



Design patterns pour les microservices

David THIBAU – 2023

david.thibau@gmail.com



Agenda

- **Introduction**
 - Objectifs, Bénéfices, Contraintes, Difficultés
 - Patterns et leurs relations
 - Services transverses : Framework vs Infra
- **Décomposition**
 - Introduction
 - Business Capacity Pattern
 - Subdomain Pattern
 - Définition des APIs
- **Interactions entre services**
 - Introduction
 - RPC
 - Messaging
- **Cohérence des données et transactions**
 - Introduction
 - Saga Pattern
- **Logique métier**
 - Introduction
 - Transactional *Script Pattern*
 - Patterns orienté objet
 - *Event Sourcing Pattern*
- **Requêtage**
 - *API Composition Pattern*
 - *CQRS Pattern*
- **API Externe**
 - *Gateway Pattern*
- **Sécurité**
 - Patterns
 - Autorisation via OAuth2
 - Propagation de jeton
 - Client Credentials
- **Observabilité**
 - Health API, Métriques, Tracig distribué
- **Annexes**
 - Tests
 - Déploiement



Introduction

Objectifs, Bénéfices, Contraintes, Difficultés

Patterns et leurs relations
Services transverses



Pourquoi les micro-services ?

Pour faire évoluer le métier le plus rapidement possible.

Boucle d'amélioration courte

Pour pouvoir faire évoluer le socle technique plus facilement

Mêmes objectifs que l'Agilité, le DevOps



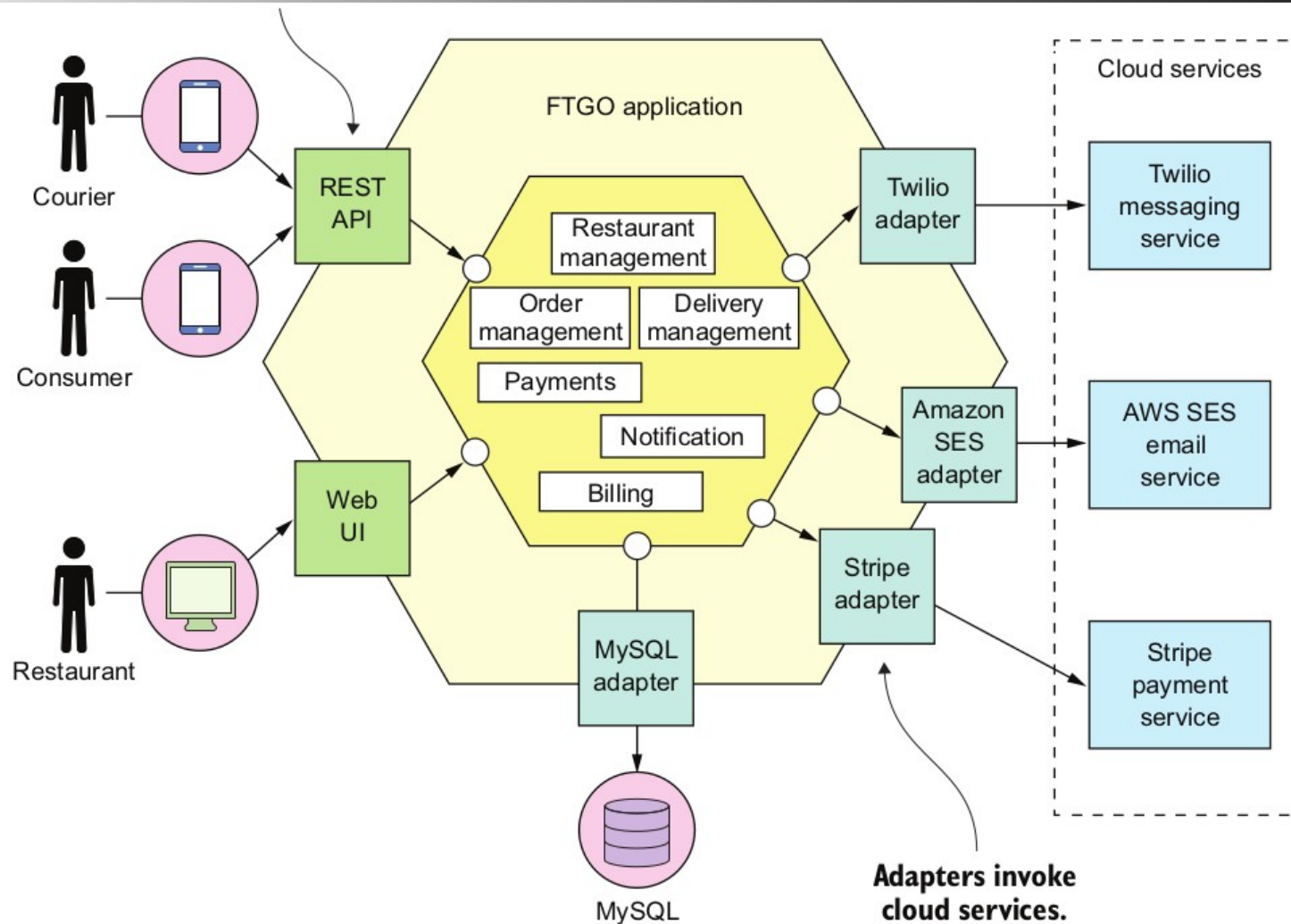
Comment ?

En découplant le problème en de plus petits problèmes indépendants !

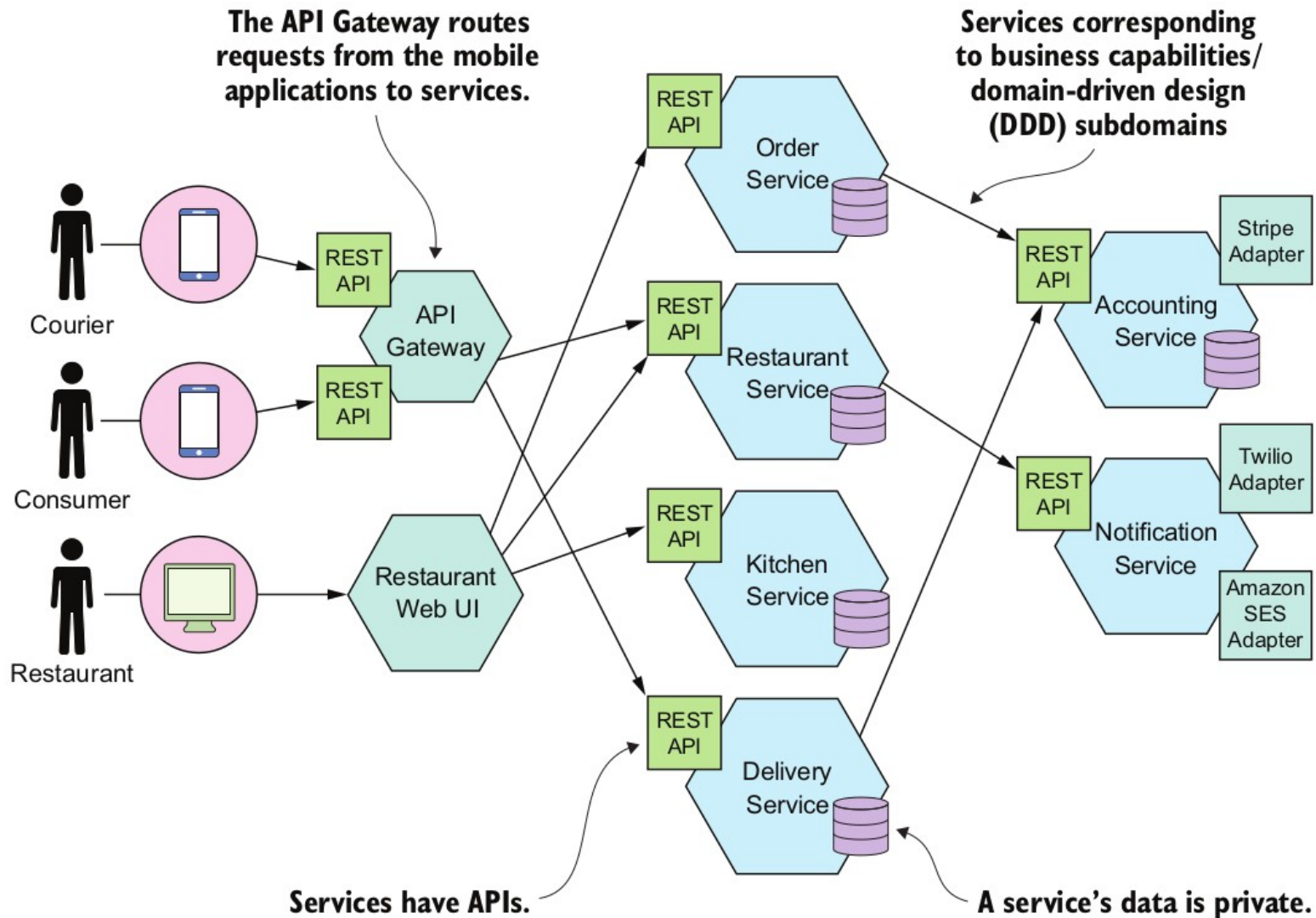
L'architecture micro-services décompose une application monolithique en de petits services

- faiblement couplés
- ayant une seule responsabilité (métier en général)
- développés par des équipes full-stack indépendantes.

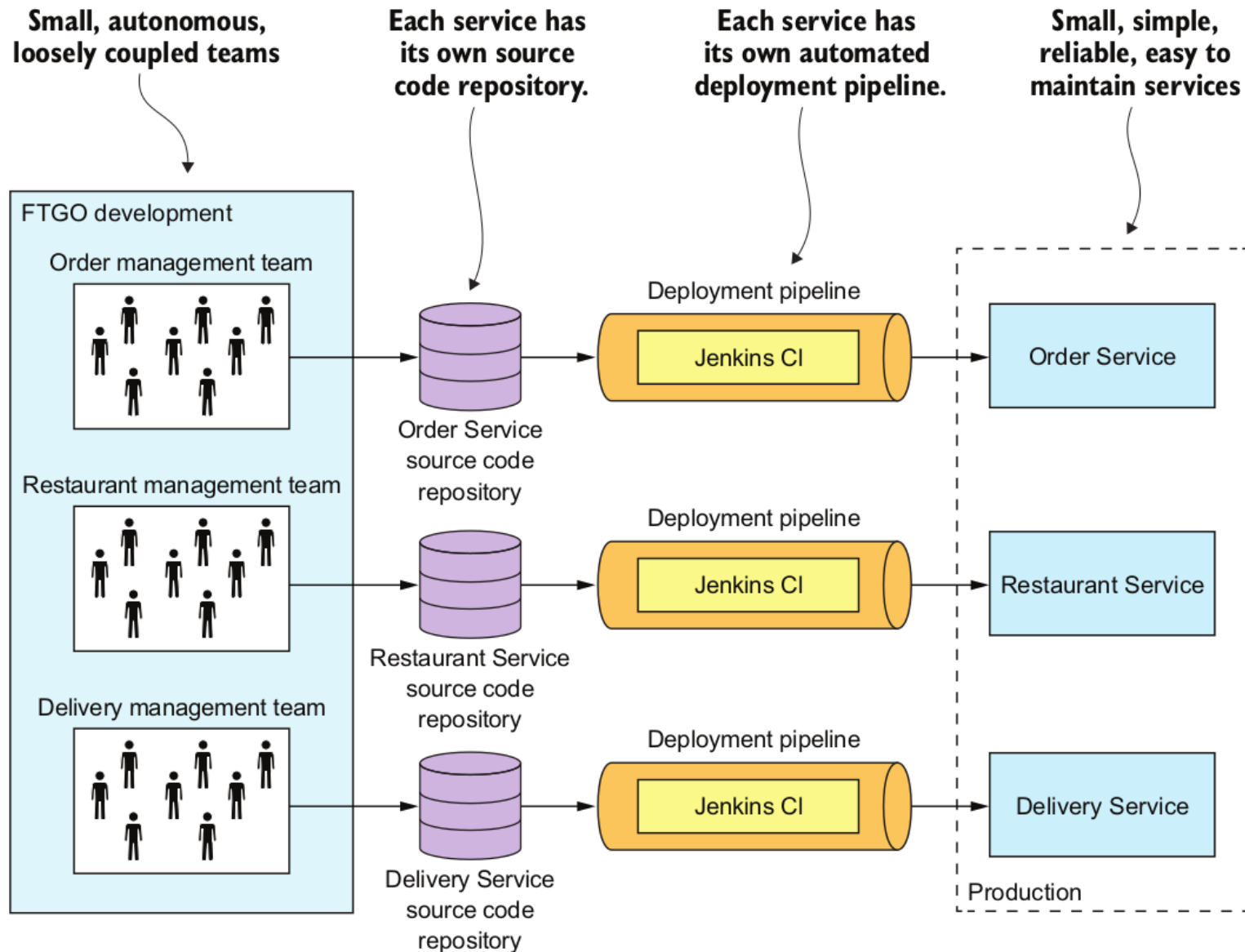
Architecture Hexagonale



Une architecture micro-service



Organisation DevOps





Bénéfices attendus

Scaling indépendant : Seuls les services les plus sollicités sont scalés
=> Économie des 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/frameworks 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



Contraintes

Réplication : Un micro-service doit être scalable
=> Design stateless favorisé
=> Nécessité d'un Service Discovery permettant de localiser les répliques

Résilience : Les probabilités d'erreur augmentent.
Le système doit être résilients aux erreurs.

Observabilité : Les services étant distribués. Il faut centraliser les points de surveillance.

DevOps : Les déploiements étant plus fréquents, il est nécessaire de les automatiser dans des pipelines CI/CD.



Difficultés de l'approche

- 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
- Immanquablement, certaines évolutions métier toucheront plusieurs micro-services et conduiront à des déploiements plus délicats
- Une entreprise n'a pas toujours les moyens de dédier une équipe à un micro-service
 - => La migration d'une application monolithique existante vers les micro-services n'est pas simple
 - => La migration s'effectue généralement progressivement



Les 12 facteurs de réussite

- I. Outil de scm** : Unique source de vérité
- II. Dépendances** : Déclarer explicitement et isoler les dépendances du code source
- III. Configuration** : Configuration externalisée du code
- IV. Services d'appui** : Les services d'appui sont des ressources attachées, possibilité de switcher sans modification de code
- V. Build, release, run** : Distinguer clairement les phases de build, release et deploy. Permet la coexistence de différentes releases en production
- VI. Processus** : Exécute l'application comme un ou plusieurs processus stateless.
- VII. Port binding** : Le service est autonome (pas de déploiement sur un serveur). Elle expose juste un port TCP
- VIII. Concurrency** : 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

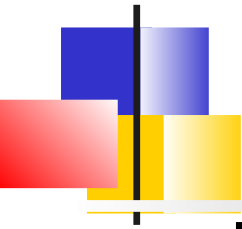


Introduction

Objectifs, Bénéfices, Contraintes,
Difficultés

Patterns et leurs relations

Services transverses



Patterns micro-service

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

- ***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.



Patterns applicatifs

Quelle Décomposition pour mes services ?

DDD/sous-domaines, Business Capability patterns

Comment définir mon API ?

Gateway pattern, Backend For FrontEnd, Agregate Model

Comment maintenir la cohérence de mes données distribuées ?

Saga Pattern

Comment interroger efficacement mes données distribuées ?

CQRS Pattern

Comment tester mes micro-services en isolation ?

Design By Contract

Comment implémenter la logique métier ?

Transaction Script, Domain Model, Domain Event, ...



Patterns infrastructure applicative

Quel mode d'interactions entre services sont nécessaires ?

RPC, Messagerie Asynchrone, Relations avec la transaction

Comment apporter la réplication ?

Service de discovery, Load Balancing

Comment apporter de la résilience ?

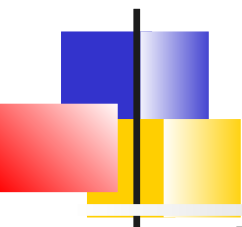
Circuit-breaker pattern, Retry, Throtling

Quels sont les moyens de l'observabilité ?

Sondes de santé, Métriques, centralisation

Sécurité : authentification, ACLs, gestion des secrets ?

SSO, OAuth2, Vault, Token Relay Pattern,



Patterns d'infrastructure

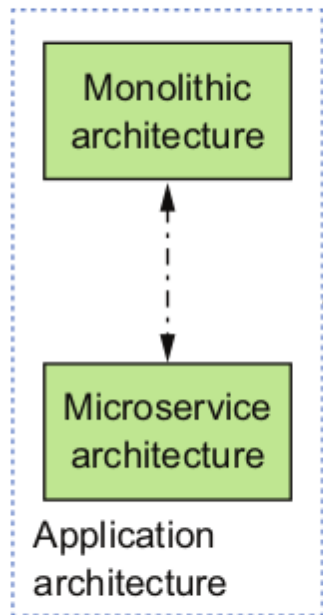
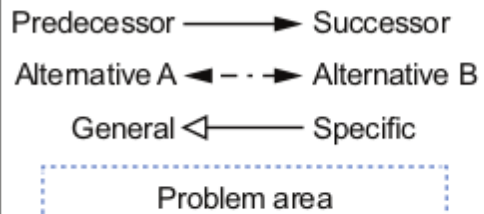
Le pattern qui s'est imposé :

« ***Deploy As A container*** »

Disposer d'un orchestrateur de containers facilite énormément le déploiement.

Les fonctionnalités natives de l'orchestrateur peuvent être augmentés par un ***service mesh***

Key



Application patterns

Decomposition

Database architecture

Querying

Maintaining data consistency

Testing

Application infrastructure patterns

Cross-cutting concerns

Security

Transactional messaging

Communication style

Reliability

Observability

Infrastructure patterns

Deployment

Discovery

External API

Communication patterns

Microservice patterns



Introduction

Objectifs, Bénéfices, Contraintes,
Difficultés
Patterns et leurs relations
Services transverses



Services Transverses

Les architectures distribuées doivent pouvoir s'appuyer sur des services transverses.

