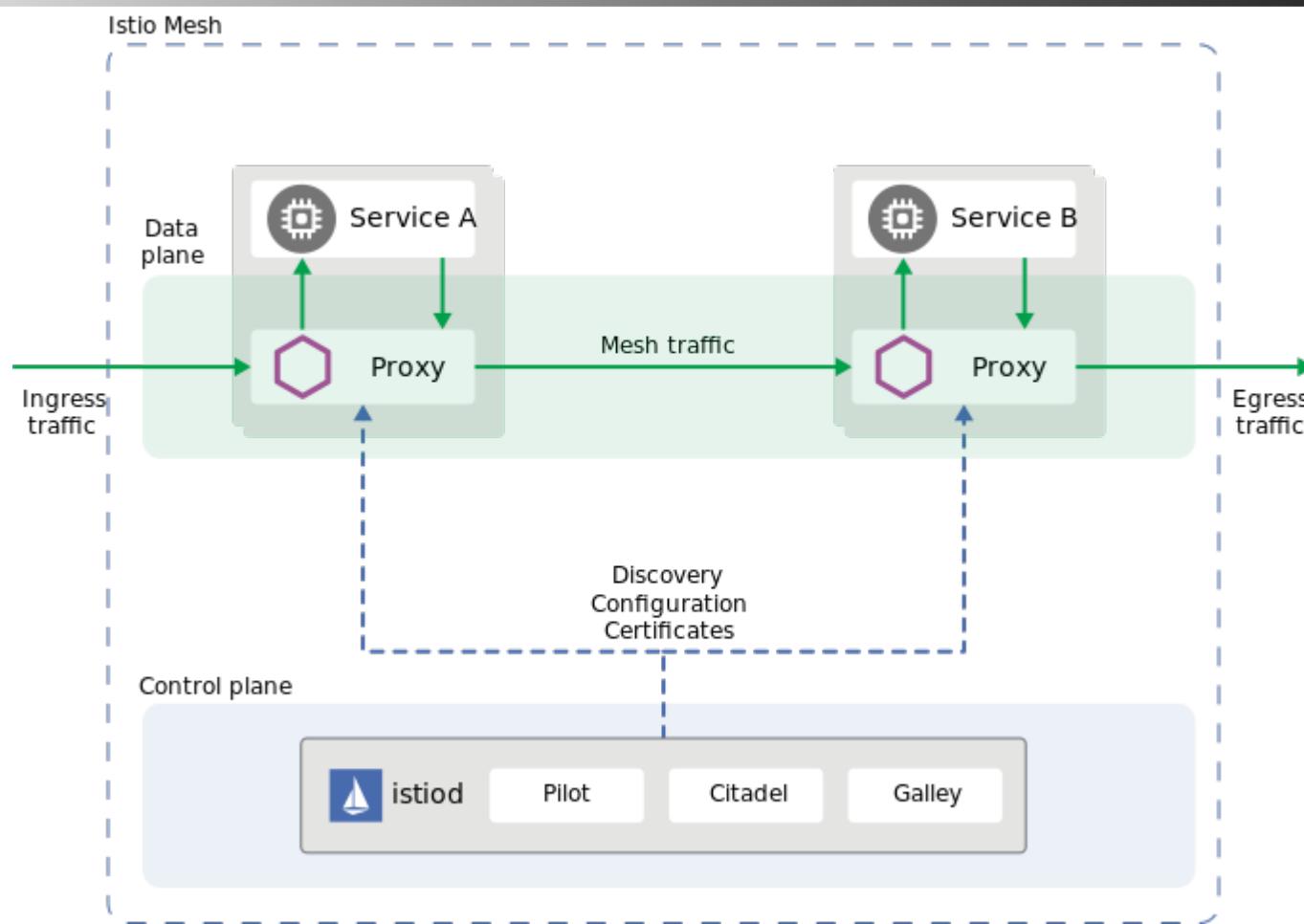
Istio est un **service mesh** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

- Un *sidecar proxy* est déployé à côté de chaque micro-service et intercepte toutes les communications.
- Un tableau de contrôle permet de gérer de façon centralisée.