La question est : Qui implémente ces services : le framework ou l'infrastructure



Services nécessaire

Service discovery : Localiser les instances répliquées

Routing : Répartir la charge, Limiter pour protéger

Edge server : Ne pas exposer tous les services et offrir un point unique d'accès à l'architecture

Configuration centralisée : Comment gérer la configuration des services, les configurations partagées, les secrets

Observabilité : Détecter les défaillances, utilisation des ressources, performances

Analyse des traces : Centraliser et corrélérer les traces

Tracing distribué : Comment tracer une requête utilisateur dans l'architecture

Gérer les services : Comment les déployer, les scaler. En relation avec le routing : Déploiement Blue/Green, Canary

Sécurité : Cryptage du trafic, Gestion centralisée de la sécurité

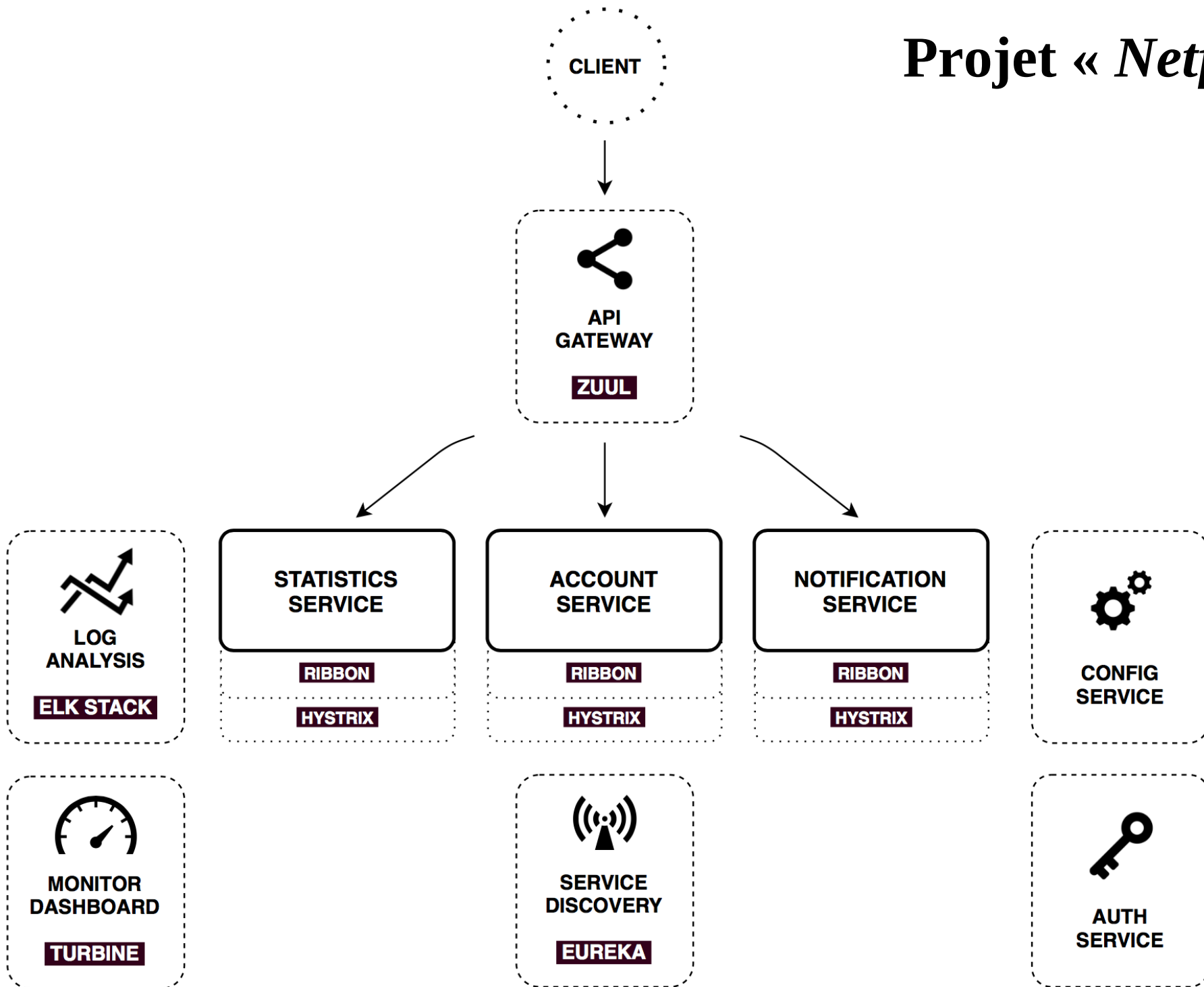


Services techniques vs Infra

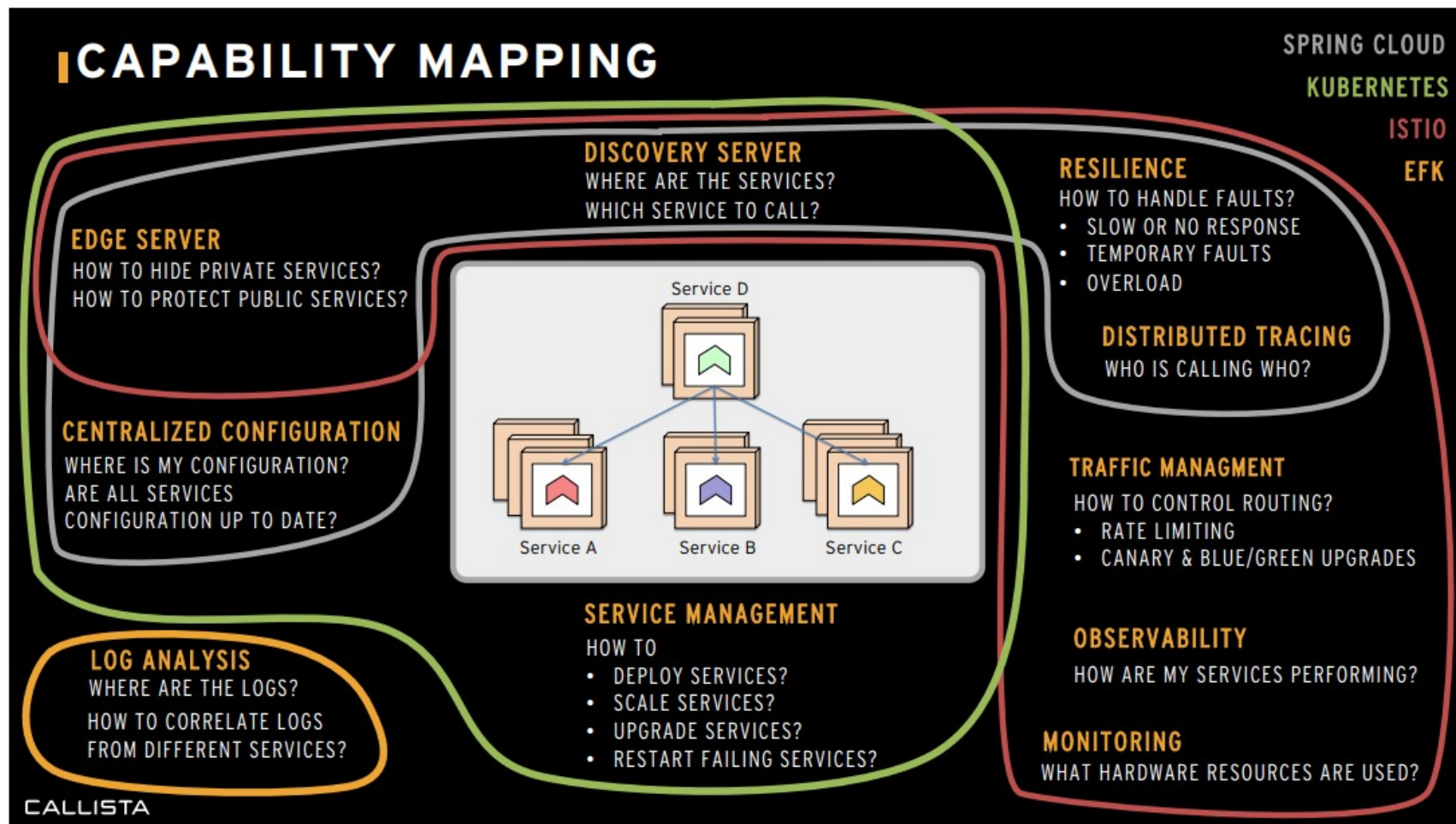
Qui fournit ces services techniques ?

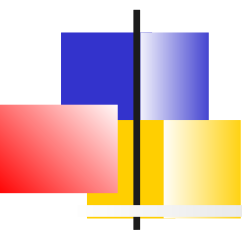
- Dans les premières architectures, c'est le software
=> Ex. framework Netflix à la base de Spring Cloud
- Actuellement, de nombreux services techniques migrent vers l'infrastructure :
 - Discovery, Config, Load Balancing offert nativement par Kubernetes
 - Résilience, Sécurité, Observabilité : Service mesh Istio

Projet « Netflix »



Capability Mapping





Stratégies de décomposition

Introduction

Business capability Pattern

Sub-Domain Pattern

Définition de l'API



Processus de définition des services

Processus itératif à 3 étapes :

- Transformer la demande métier en opérations système de 2 types :
 - Des commandes => Écriture
 - Des requêtes => Lecture de données
- Décomposer en services en fonction de concepts métier plutôt que techniques
 - Pattern « **Business Capacity** »
 - Pattern **DDD** et « Identification des sous-domaines »
- Déterminer et spécifier l'API pour chaque service



Contraintes des patterns de décomposition (1)

L'architecture doit être **stable**.

=> Les évolutions fonctionnelles ne remettent pas en cause la décomposition.

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** :

=> Une évolution fonctionnelle n'affecte qu'un seul service

Les services faiblement couplés ont une **API explicitement publiée**

=> L'implémentation peut être modifiée sans affecter les clients



Contraintes des patterns de décomposition (2)

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**.

=> Elle doit être capable de développer et de déployer ses services avec une collaboration minimale avec les autres équipes.



BD et librairies partagées

=> Les micro-services collaborent seulement via les APIs distantes, 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 une collaboration minimale avec les autres équipes.



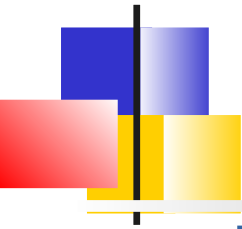
Stratégies de décomposition

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.



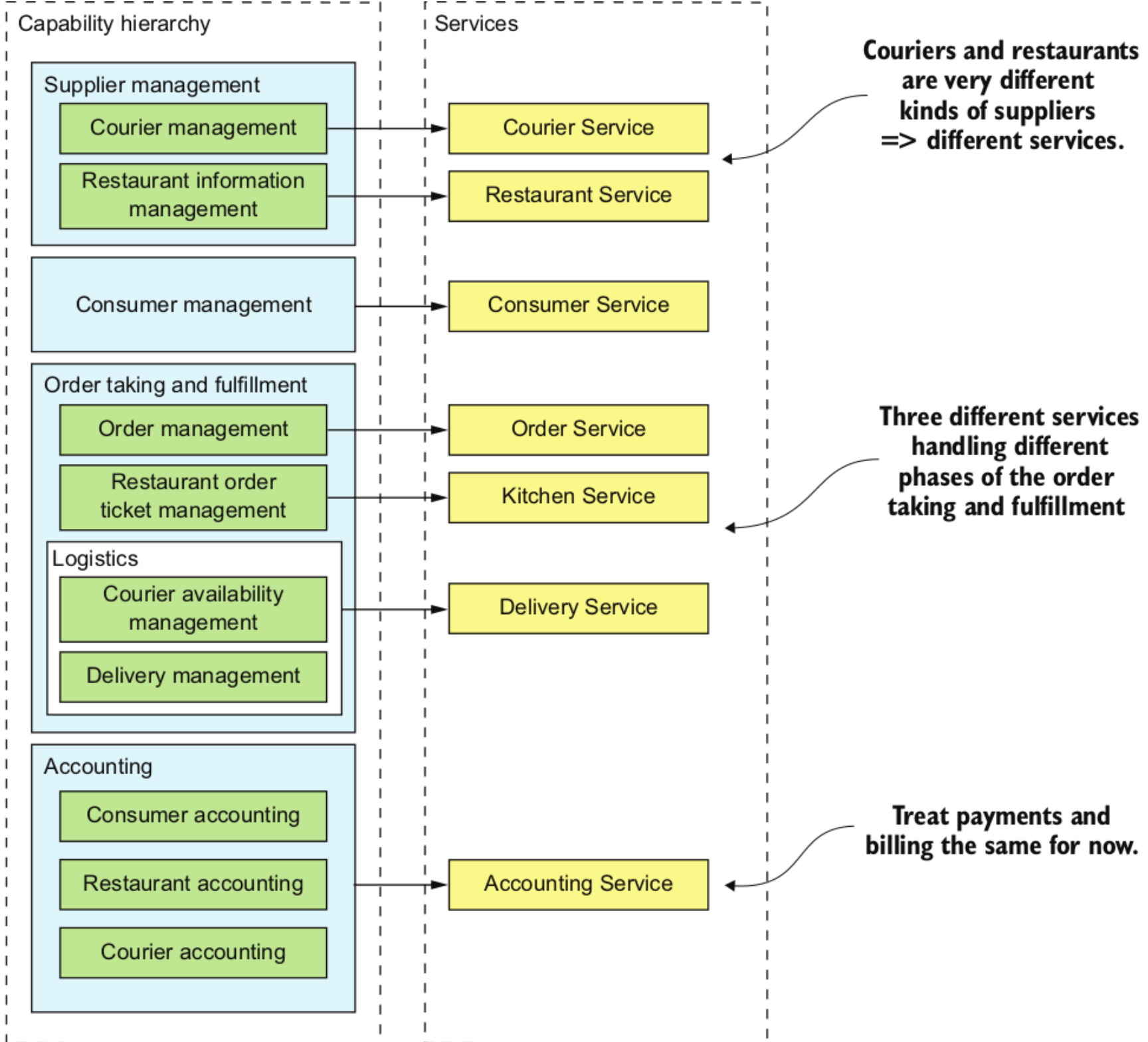
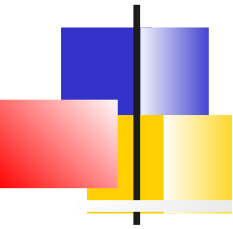
Décomposition par Capacité métier

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).

- Depuis tout temps, l'essence de la Poste est de délivrer du courrier, les technologies elles ont changées

Chaque capacité métier donne alors lieu à un service.

- Sa spécification est constituée de divers composants, y compris les entrées, les sorties et les SLA.
- Une capacité peut souvent être décomposée en sous-capacités. (pouvant nécessiter un autre service)





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

Les sous-domaines étant des sous ensembles de classes du domaine fortement liées.
(Typiquement relation de composition au sens UML)

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 2 concepts très utiles lors de la décomposition en micro-services :

- les **sous-domaines**.
- les **contextes délimités**

DDD s'applique particulièrement bien lorsque l'on a déjà un modèle monolithique existant



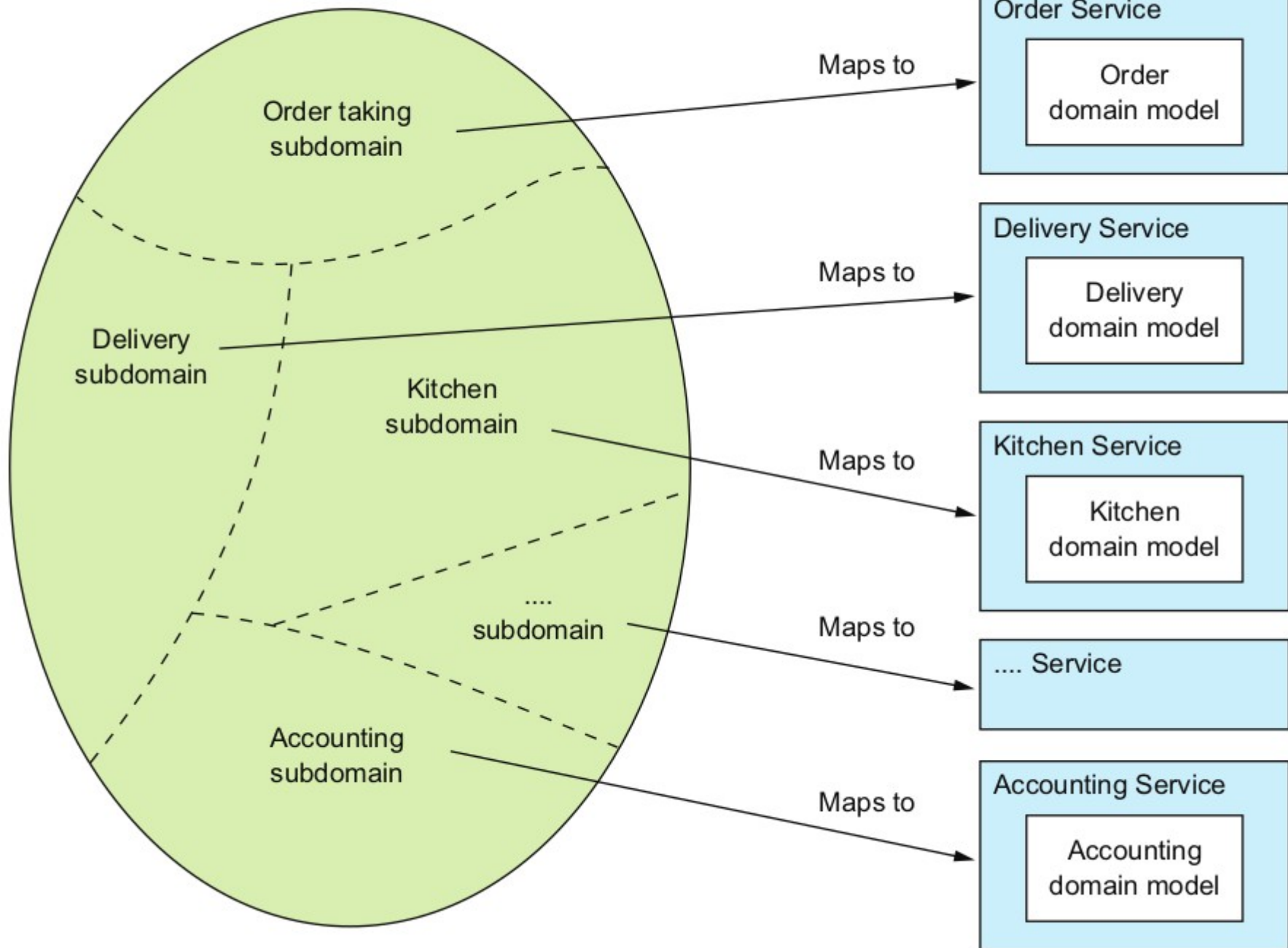
Approche traditionnelle de modélisation objet

Les monolithiques 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.

FTGO domain





Obstacles à la décomposition

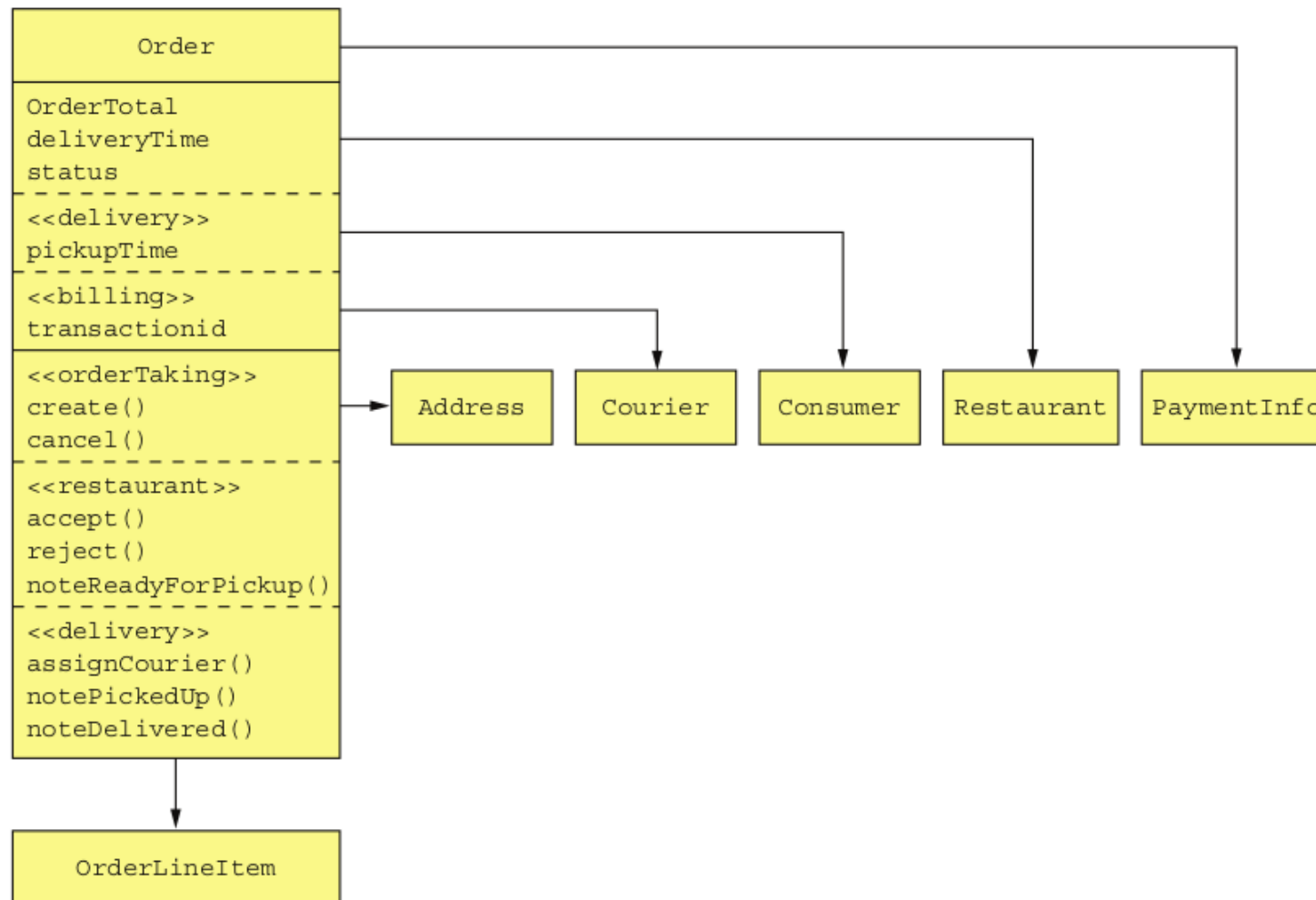
La **latence réseau** : Certaines décomposition sont infaisables à cause de trop nombreux aller-retours entre les services.

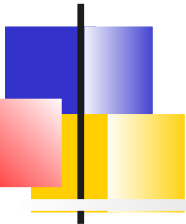
Les **communications synchrones** entre services réduisent la disponibilité

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

Les **god classes**, représentant un concept central utilisé par tous les services, .

Exemple God Class

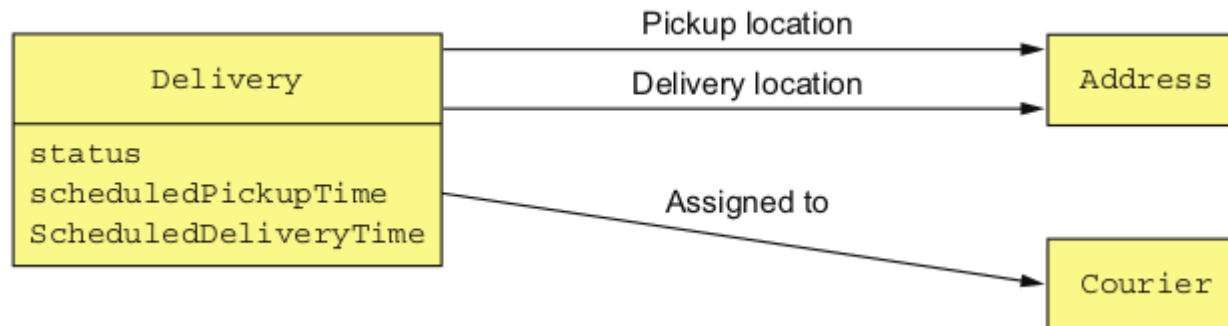




Solution apportée par la DDD

Chacun des services en relation avec la *God class* a son propre modèle de domaine avec sa propre version de la *God class*.

Par exemple, pour le service de livraison, une commande est juste un statut, et 2 dates





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:

- **Opération système.**
Elle est alors invoquée par des clients externes et peut-être par d'autres services.
- **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 consiste à mapper chaque opération système à un service.

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

- Si une collaboration est requise, on détermine quelles API ces autres services doivent fournir pour la collaboration.



Étape 1 : Mapping opération système

Service	Operations
Consumer Service	<code>createConsumer()</code>
Order Service	<code>createOrder()</code>
Restaurant Service	<code>findAvailableRestaurants()</code>
Kitchen Service	<ul style="list-style-type: none">■ <code>acceptOrder()</code>■ <code>noteOrderReadyForPickup()</code>
Delivery Service	<ul style="list-style-type: none">■ <code>noteUpdatedLocation()</code>■ <code>noteDeliveryPickedUp()</code>■ <code>noteDeliveryDelivered()</code>

Étape 2 : Collaboration entre services

Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	<ul style="list-style-type: none"> ■ Consumer Service verifyConsumerDetails() ■ Restaurant Service verifyOrderDetails() ■ Kitchen Service createTicket() ■ Accounting Service authorizeCard()
Restaurant Service	<ul style="list-style-type: none"> ■ findAvailableRestaurants() ■ verifyOrderDetails() 	—
Kitchen Service	<ul style="list-style-type: none"> ■ createTicket() ■ acceptOrder() ■ noteOrderReadyForPickup() 	<ul style="list-style-type: none"> ■ Delivery Service scheduleDelivery()
Delivery Service	<ul style="list-style-type: none"> ■ scheduleDelivery() ■ noteUpdatedLocation() ■ noteDeliveryPickedUp() ■ noteDeliveryDelivered() 	—
Accounting Service	<ul style="list-style-type: none"> ■ authorizeCard() 	—



Rôles de l'API

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 lors des évolutions d'API micro-services

Il est donc important de disposer langage de définition d'interface (IDL).

- *OpenAPI* pour Rest, *GraphiQL* pour GraphQL
- Pas de standard pour les autres interactions
- *SpringCloudContract* permet de décrire tous les types d'interactions (Rest, GraphQL, Messaging)



Evolutions

Les évolutions d'API doivent être contrôlés.

La requête ou le message doit spécifier le n° de version (URL, Entête, Format Avro, ...)

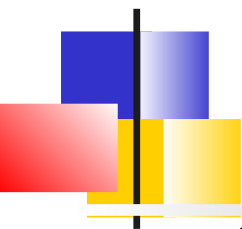
Le service doit être capable de gérer plusieurs versions en même temps :

- Déploiement des versions n et $n+1$
- Version $n+1$ comprenant les requêtes de la version n



Configuration

Configuration centralisée



Configuration centralisée

Contraintes :

- Certains micro-services partagent les mêmes valeurs de configuration.
Ex : Localisation du cluster Kafka
- La configuration doit pouvoir être mise à jour sans redémarrage des services
- Les valeurs de la configuration sont différentes en fonction des environnements ou des branches

Solution :

- Les micro-services lisent leur configuration au démarrage à partir d'un endroit centralisé.
- Des mécanismes permettent de surveiller les changements



Plusieurs solutions

Spring et SpringCloud :

- **Bootstrap** : Mécanisme Spring permettant de charger sa configuration à partir d'un serveur distant
- **Config Server** : serveur de configuration s'adossant à un dépôt Git ou un simple système de fichiers
- **Zookeeper / Consul** : Propriétés de configuration stockés dans un ensemble Zookeeper ou dans Consul
- **Spring Cloud Kubernetes** : Permet de lire sa configuration à partir des *ConfigMap* Kubernetes.
Supporte le rechargement de config dynamique

Quarkus :

- S'appuie sur **SmallRye Config** (Standard JakartaEE)
- Supporte les ConfigMap et Zookeeper



Interactions entre services

RPC
Messaging

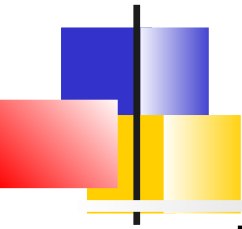


Introduction

Lors de l'utilisation d'un mécanisme RPC
(Remote Procedure Call)

- un client envoie une demande à un service,
- le service traite la demande et renvoie une réponse dans un délai court.

Certains clients peuvent se bloquer dans l'attente de la réponse, et d'autres peuvent avoir un comportement réactif, non bloquant.



Pattern

Le modèle le plus répandu est RESTFul

D'autres alternatives existent :

- GraphQL (Facebook)
- Netflix Falcor
- gRPC (Google)



Avantages de REST

C'est simple et familier.

Un format de description standard existe : ***OpenAPI***

Facile à tester : navigateur avec Postman, curl, Swagger

HTTP est compatible avec les pare-feu.

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



Inconvénients

Un seul type d'interaction

Il ne prend en charge que le style requête/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 des services ont des emplacements réseau attribués dynamiquement.

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

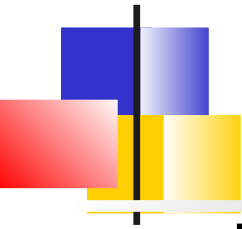
=> Par conséquent, les architectures micro-services utilisant des mécanisme RPC doivent disposer d'un **service de découverte**.



Mise en œuvre

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.



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



Patterns de discovery

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>

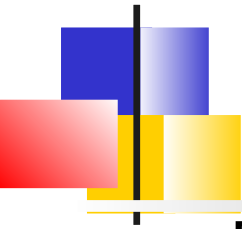


Résilience et tolérance aux pannes

Les stratégies de tolérances aux pannes sont indispensables dans les architectures fortement distribuées telles que les micro-services.

Les défaillances des services peuvent être de toute sorte :

- Défaillance réseau
- Surcharge d'un service
- Crash
- Redémarrage
- Montée de version avec interruption momentanée de service



Patterns

Différents patterns permettent d'appliquer des stratégies de résilience :

- **Timeout** : Définir une durée maximale d'exécution pour ne pas s'épuiser à appeler un service défaillant
- **Retry** : Autoriser plusieurs tentatives d'exécution pour face des défaillances momentanées
- **Bulkhead** : Limiter les exécutions concurrentes afin qu'une surcharge ne fasse écrouler le système.
- **CircuitBreaker** : Disjoncter un service présentant trop de défaillances
- **Fallback** : Fournir une solution alternative en cas d'échec d'une l'exécution



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.



Implémentations Java

MP Fault Tolerance¹ définit des annotations pour chacune des stratégies de résilience.

Dans l'environnement Spring, **Spring Cloud Circuit Breaker** fournit une abstraction et donc une API cohérente à utiliser pour les librairies implémentant le pattern circuit breaker. Il supporte :

- *Netflix Hystrix², Resilience4J (implémentation par défaut), Sentinel, Spring Retry*

1. Standard MicroProfile, Fait partie de JakartaEE

2. En maintenance



Exemple SpringCloud CircuitBreaker

Spring Cloud permet d'injecter un ***CircuitBreakerFactory***.

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

- Le code à exécuter dans une thread séparé
- Optionnellement, un code de fallback