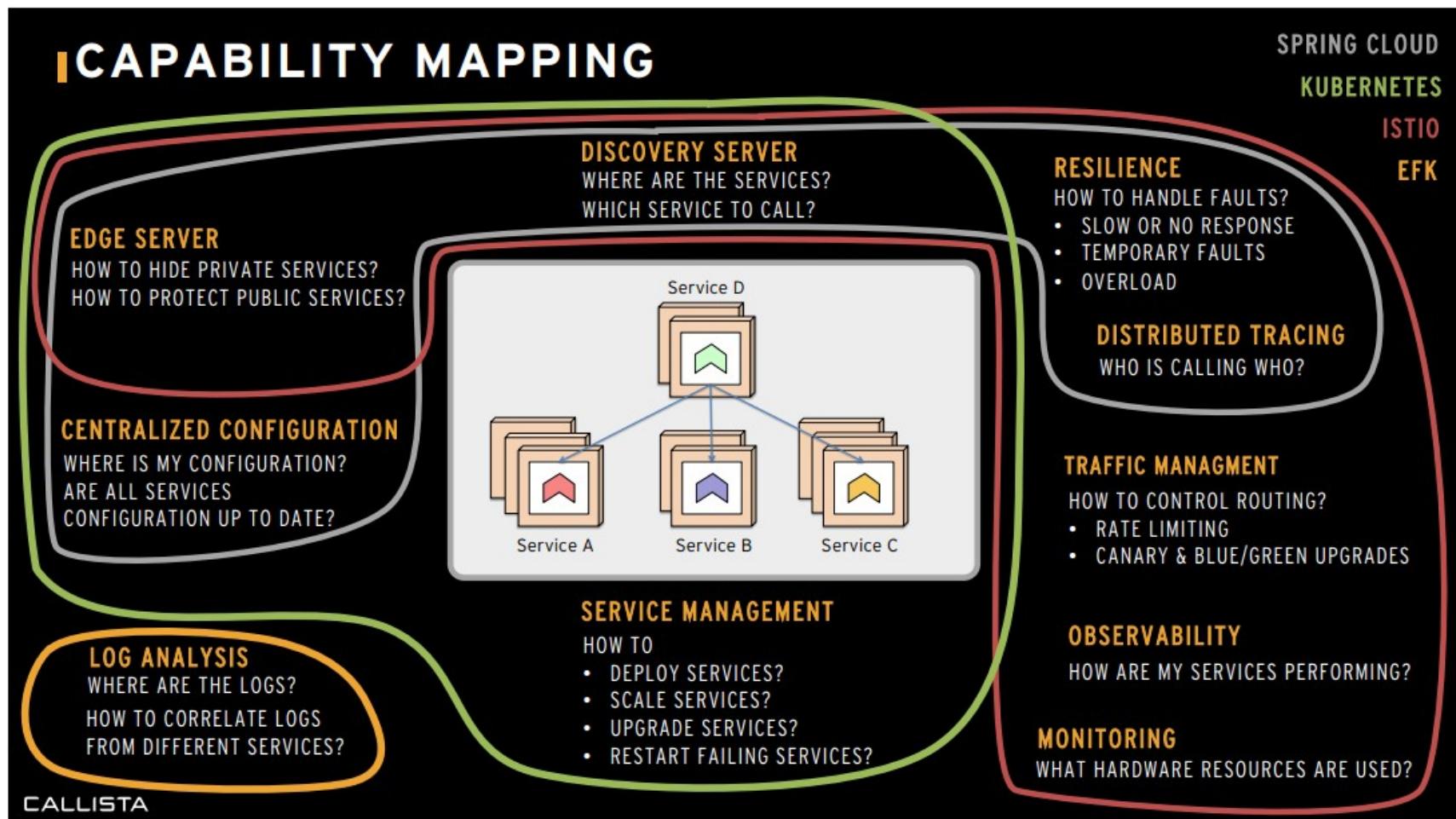
Les fonctionnalités apportées sont nombreuses :

- Équilibrage de charge automatique
- Contrôle du trafic avec des règles de routage, des politiques de ré-essai, du failover, l'injection de pannes
- Sécurisation via des contrôles d'accès, des limites de taux et des quotas.
- Sécurisation des communications via SSL et certificats.
- Collecte de métriques et tracing.