Exemple

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

    public DemoService(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"
        );
    }
}
```



Interactions entre services

RPC
Messaging



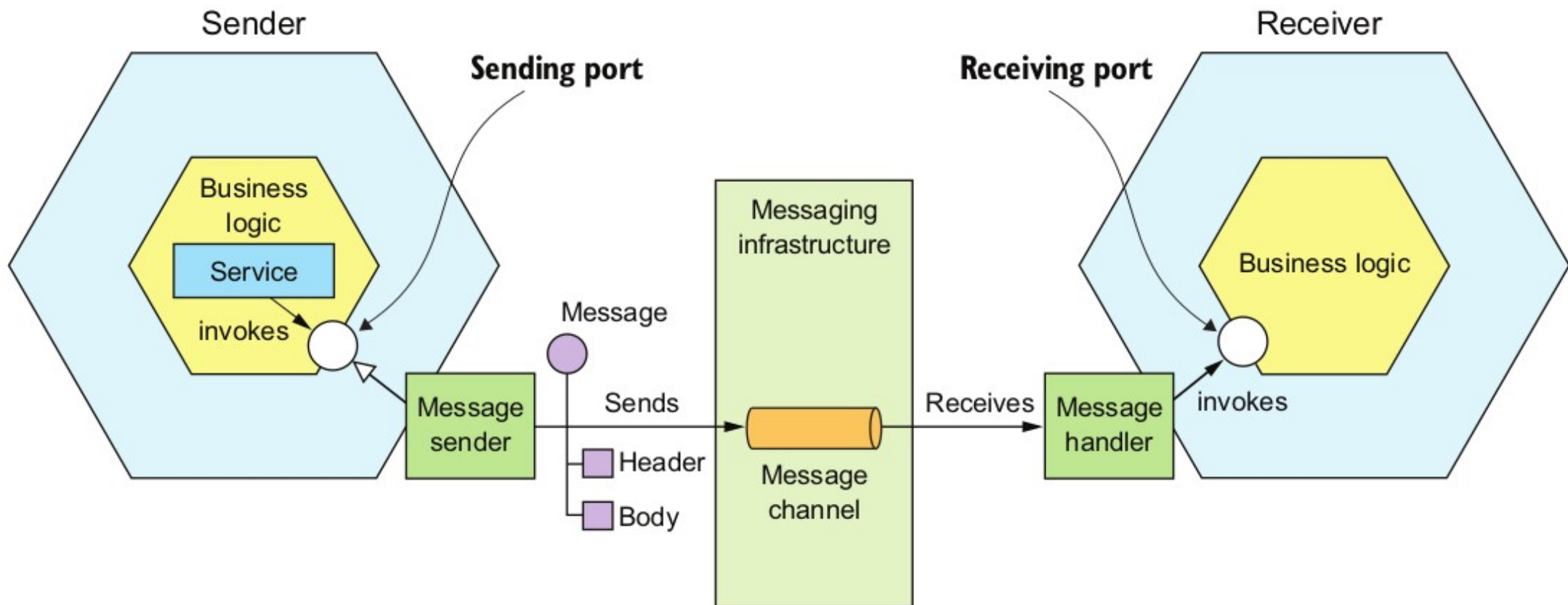
Introduction

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





Services d'un broker

Le fait de disposer d'un broker permet en général de nombreux autres patterns distribués car un broker permet un large éventail d'interaction :

- Sémantique du message : Document, Requête ou événement
- Canaux : Point 2 point ou PubSub
- Styles de communication : Requête synchrone/asynchrone, Fire and Forget, Publication vers abonnés avec ou sans acquittement
- Garanties de service : Ordre, At Most, At Least or Exactly Once

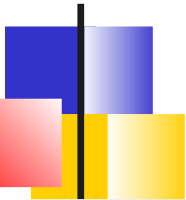


Messaging transactionnel

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

L'envoi du message doit faire partie de la transaction

- La solution traditionnelle est d'utiliser une transaction distribuée¹ qui englobe la base de données et le message broker mais cela n'est pas adapté aux applications modernes.

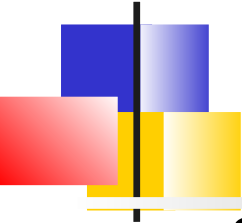


Transactional Outbox Pattern

Problème : Comment mettre à jour de manière fiable/atomique la base de données et envoyer des messages/événements ?

Forces :

- Protocole de commit à 2 phases n'est pas possible
- Si la transaction de base de données est committée, des messages doivent être envoyés.
Inversement, si la transaction est annulée, les messages ne doivent pas être envoyés
- Les messages doivent être envoyés au broker dans l'ordre dans lequel ils ont été envoyés par le service.
Cet ordre doit être conservé sur plusieurs instances de service qui mettent à jour le même agrégat.



Transactional Outbox Pattern

(2)

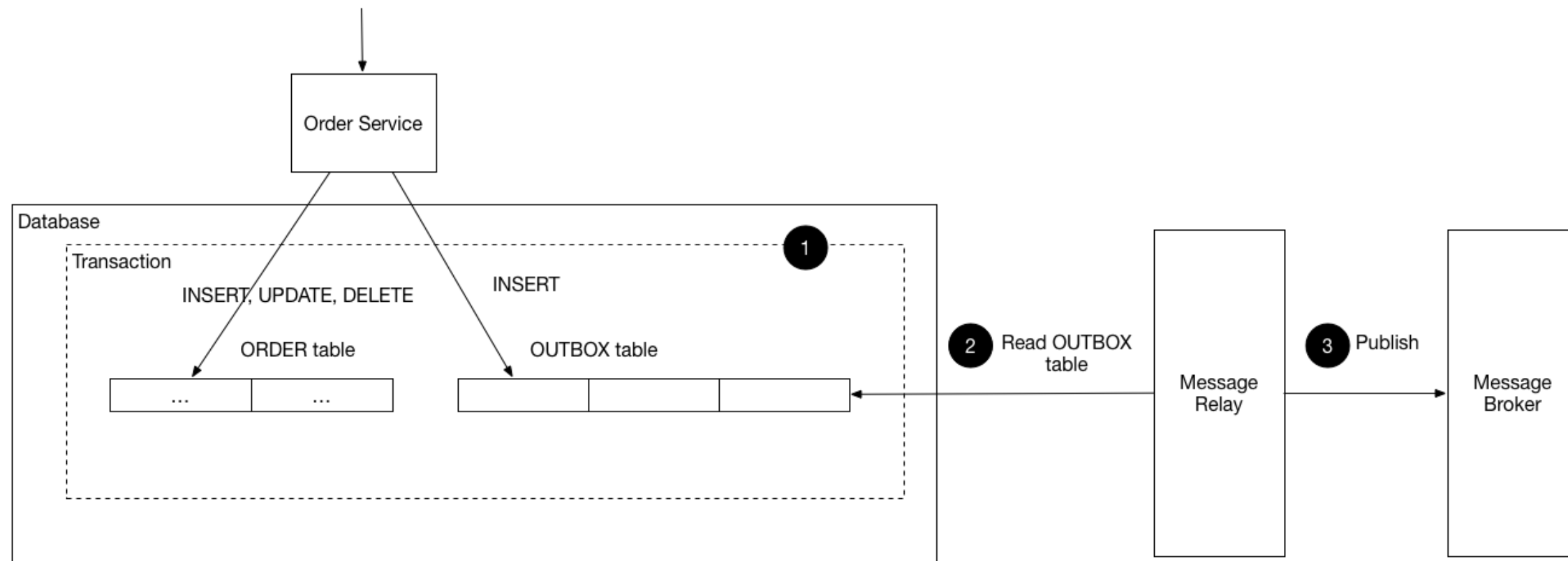
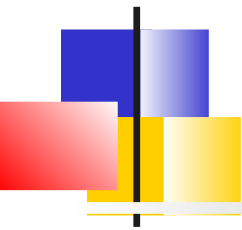
Solution :

Un service qui utilise une base de données relationnelle insère des messages/événements dans une table de boîte d'envoi (par exemple MESSAGE) dans le cadre de la transaction locale.

Un service qui utilise une base de données NoSQL ajoute les messages/événements à l'attribut de l'enregistrement (par exemple, un document ou un élément) mis à jour.

Un processus de relai de messages distinct publie les événements insérés dans la base de données vers un broker

Transactional Outbox Pattern (3)





Message Relay

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

- Soit en pollant régulièrement la base avec un SELECT : *Pattern Polling Publisher*¹
- Soit en accédant au journal de transactions. *Pattern Transaction Log Tailing*²

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

2. <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, plus compliquée si le monolithique accède plusieurs supports de persistance.

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

- C'est 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 alors 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**



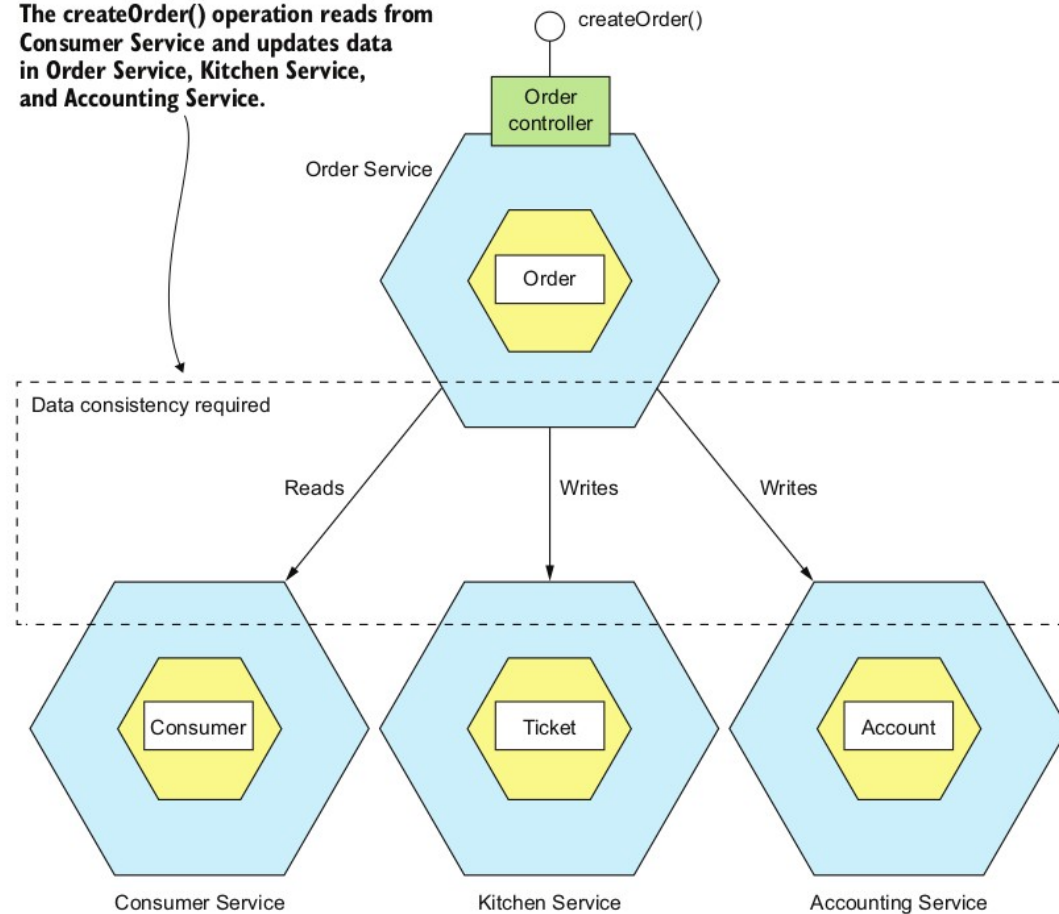
Exemple : Gestion des transactions avec Saga

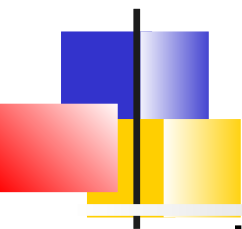
Problème : Comment implémenter des transactions englobant plusieurs services, i.e des opérations qui mettent à jour des données dispersées dans plusieurs services

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.

Example

The createOrder() operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.





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 et sont assurées de finalement réussir



Exemple

Étapes	Service	Transaction	Tr. de compensation
1	OrderService	<i>createOrder()</i>	<i>rejectOrder()</i>
2	ConsumerService	<i>verifyConsumerDetail()</i>	-
3	KitchenService	<i>createTicket()</i>	<i>rejectTicket()</i>
4	AccountingService	<i>authorizeCreditCard()</i>	
5	RestaurantService	<i>ackRestaurantOrder()</i>	
6	OrderService	<i>approveOrder()</i>	

- 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 ré-essayable, jusqu'à ce qu'elles réussissent

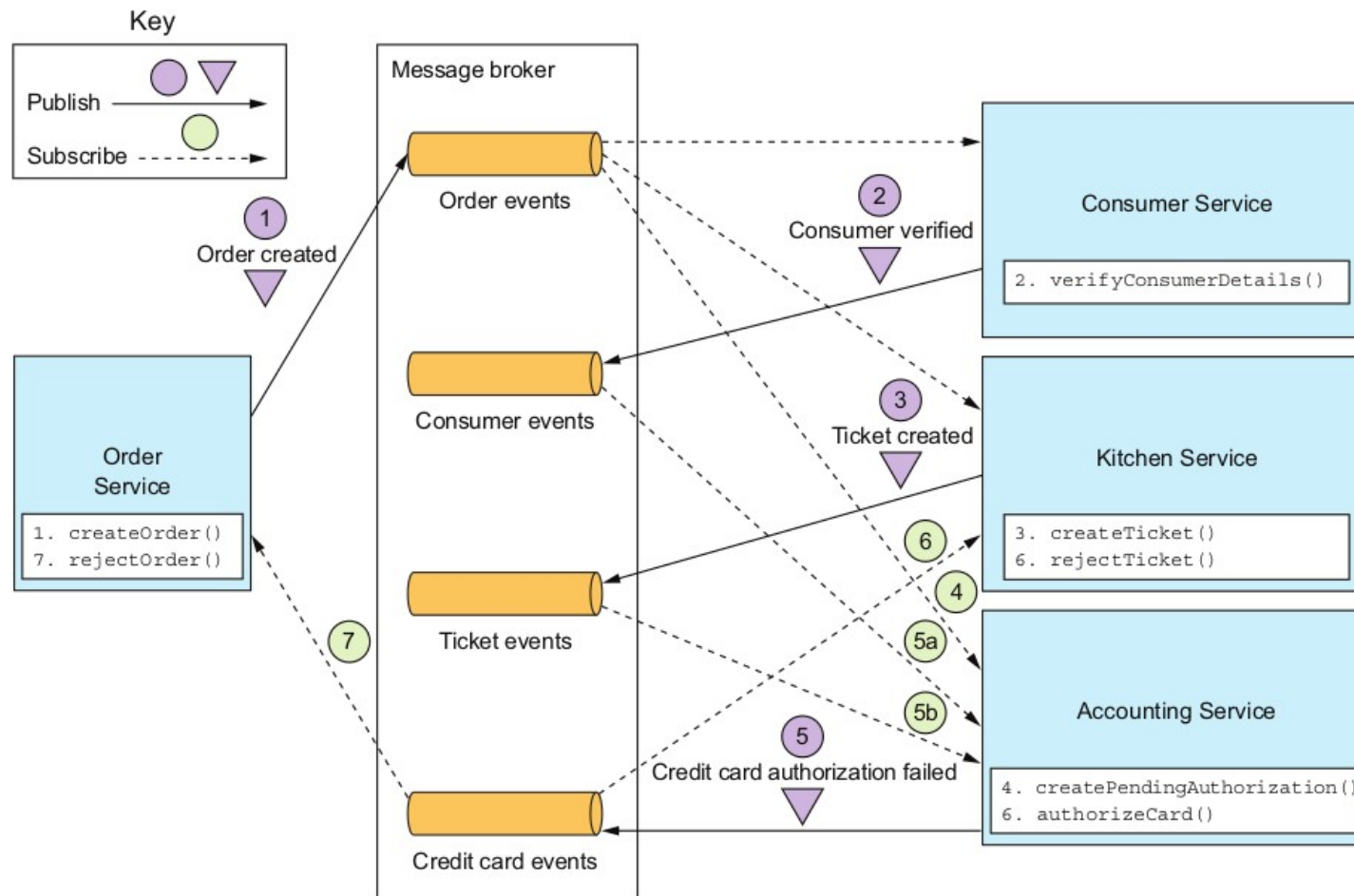


Implémentation

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.
L'orchestrateur envoie des **messages commandes** aux participants

Chorégraphie canaux d'événements





Séquence, si tout se passe bien

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



Séquence si cela se passe mal

1. *OrderService* crée une commande dans l'état **APPROVAL_PENDING** et publie un événement **OrderCreated**.
 1. *ConsumerService* utilise l'événement *OrderCreated*, vérifie que le consommateur peut passer la commande et publie un événement **ConsumerVerified**.
 2. *KitchenService* consomme l'événement *OrderCreated*, valide la commande , crée un ticket dans un état **CREATE_PENDING** et publie l'événement **TicketCreated**.
 3. *AccountingService* consomme l'événement *OrderCreated* et crée une *CreditCardAuthorization* dans un état **PENDING**.
2. *AccountingService* consomme les événements *TicketCreated* et *ConsumerVerified*, échoue à débiter la carte de crédit et publie un événement **CreditCardAuthorizationFailed**.
3. *KitchenService* consomme l'événement *CreditCardAuthorizationFailed* et change l'état du billet en **REJECTED** .
4. *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. (*Domain Event Pattern*)

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

Inconvénients

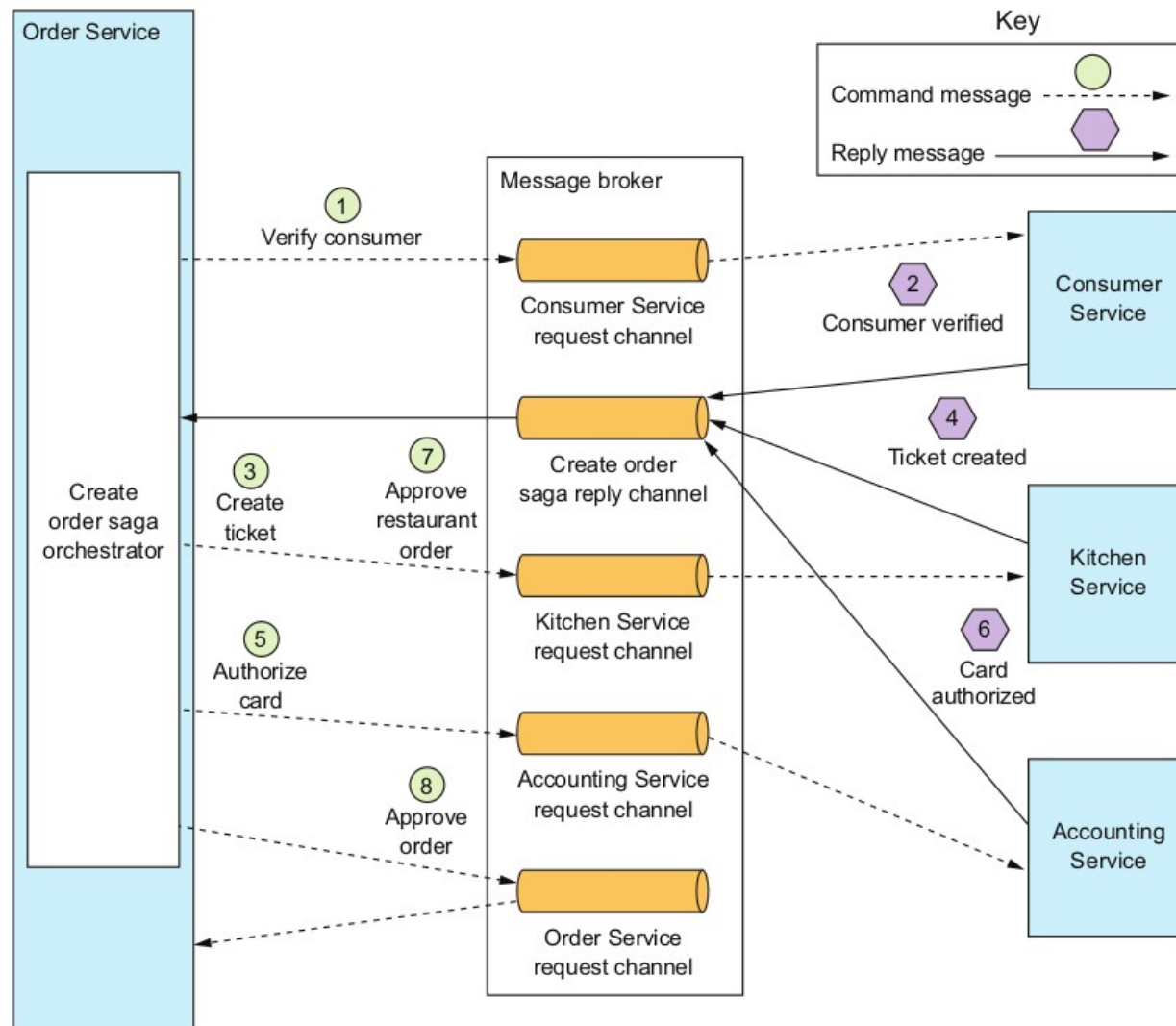
Plus difficile à comprendre : Contrairement à l'orchestration le code est décentralisé.

Dépendances cycliques entre les services : Les participants à la saga s'abonnent aux événements des autres, ce qui peut créer 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

Canaux request/reply





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*.



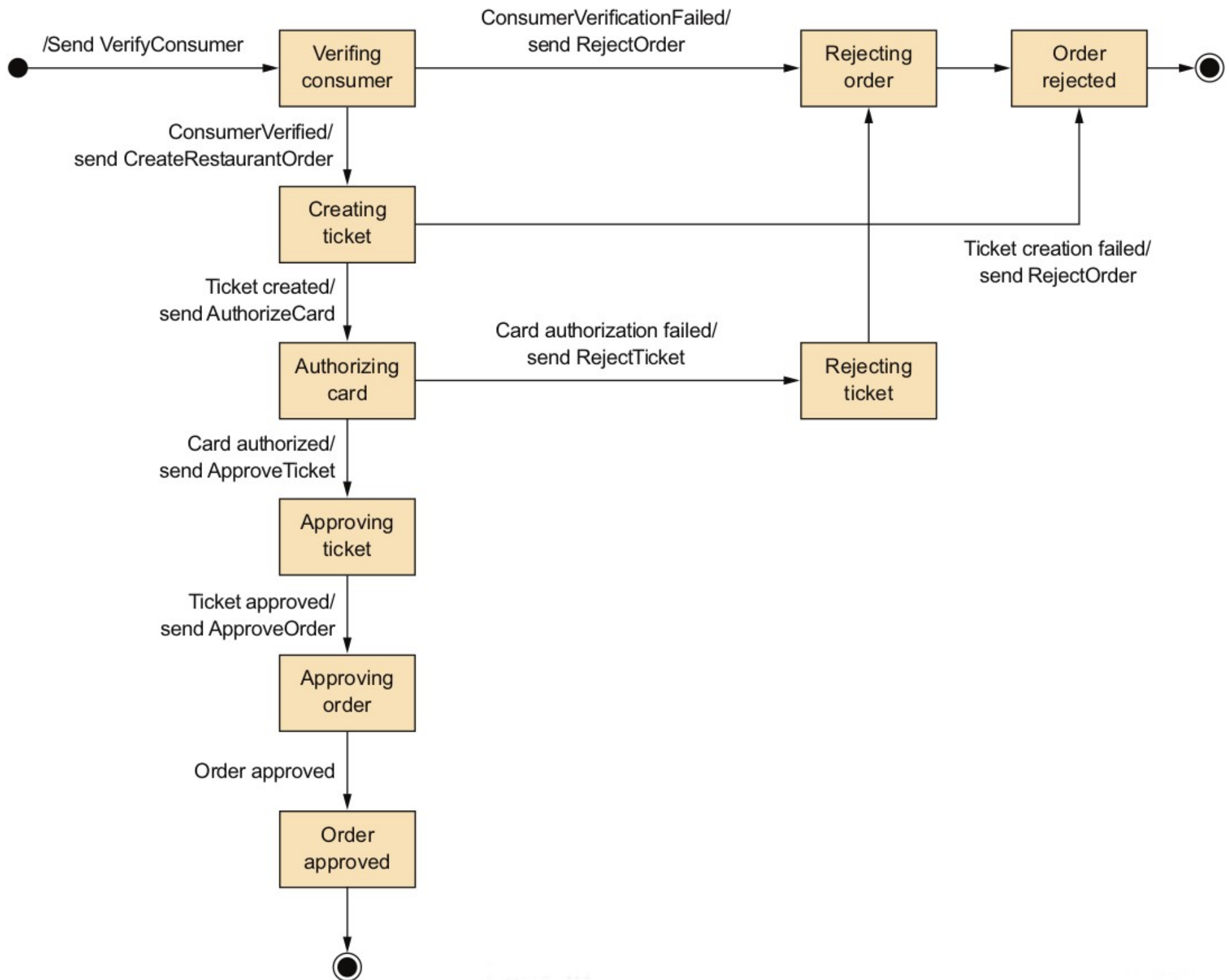
Modélisation d'une saga

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 messagerie transactionnelle

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

=> Nécessité d'implémenter une messagerie transactionnelle via *Transactional Outbox* par exemple

=> Possibilité d'implémenter le pattern en mode RPC (RestFul par exemple)



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 que 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 par les autres sagas concurrentes, 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.
- **Relecture** : 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.

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

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



Exemple : Verrou sémantique

Les transactions compensables d'une saga 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.



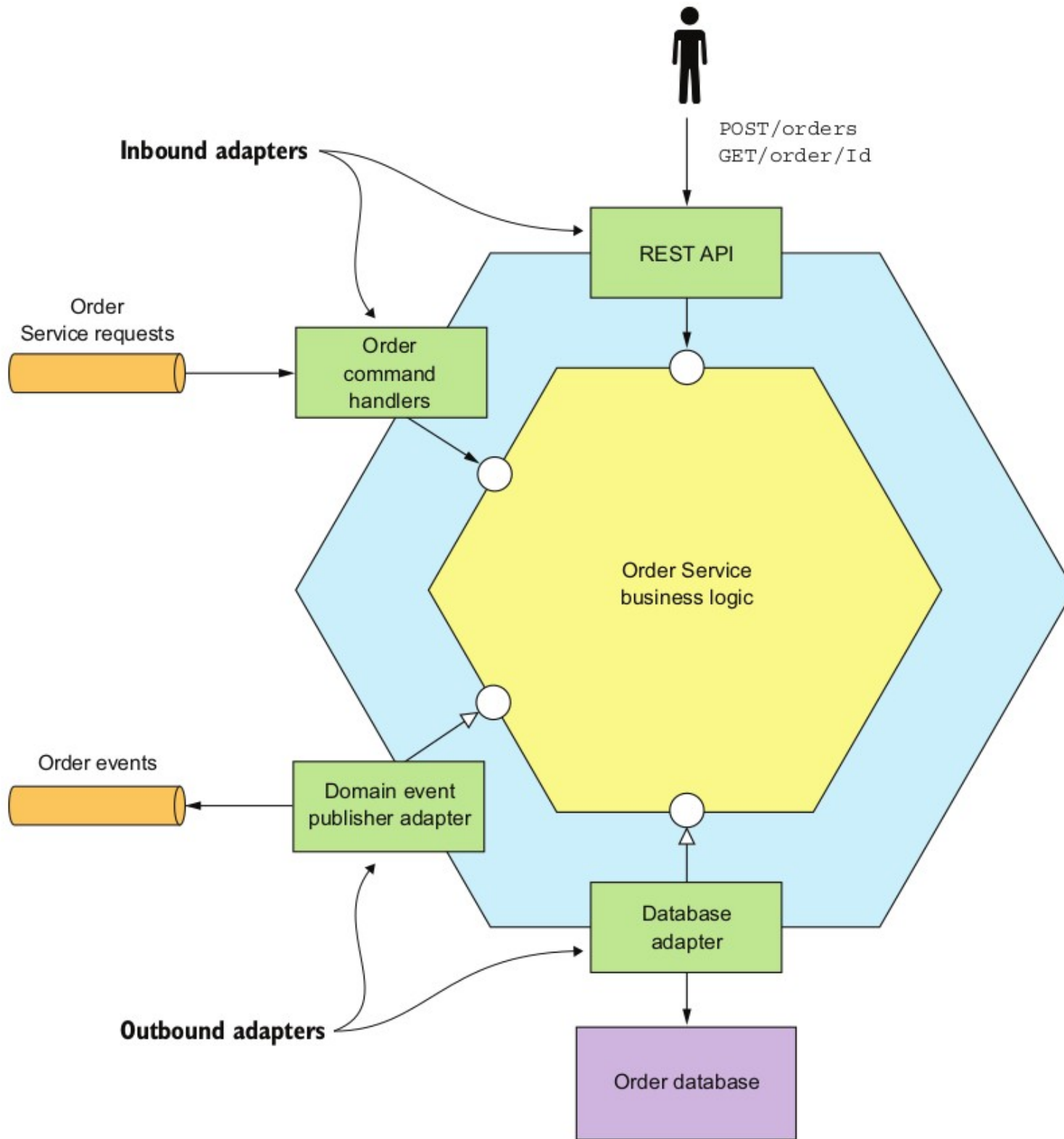
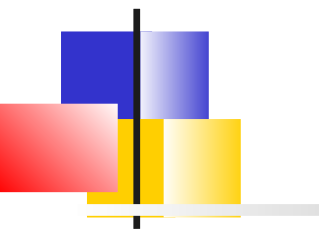
Logique métier

Introduction

Transaction Script Pattern

Patterns OO

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*.



Logique métier

Introduction

Transaction Script Pattern

Patterns OO

Event Sourcing Pattern

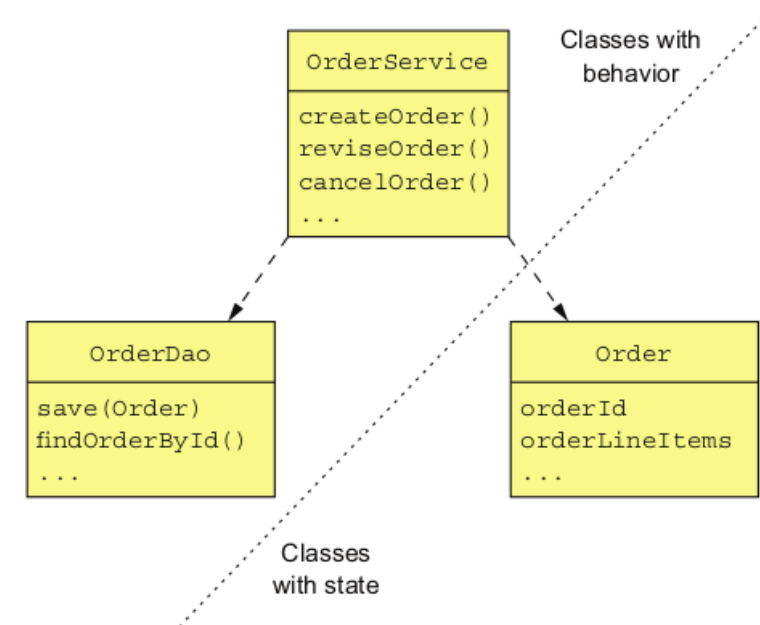
Transaction Script Pattern

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



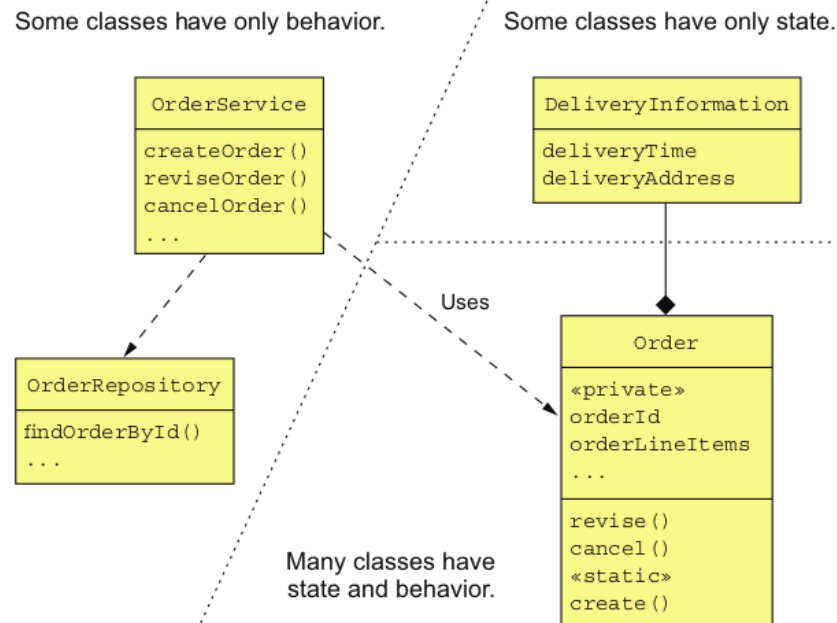


Logique métier

Introduction
Transaction Script Pattern
Patterns OO
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/eaCatalog/domainModel.html>



Avantages du Domain Model pattern

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

- Elle délègue 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*¹ ou *Template Method*² sur les classes du domaine

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

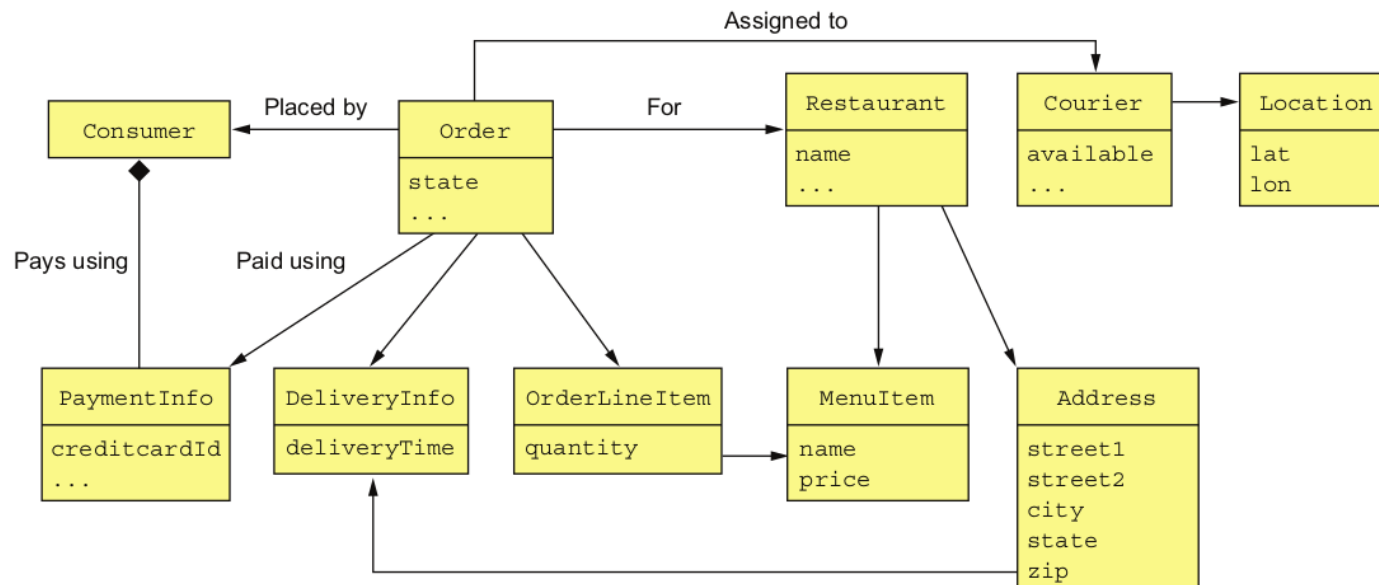
2. https://en.wikipedia.org/wiki/Template_method_pattern

Délimitation du Domain Model

Un modèle de domaine est un ensemble de classes et de relations entre les classes.

Lors de l'implémentation de la logique métier de la classe *Order*, quelles relations sont gérées ?

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 concernant le même *Order*, travaille au niveau d'un de ces items
 - Chacune modifie un item dans l'*Order*
 - Avant chaque transactions, l'invariant est vérifié et si il est respecter. La transaction peut se valider
 - Mais au final, on peut se retrouver avec un *Order* ne respectant plus l'invariant



Aggregate Pattern

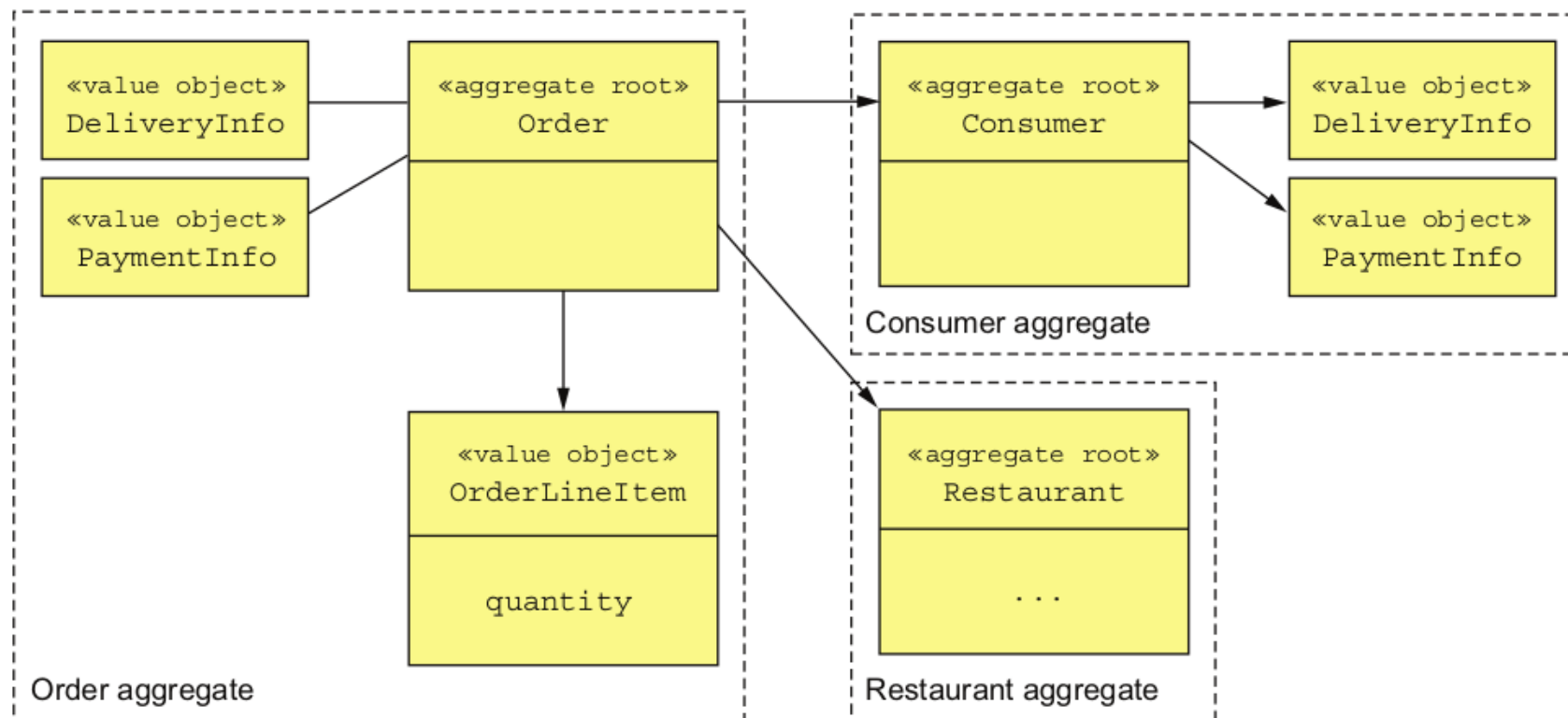
Aggregate Pattern¹ : Organise le modèle du domaine comme une collection d'agrégats, i.e. un graphe d'objets traité unitairement

- Un agrégat est constitué d'une entité racine et éventuellement composé d'autres entités ou d'objets valeur
- 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

Voir également <https://blog.engineering.publicissapient.fr/2018/06/25/craft-les-patterns-tactiques-du-ddd/>

Example





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 crée 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 intéressées 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 publiés vers un message broker.

- La création de l'évènement concerne typiquement l'agrégat (*Domain Model Pattern*)
- Afin que l'agrégat n'ait pas de dépendance sur l'API de messaging, il est 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);  
    }  
}
```



Logique métier

Introduction
Transaction Script Pattern
Patterns OO
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

Plus de détails en annexe

1. <http://microservices.io/patterns/data/event-sourcing.html>



Requêtage

API Composition Pattern
CQRS Pattern



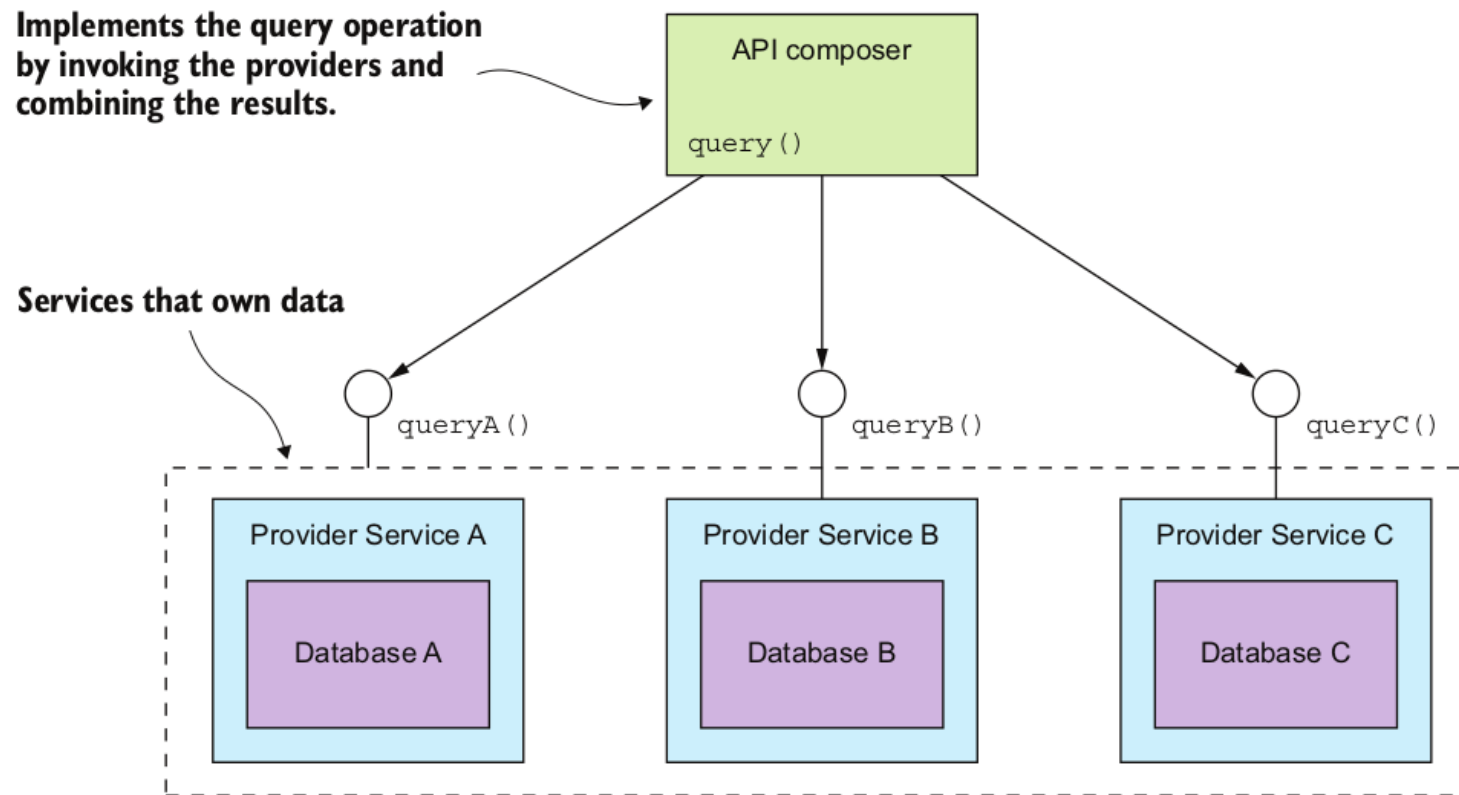
Introduction

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 résultats provenant de différents services.
- ***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 mettre en place 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

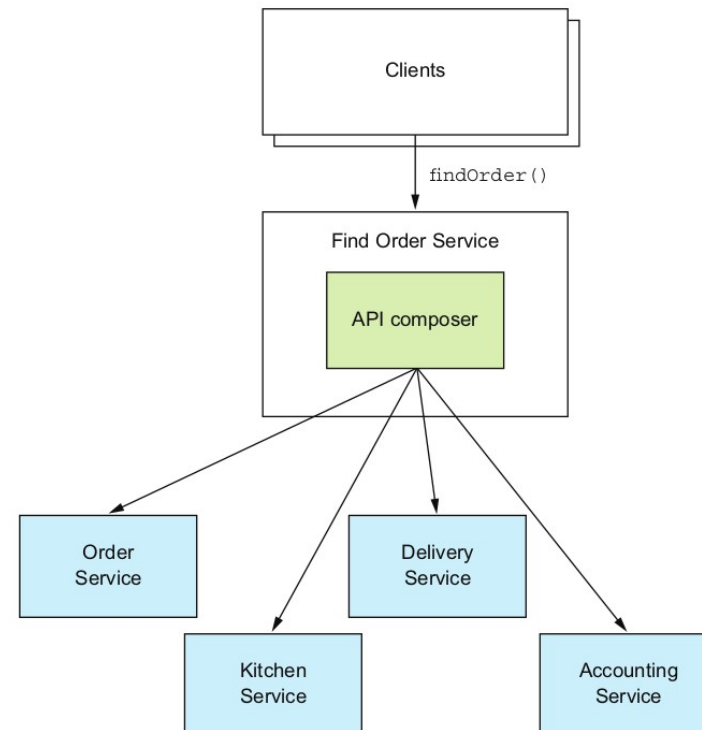
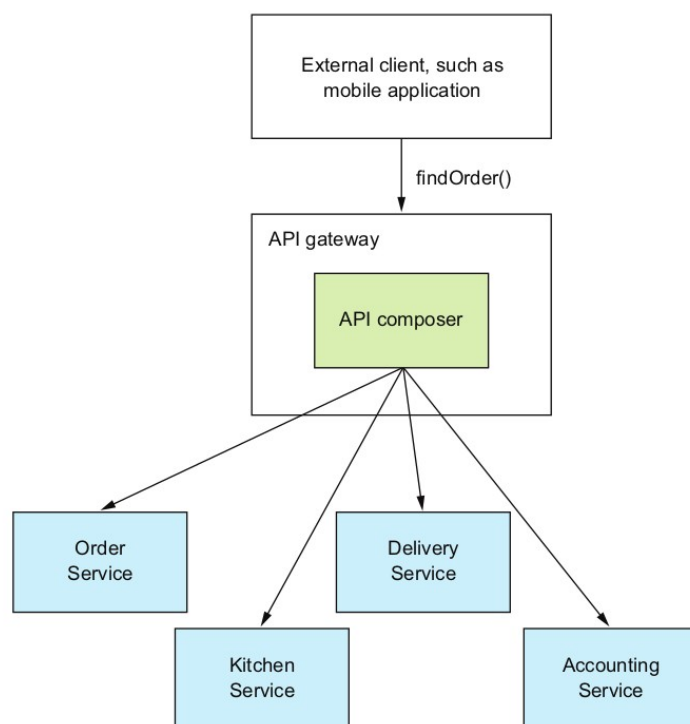
Composer

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 composée 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



Requêtage

API Composition Pattern
CQRS Pattern



CQRS

Command query responsibility

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>



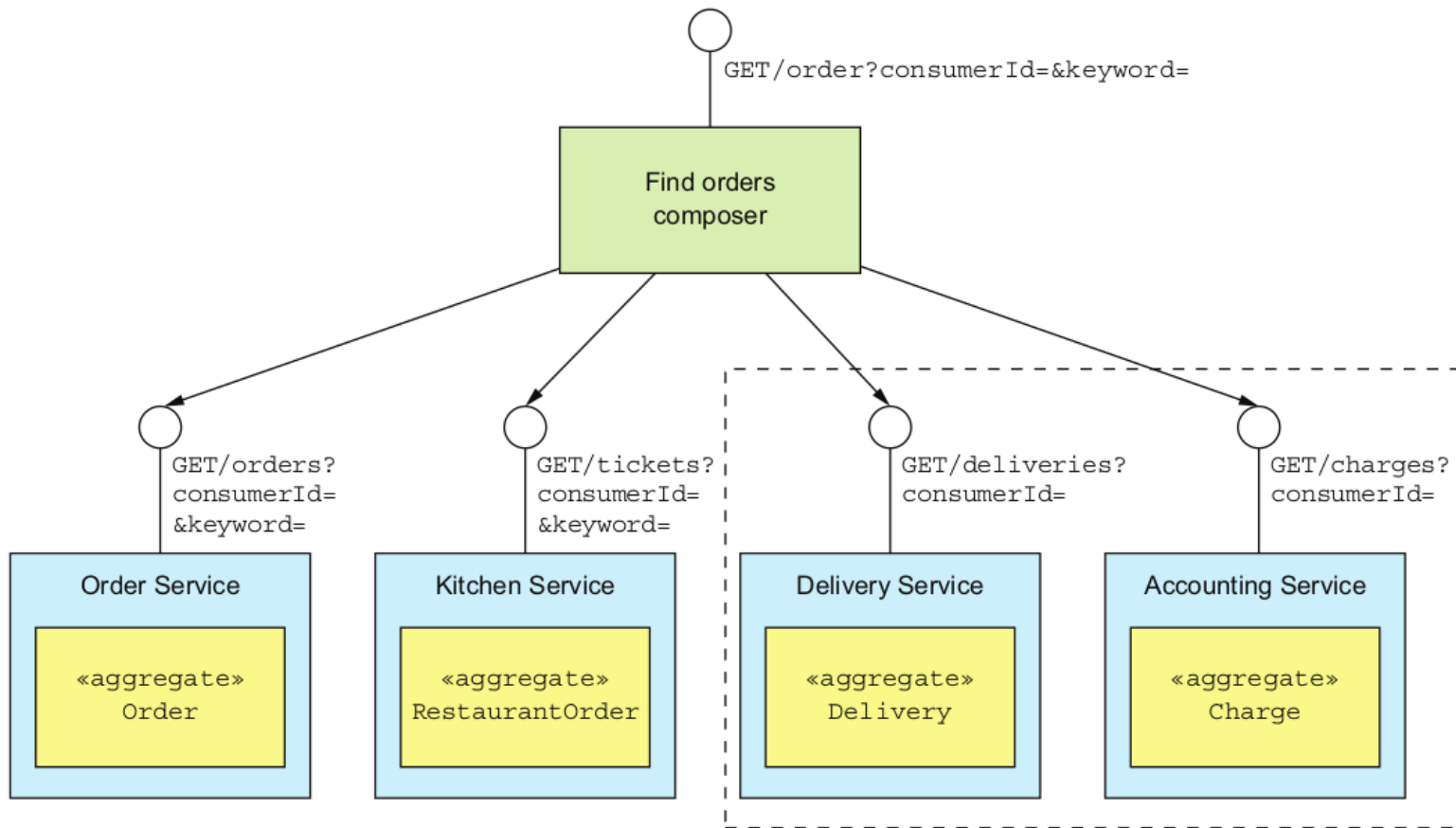
Motivation pour CQRS

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

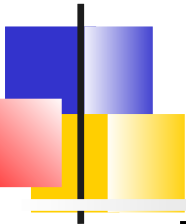
- Par ex : Rechercher les commandes depuis une certaine date dont le restaurant associé ou les entrées du menu correspond au mot clé «mafé »

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

Problème 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.