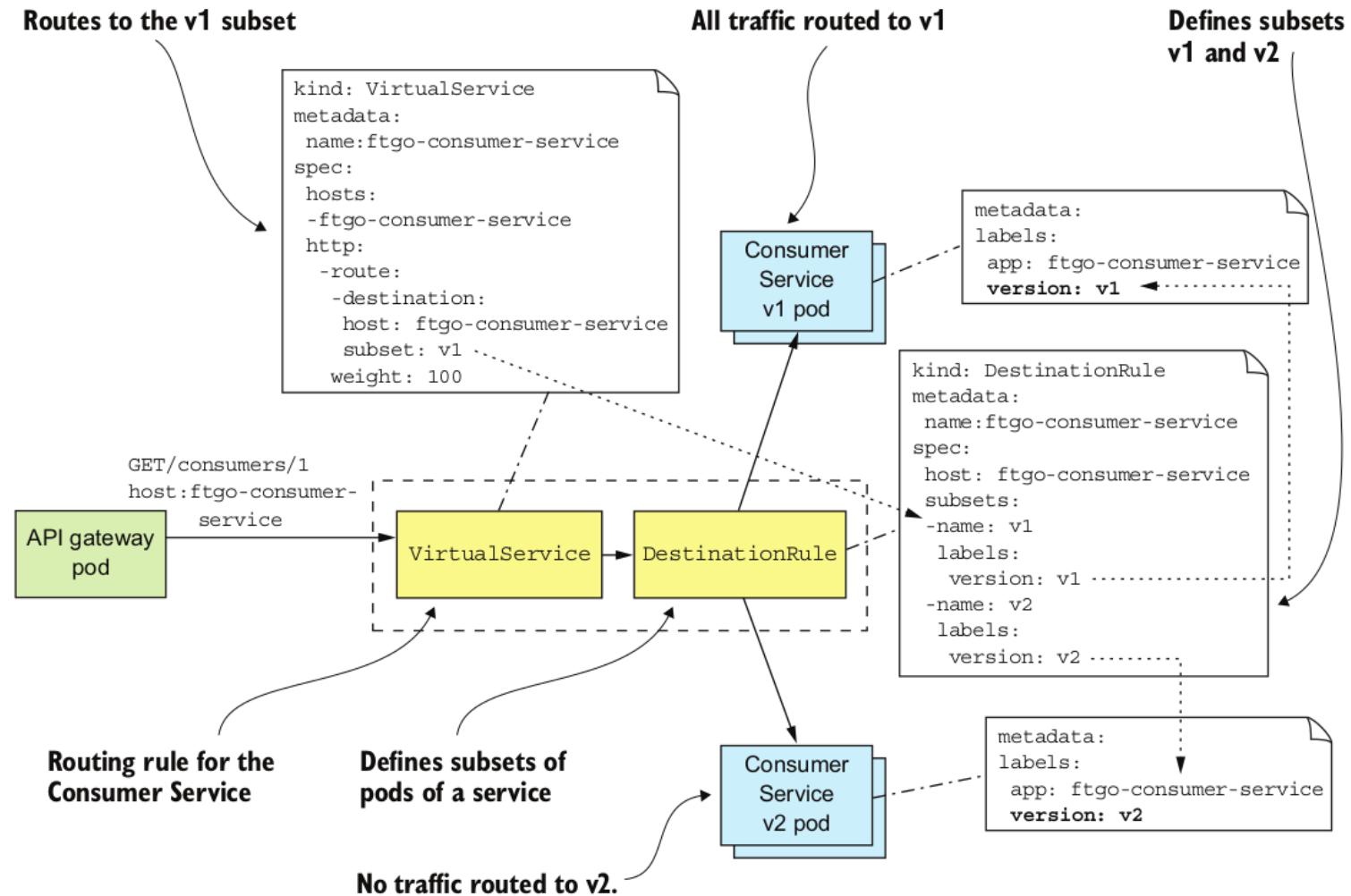
Architecture

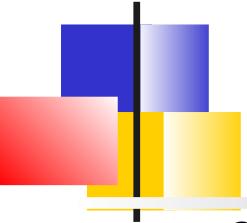


Service requis



Routage vers 2 versions





Ex. Ressource Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ftgo-consumer-service
spec:
  host: ftgo-consumer-service
  subsets:      // Définition de 2 subsets via les labels
    - name: v1    // Ces subsets seront utilisés dans des règles de routage
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
```