Autre 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 requêtes 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

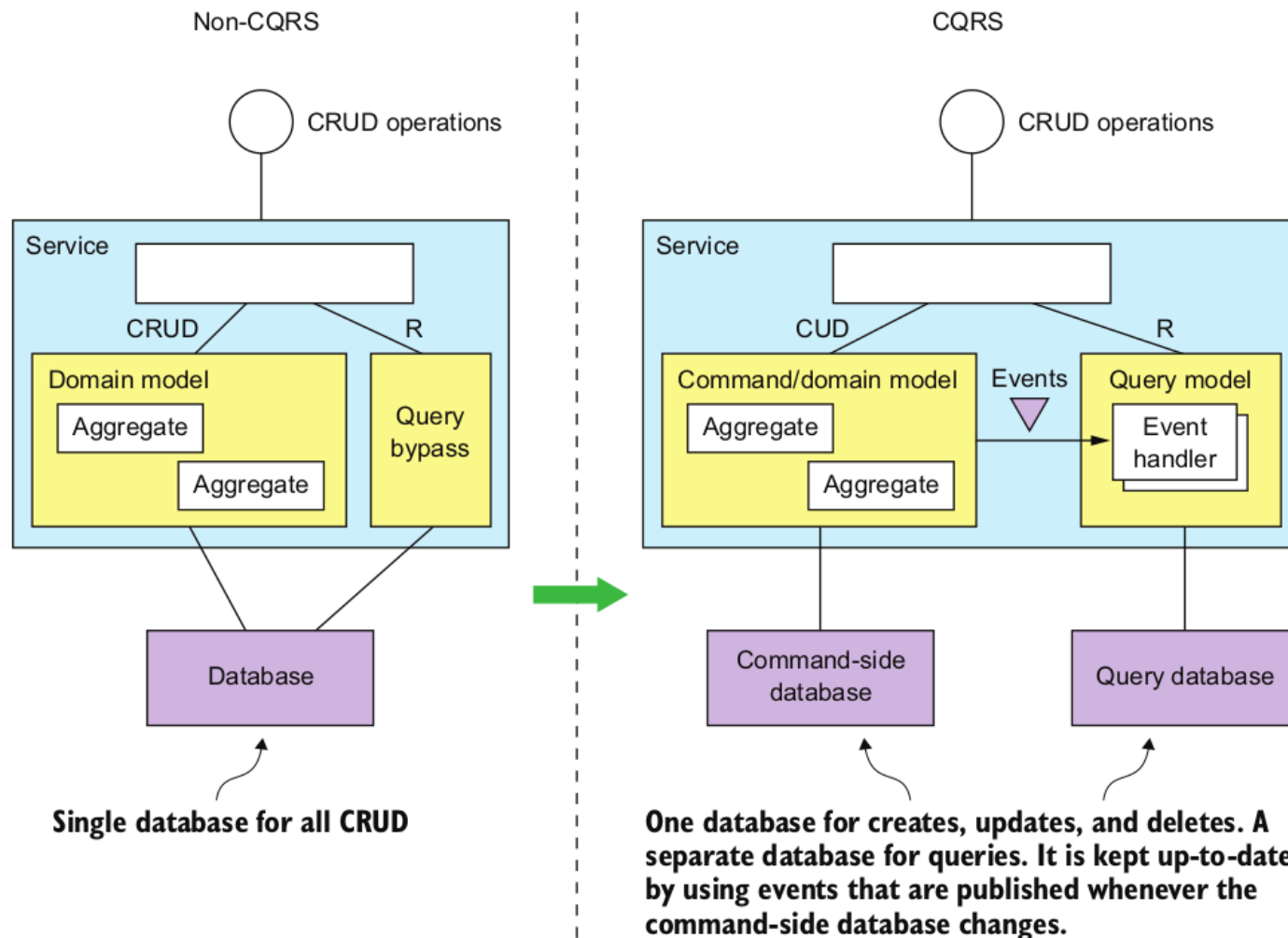
Dans une architecture basée sur CQRS, le modèle de domaine côté commande

- 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





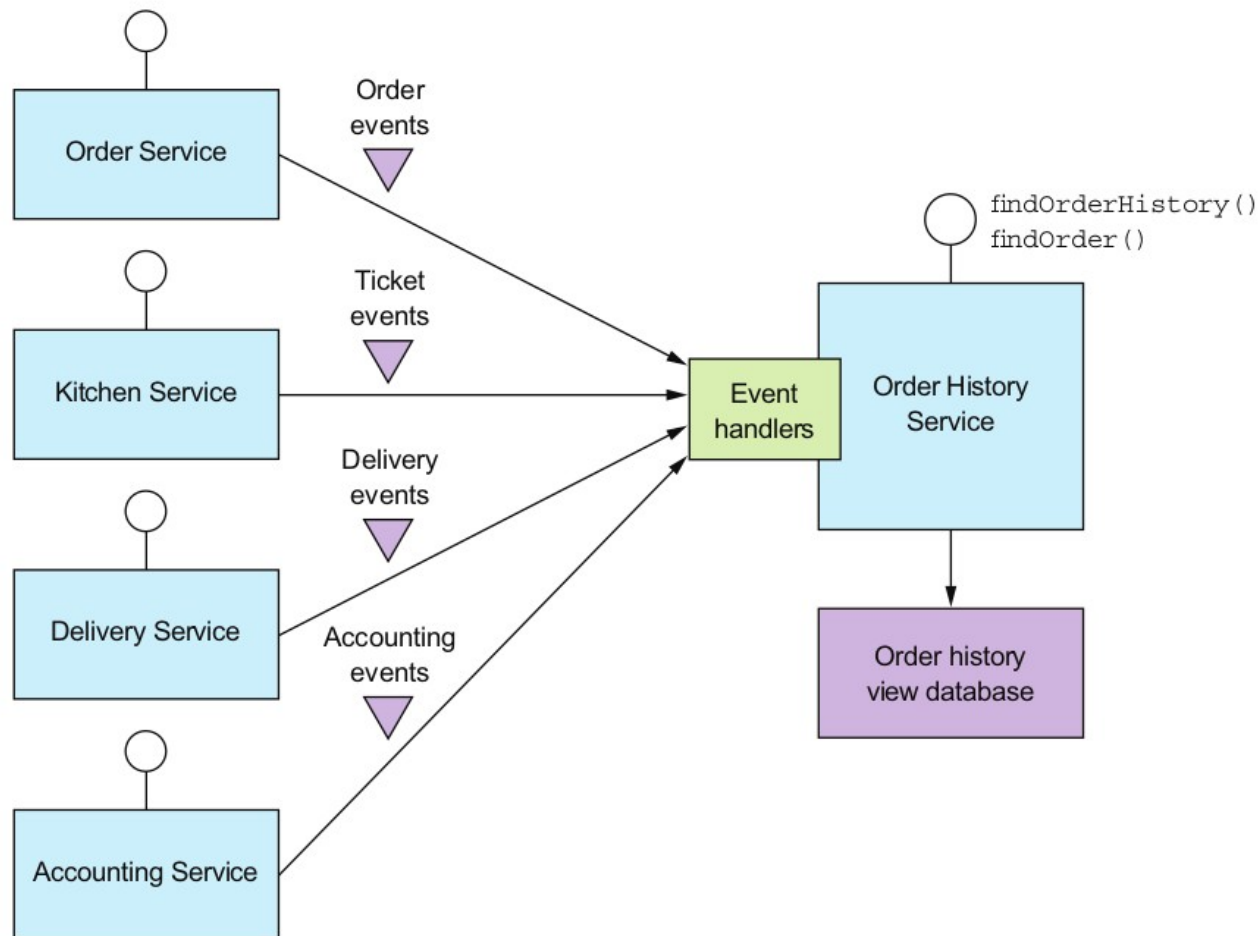
Service *Query-only*

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.

Example





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

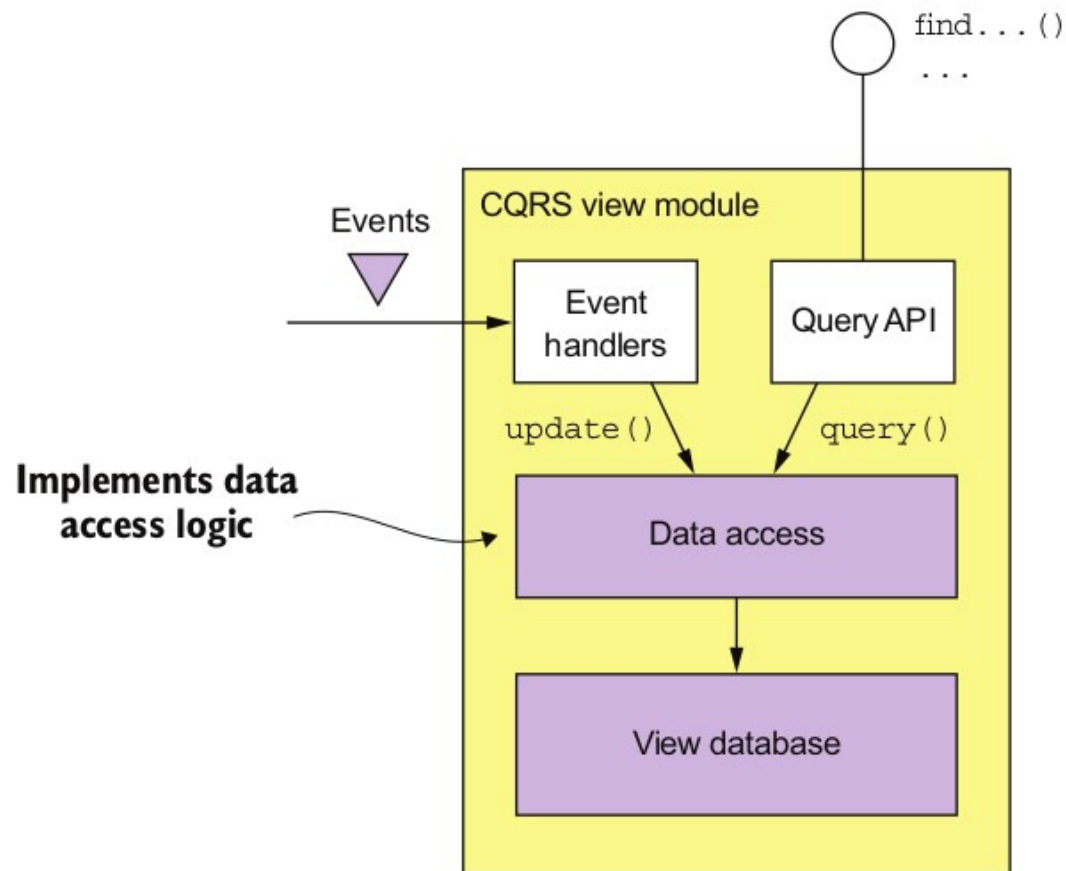
Architecture plus complexe

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 d'un *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



Évolutivité

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 APIs 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 dû au fait que les clients connaissent 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

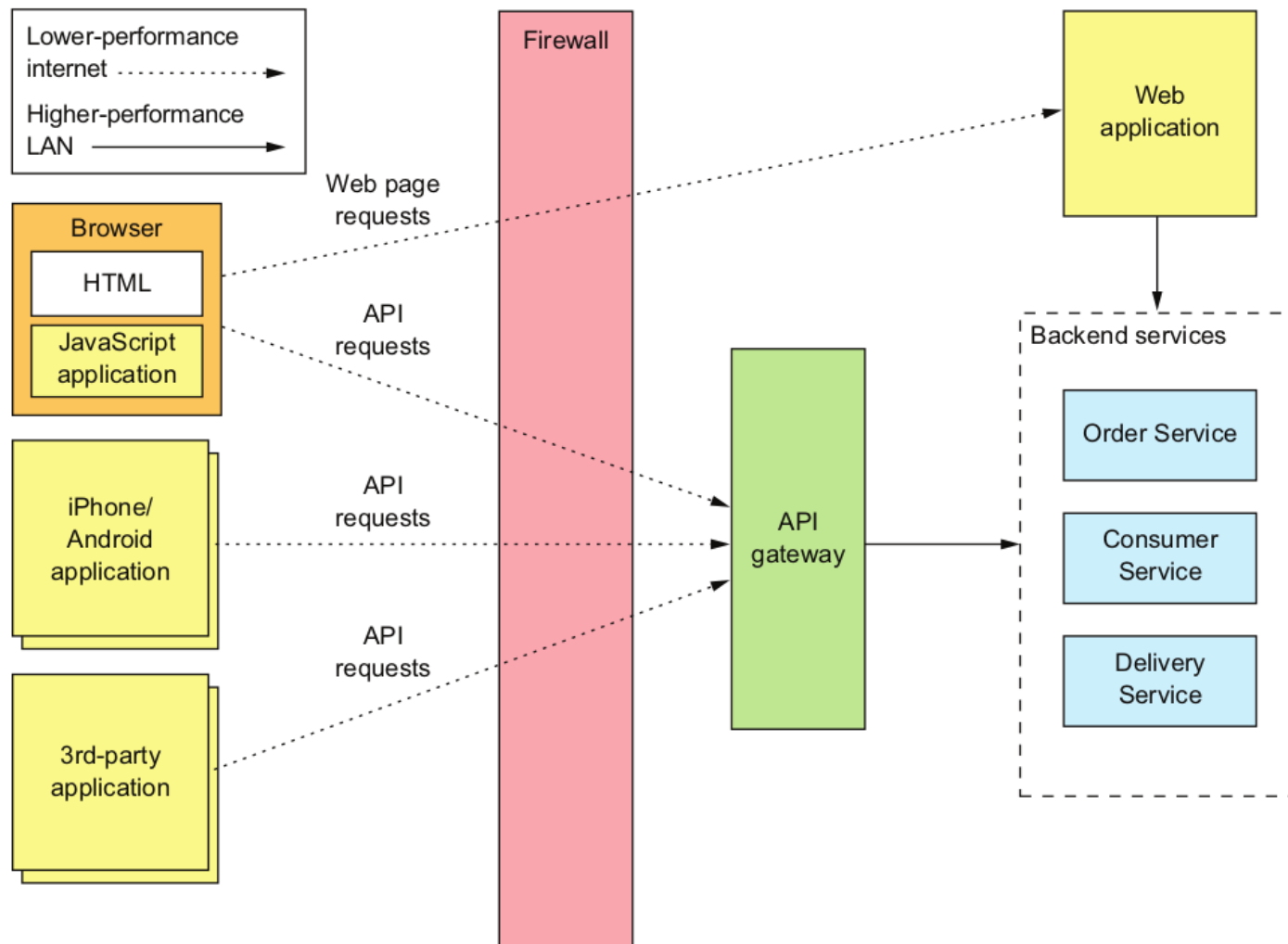
API gateway Pattern¹ : Implémente un service qui est le point d'entrée de l'application micro-service pour les clients externes

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

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

API Gateway





Fonctions de la Gateway

Routage : En fonction d'une 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 requêtes)

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

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

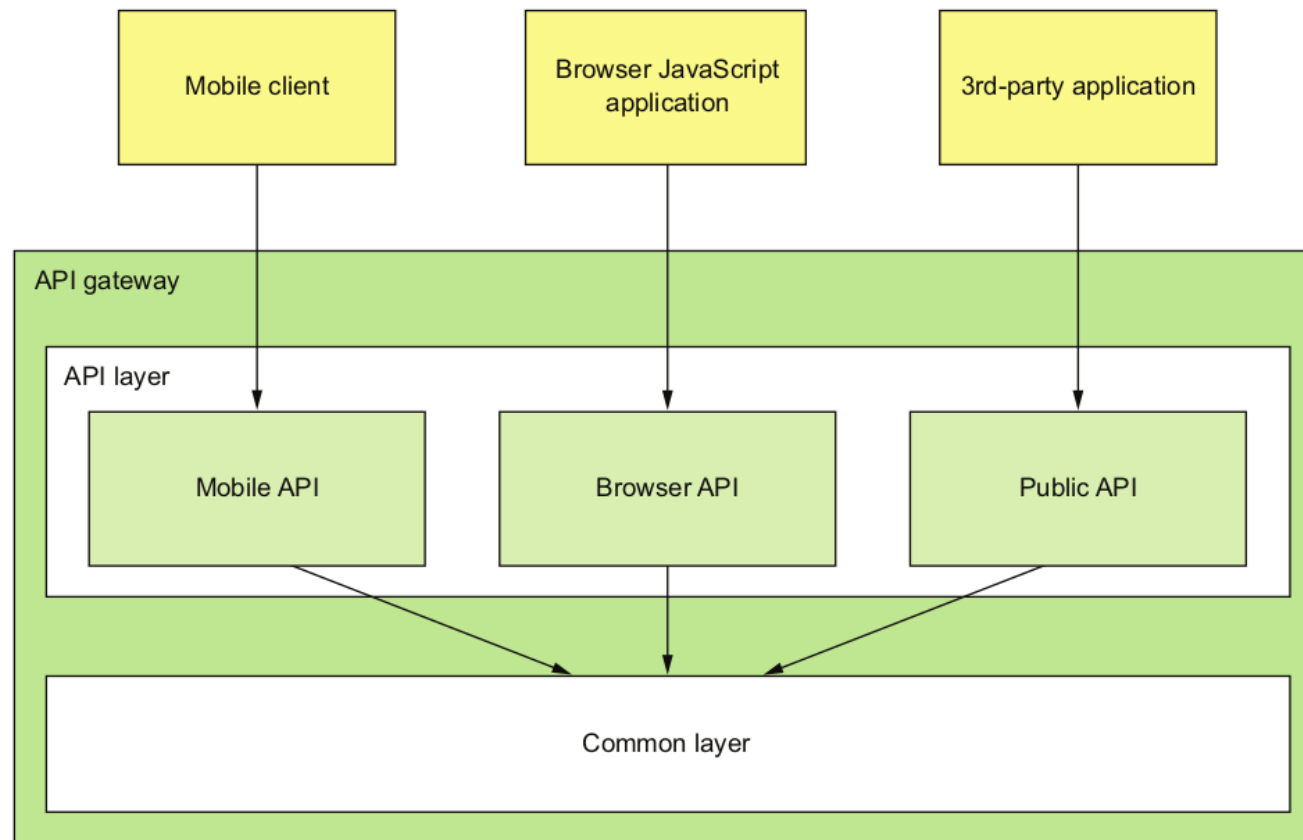
API spécifique par clients : La gateway n'offre pas la même API pour un mobile que pour les partenaires externes

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

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

Design de la Gateway

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





Organisation

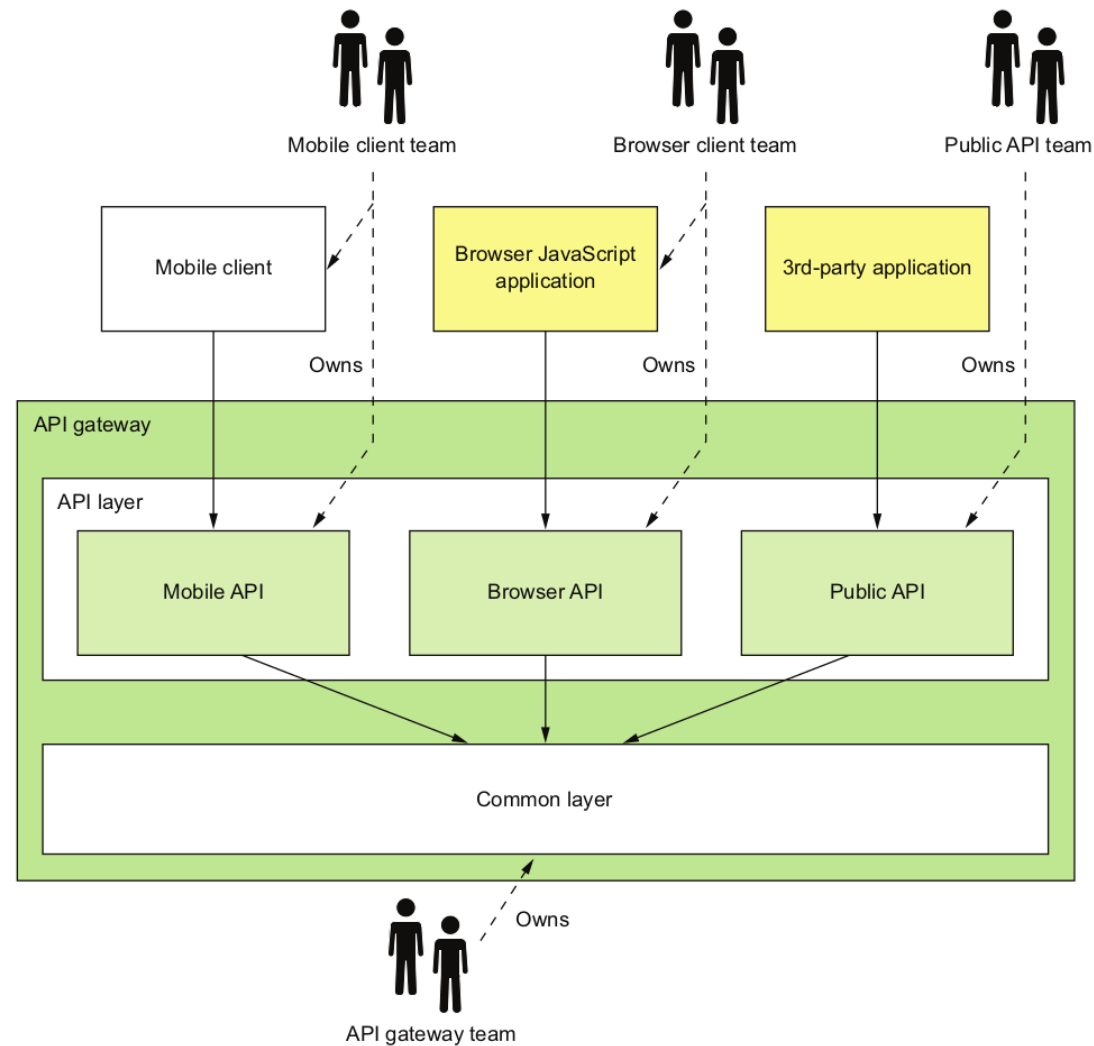
Qui est responsable du développement et de l'exploitation de la gateway ?

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

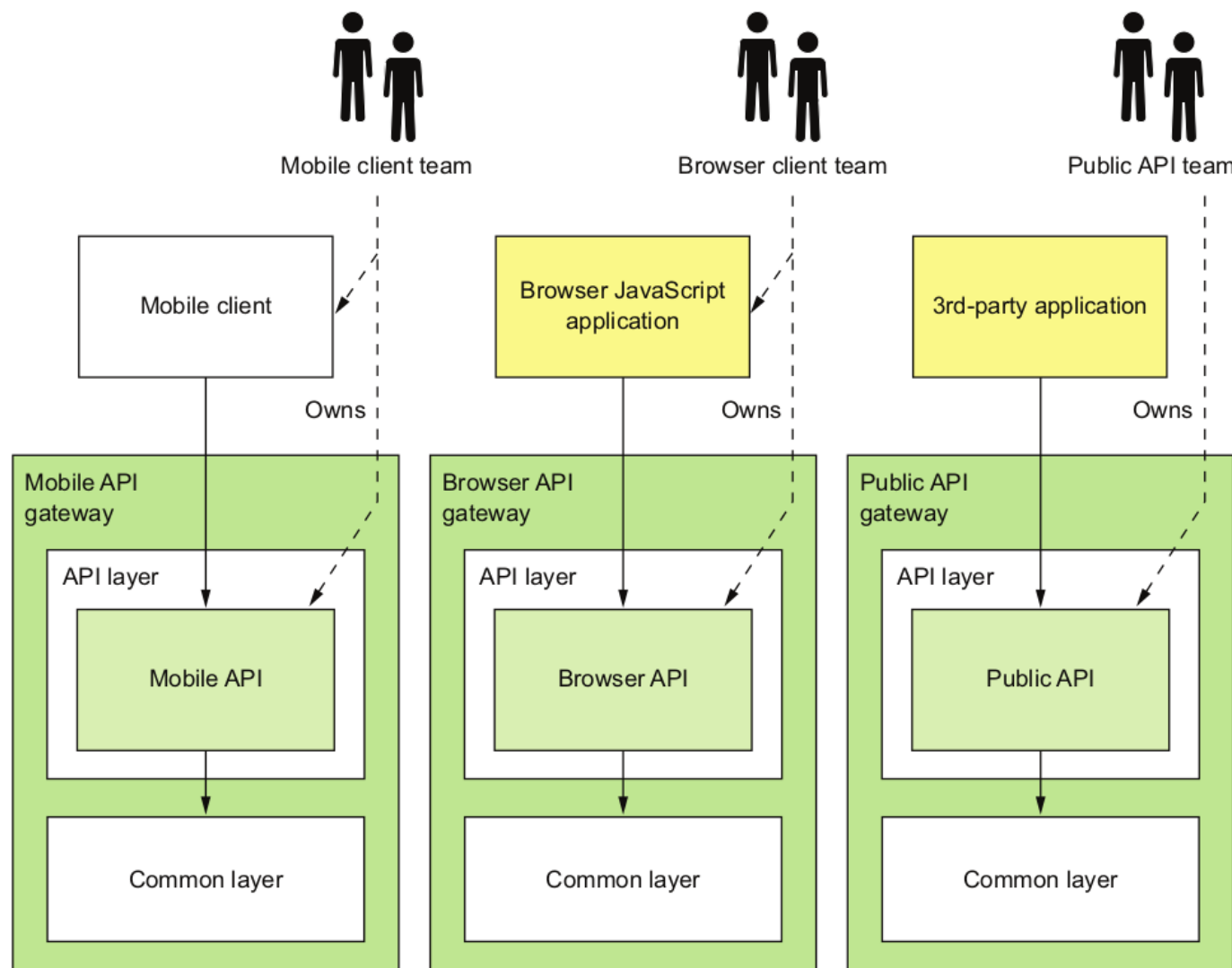
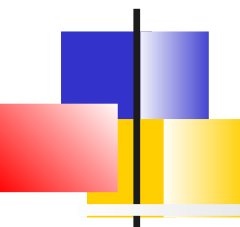




Backends for frontends Pattern

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

=> **Backends for frontends Pattern**¹ :
Implémenter une gateway différente pour chaque type de client.





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, appliquer 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



Sécurité

Patterns

Sécurisation via OAuth2
Propagation de Jeton
Client-credentials



Sécurité micro-services

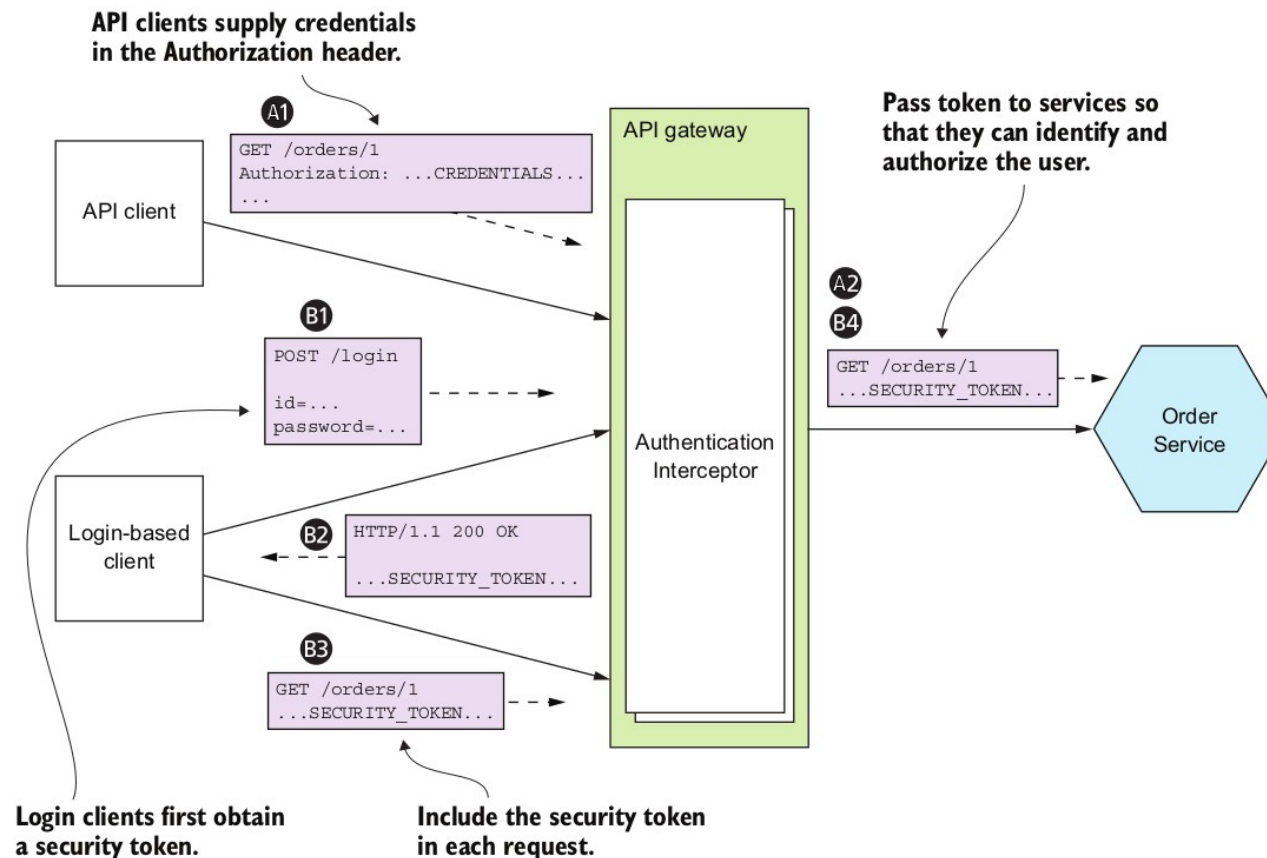
Plusieurs approches pour sécuriser une architecture micro-services :

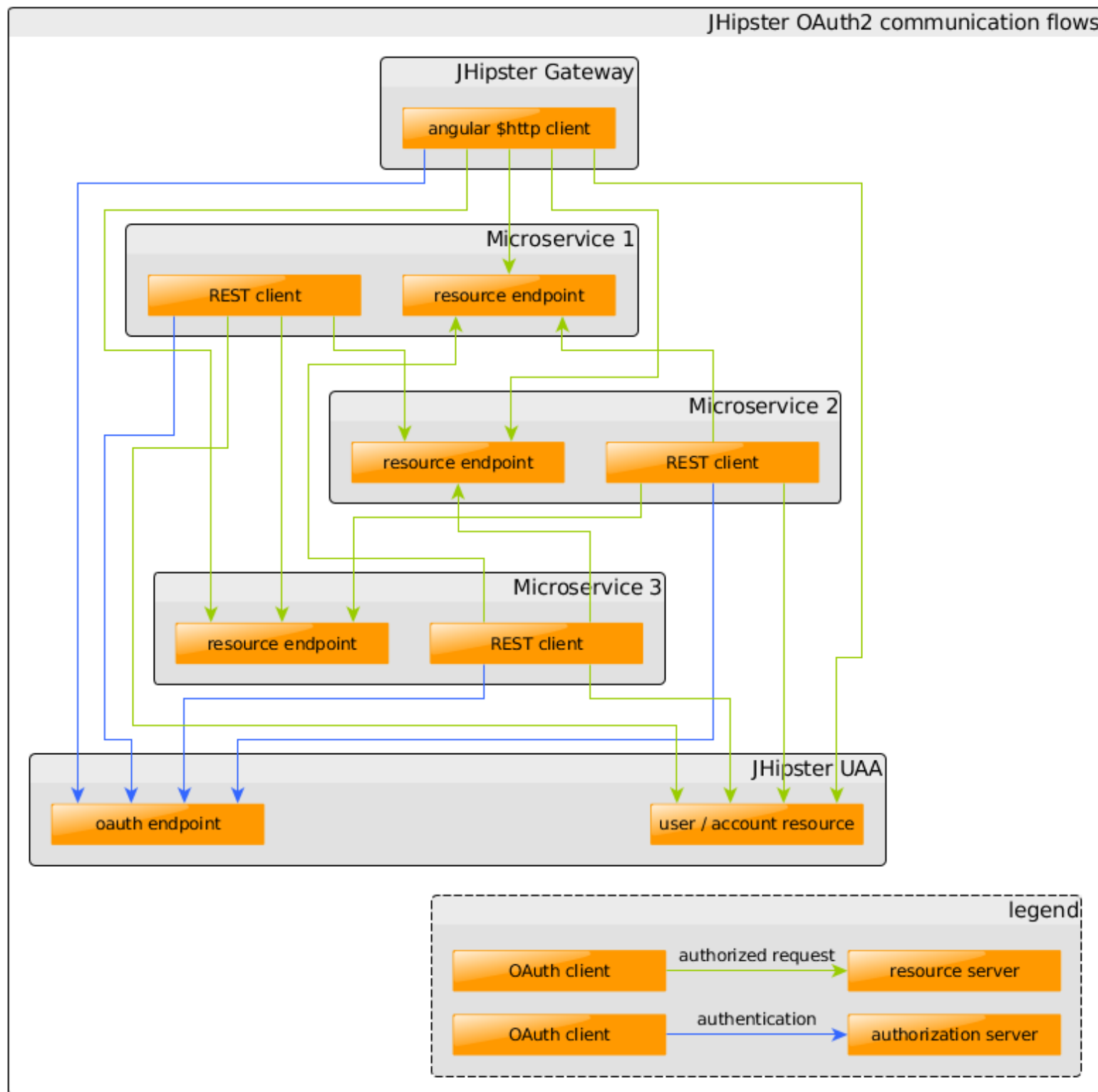
- N'implémenter la sécurité qu'au niveau de la gateway proxy. Les micro-services back-end ne sont protégés que par le firewall
- **Access token Pattern**¹ : La gateway passe un jeton contenant l'information sur le user (identité et rôles) Chaque micro-service a alors sa propre politique de sécurité.
- 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

L'API gateway est le point d'entrée unique pour les demandes des clients. Il authentifie les requêtes et les transmet à d'autres services, qui peuvent à leur tour appeler d'autres services.







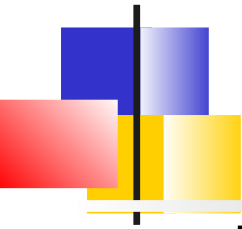
Sécurité

Patterns

Sécurisation via OAuth2

Propagation de Jeton

Client-credentials



Problématique

Le serveur de ressource doit :

- Extraire le jeton de la requête
- Le valider pour s'assurer qu'il a bien été produit par le serveur d'autorisation
- Extraire les informations et les transformer dans son modèle de sécurité
- Appliquer les ACLs



Réponse Spring Security

Le starter oauth2-resource server permet de :

- Configurer le filtre qui extrait le token
- Valider un jeton JWT grâce à la clé publique du serveur d'autorisation
- Extraire les informations et les transformer en `GrantedAuthority` en s'appuyant sur un bean *`JwtAuthenticationConverter`*
- Définir les ACLs programmatiquement ou déclarativement



SecurityFilterChain

@Bean

```
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {  
    return http.authorizeHttpRequests(acl ->  
        acl.requestMatchers("/actuator/**").permitAll().anyRequest().hasAnyRole("CLIENT"))  
        .oauth2ResourceServer(r -> r.jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()))  
            .csrf(csrf -> csrf.disable())  
            .build());  
}
```

@Bean

```
JwtAuthenticationConverter jwtAuthenticationConverter() {  
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter = new JwtGrantedAuthoritiesConverter();  
    grantedAuthoritiesConverter.setAuthoritiesClaimName("groups");  
    grantedAuthoritiesConverter.setAuthorityPrefix("ROLE_");  
  
    JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();  
    jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(grantedAuthoritiesConverter);  
    return jwtAuthenticationConverter;  
}
```



Configuration serveur d'autorisation

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8180/realms/store
```



Sécurité

Patterns
Sécurisation via OAuth2
Propagation de Jeton
Client-credentials



Problématique

Le serveur de ressource reçoit un jeton,
il le transfère lorsqu'il exécute des
requêtes vers des services en aval



Réponse Spring

WebClient (Reactive) : On peut appliquer le filtre *ServletBearerExchangeFilterFunction()* lors de la création du WebClient¹

RestTemplate (Web) : Pas de support, il faut soi-même implémenter l'intercepteur

Spring Cloud Gateway : Par défaut transfère les entêtes. Propose le TokenFilter pour ne transmettre que l'entête Bearer

1. <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/bearer-tokens.html>



Sécurité

Patterns
Sécurisation via OAuth2
Propagation de Jeton
Client-credentials



Problématique

Le micro-service reçoit une jeton mais ne l'utilise dans les services aval

Il utilise son propre jeton obtenu via un grant_type ***client_credentials***



Classes de Spring Security

ClientRegistration : représente un client enregistré avec OAuth 2.0. Encapsule les informations sur le client : *id*, *secret* et *scopes*. Configuré via :
`spring.security.oauth2.client.registration.[registrationId]`

OAuth2AuthorizedClient : représente un client autorisé. Il contient un *ClientRegistration* + *principalName*, *accessToken* et *refreshToken*

OAuth2AuthorizedClientManager : Gestionnaire qui contient la logique permettant de gérer le flux d'autorisation. Sa méthode `authorize(OAuth2AuthorizeRequest request)` effectue les demandes de jetons. Il est configuré via :
`spring.security.oauth2.client.provider.[provider name]`



Solutions Spring

Ajouter un filtre de *WebClient* ou un intercepteur *RestTemplate* qui obtient le jeton et le positionne dans l'entête.

```
RestTemplate rest = builder.rootUri("http://localhost:8080").build();

rest.getInterceptors().add((request, body, execution) -> {
    OAuth2AccessToken accessToken = tokenService.getToken();
    request.getHeaders().setBearerAuth(accessToken.getTokenValue());
    return execution.execute(request, body);
});
```



Obtention des tokens

```
public OAuth2AccessToken getToken() {  
    if (oAuth2AuthorizedClient == null) {  
        // On n'est pas encore authentifié  
        oAuth2AuthorizedClient = authorizedClientServiceManager.authorize(initialRequest());  
    } else if  
        (oAuth2AuthorizedClient.getAccessToken().getExpiresAt().minusSeconds(60).isBefore(Instant.now())) {  
        // Le token a expiré ou va bientôt expirer  
        oAuth2AuthorizedClient = authorizedClientServiceManager.authorize(initialRequest());  
    }  
    return oAuth2AuthorizedClient.getAccessToken();  
}
```



Configuration

```
spring:
  security:
    oauth2:
      client:
        provider:
          keycloak:
            token-uri: http://localhost:8180/realms/store/protocol/openid-connect/token
      registration:
        store:
          provider: keycloak
          client-id: order
          client-secret: secret
          client-authentication-method: client_secret_basic
          authorization-grant-type: client_credentials
          scope:
            - openid
            - service
```



Observabilité

Patterns
Health API
Metrics
Tracing distribué



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 parcours 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>



Observabilité

Patterns
Health API
Metrics
Tracing distribué



Endpoints

Actuator fournit de nombreux endpoints :

- **beans** : Une liste des beans Spring
- **env / configprops** : Liste des propriétés configurables
- **health** : Etat de santé de l'appli
- **info** : Informations arbitraires. En général, Commit, version
- **metrics** : Mesures
- **mappings** : Liste des mappings configurés
- **trace** : Trace des dernières requête HTTP
- **docs** : Documentation, exemple de requêtes et réponses
- **logfile** : Contenu du fichier de traces



Configuration

Les *endpoints* peuvent être configurés par des propriétés.

Chaque endpoint peut être

- Activé/désactivé
- Sécurisé par Spring Security
- Mappé sur une autre URL

Par défaut, seuls les endpoints */health* et */info* sont activés par défaut

Pour activer les autres :

- *management.endpoints.web.exposure.include=**
- Ou les lister un par un



Endpoint */health*

L'information fournie permet de déterminer le statut d'une application en production.

- Elle peut être utilisée par des outils de surveillance responsable d'alerter lorsque le système tombe (Kubernetes par exemple)

Par défaut, le endpoint affiche un statut global mais on peut configurer Spring pour que chaque sous-système (beans de type *HealthIndicator*) affiche son statut :

```
management.endpoint.health.show-details= always
```



Sondes Kubernetes et Actuator

Les applications déployées sur Kubernetes peuvent fournir des informations sur leur état interne avec les *Container Probes*

- ***livenessProbe***: Indique si le container s'exécute
- ***readinessProbe***: Indique si le container est prêt à répondre à des requêtes
- ***startupProbe***: Indique si l'application à l'intérieur du conteneur est démarré.

Actuator est capable d'exposer les informations "Liveness" et "Readiness" en http si il détecte un environnement Kubernetes ou via la propriété `management.endpoint.health.probes.enabled`



Support SpringBoot

Spring Boot supporte directement les états de disponibilité

- « **alive** » : Le contexte Spring a été chargé ou rechargé
- « **ready** » : Prêt à accepter du trafic

Ces indicateurs sont affichés via */actuator/health*. Ou directement via les sondes :

- */actuator/health/liveness*
- */actuator/health/readiness*



Observabilité

Patterns
Health API
Metrics
Tracing distribué



Observabilité avec SB3

Avec SB3, l'observabilité est basée sur
Micrometer et ***Micrometer Tracing***
(Anciennement Spring Cloud Sleuth)

Actuator configure automatiquement les objets
nécessaires à Micrometer, en particulier
ObservationRegistry et *ObservationHandler*

Les métriques relatifs aux starters de
l'application sont automatiquement
disponibles



Métriques Micrometer

Micrometer supporte 4 principaux types de métriques :

- **Counter** : Compte le nombre d'occurrence d'un évènement
- **Timer** : mesure le temps d'une exécution et fournit des statistiques telles que la moyenne, le maximum et le minimum.
- **Gauge** : Valeur instantanée
- **DistributionSummary** : Distribution d'évènements



Métriques Custom

```
@RestController
@RequestMapping("/api")
public class NewsController {

    private Counter counter;

    public NewsController(MeterRegistry registry) {
        // counter
        this.counter = Counter.builder("news_fetch_request_total").
            tag("version", "v1").
            description("News Fetch Count").
            register(registry);
    }

    @GetMapping("/news")
    public List<News> getNews() {
        counter.increment();
        return List.of(new News("Good News!"), new News("Bad News!"));
    }
}
```




Observabilité

Patterns
Health API
Metrics
Tracing distribué



Tracing avec SB3

L'ancien projet *Spring Cloud Sleuth* a été transféré vers Micrometer.

Les dépendances nécessaires sont désormais :

- *io.micrometer : micrometer-tracing*
- Une dépendance sur une implémentation (Brave ou OpenTelemetry) :
 - *micrometer-tracing-bridge-brave*
 - *micrometer-tracing-bridge-otel*

Actuator fournit une auto-configuration pour les deux traceurs

Il suffit donc de mettre à disposition un agrégateur de trace comme Zipkin



Annexes

Event Sourcing Pattern

Tests

Déploiement



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 ?

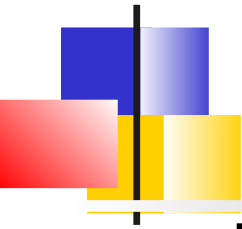
Pourrait-on avoir la garantie que dès qu'un agrégat est créé ou mis à jour, un événement est publié ?



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

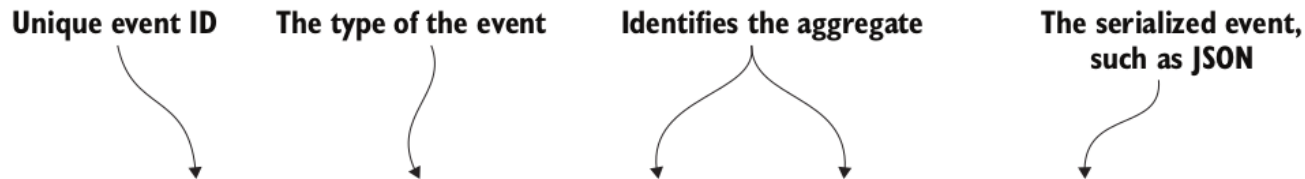
Inconvénients

- 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

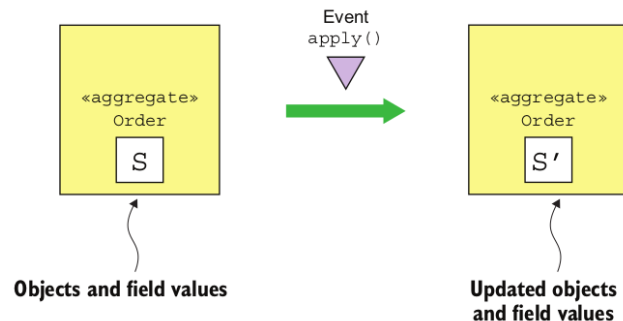
En programmation fonctionnelle, c'est une fonction ***reduce***

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 n'y ait ou pas des consommateurs

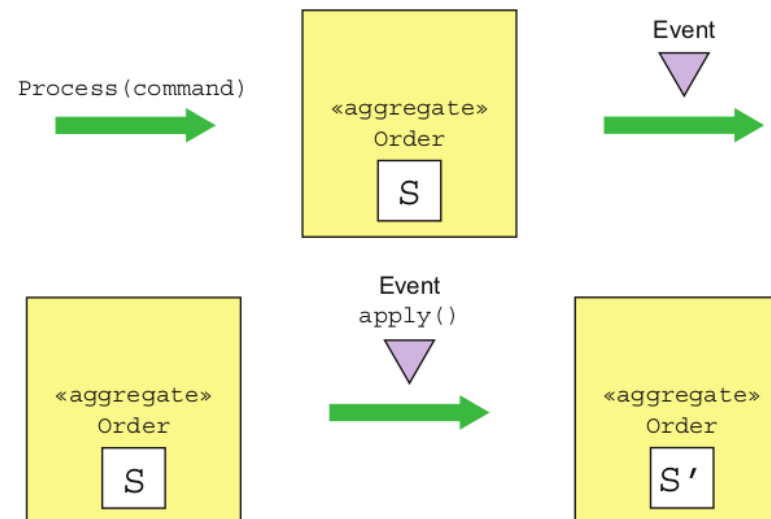
L'événement doit contenir toutes les données que l'agrégat nécessite pour faire sa transition d'état.
=> Certains événements auront peu de données (changement d'un statut) , d'autres beaucoup :
Ex. Création initiale de l'agrégat



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*)



Example

```
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.isGreaterThanOrEqual(orderMinimum)) {  
                    throw new OrderMinimumNotMetException();  
                }  
                return singletonList(  
                    new OrderRevisionProposed(  
                        orderRevision, change.currentOrderTotal,  
                        change.newOrderTotal));  
  
            default:  
                throw new UnsupportedOperationException(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.

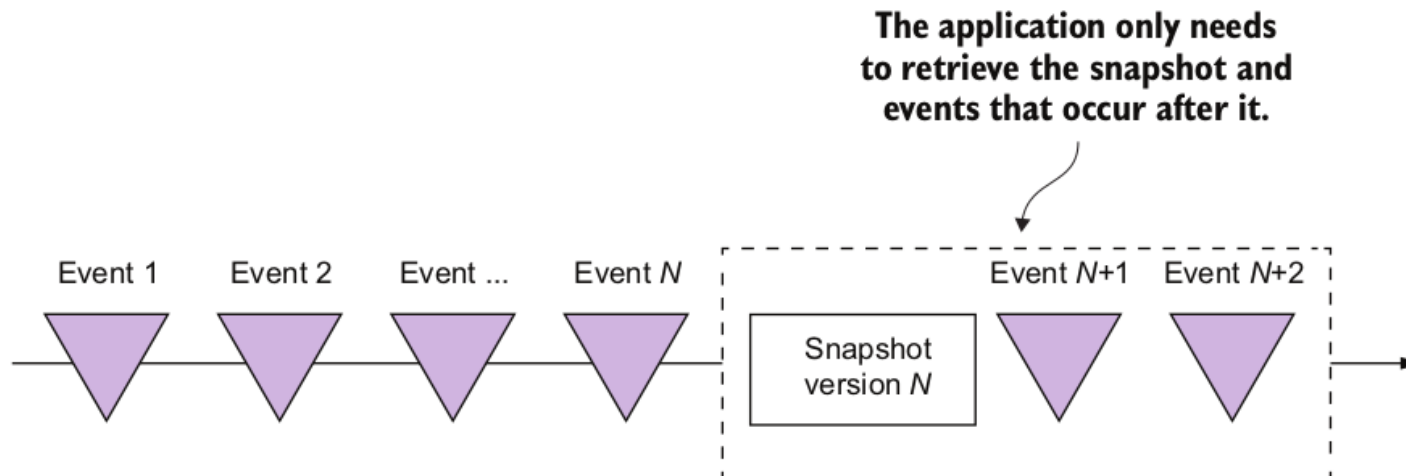
- Lors de l'ajout d'évènement, le store vérifie que l'agrégat n'a pas été modifié entre temps via un n° de version.
- Le n° de version peut tout simplement être le nombre d'événements liés à l'agrégat.




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***.





Évolutivité, *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*

1.<https://gdpr-info.eu/art-17-gdpr/>



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



Annexes

Tests
Déploiement

Pyramide des tests

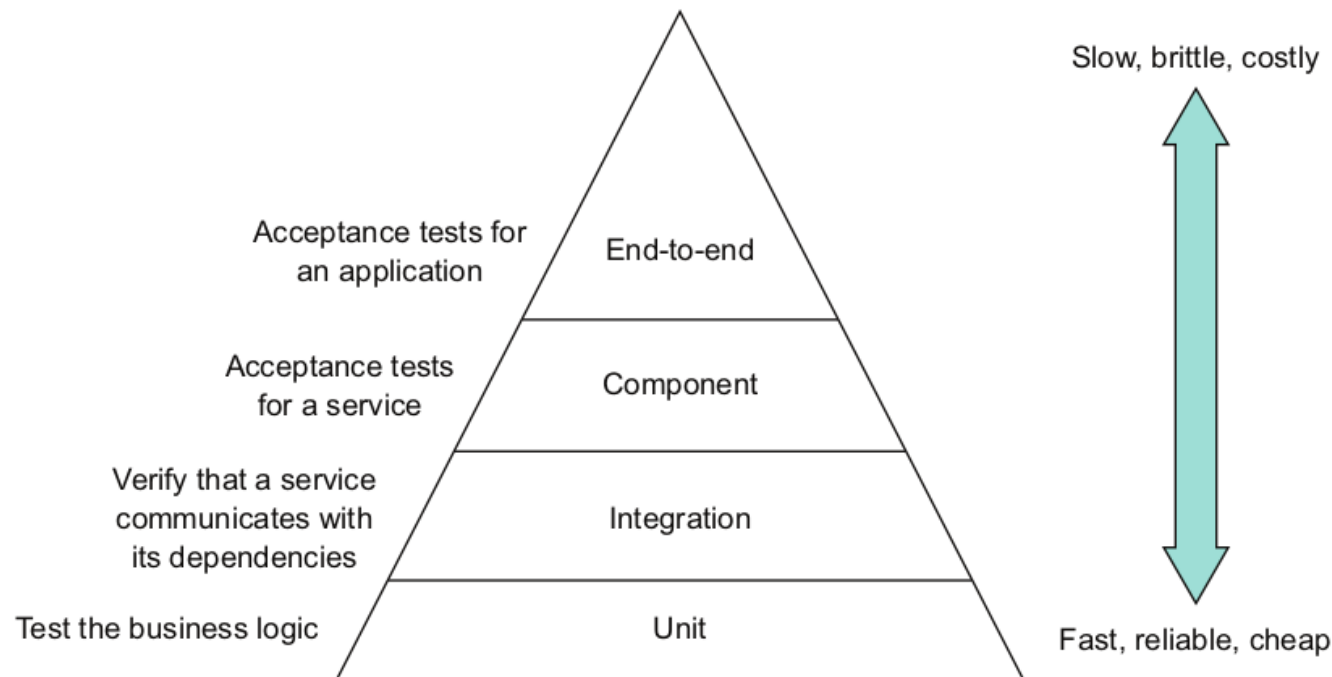


Figure 9.5 The test pyramid describes the relative proportions of each type of test that you need to write. As you move up the pyramid, you should write fewer and fewer tests.



Tests de composant

Les tests de composants sont les tests d'acceptation d'un service, i.e. le service remplit son contrat.

Ce sont donc 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



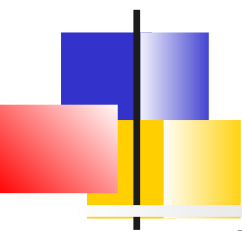
Consumer Driven Contract Test

Consumer-driven contract test Pattern¹ : Vérifier que la « forme » de l'API d'un service 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 un consommateur via des **exemples**¹, i.e les contrats

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



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 les tests côté consommateur

Process

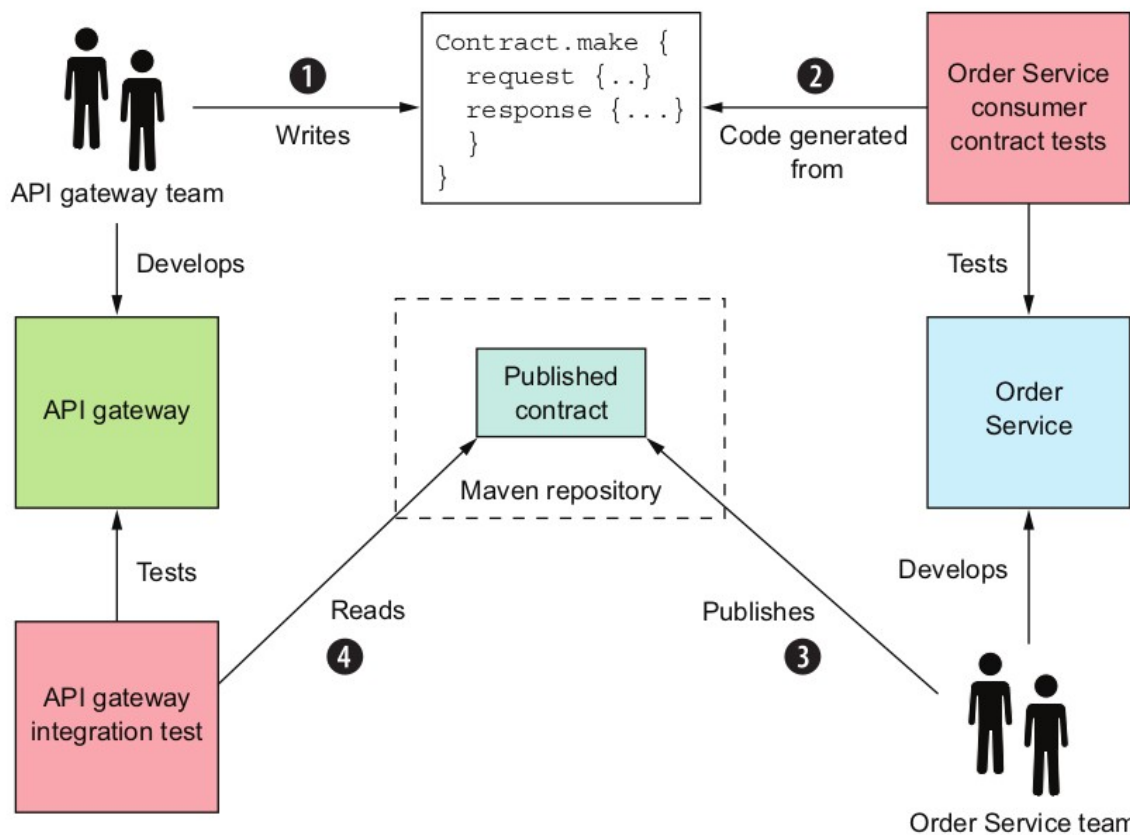


Figure 9.8 The API Gateway team writes the contracts. The Order Service team uses those contracts to test Order Service and publishes them to a repository. The API Gateway team uses the published contracts to test API Gateway.



Example 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
    }
}
```



Exemple : Génération des tests d'acceptation côté producteur

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



Utilisation du mock serveur côté 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

La spécification inclut le nom du canal de communication et la structure des messages.

Le contrat spécifie des exemples de message



Contrat

```
org.springframework.cloud.contract.spec.Contract.make {
    label 'orderCreatedEvent'
    input {
        triggeredBy('orderCreated()')
    }
    outputMessage {
        sentTo('order-channel')
        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','orderservice.domain.Order')
            header('event-aggregate-id','1')
        }
    }
}
```



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 :

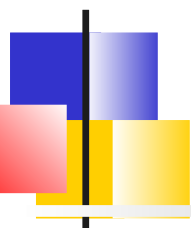
- Utiliser des BD/Message Broker in-memory
Attention on n'est plus dans les conditions de production
- Utiliser des images docker pour démarrer les services nécessaires
Plus lourd à mettre en place

Pour mocker les services dépendants, on peut utiliser les approches *Consumer Driven Contract* ou directement *WireMock*



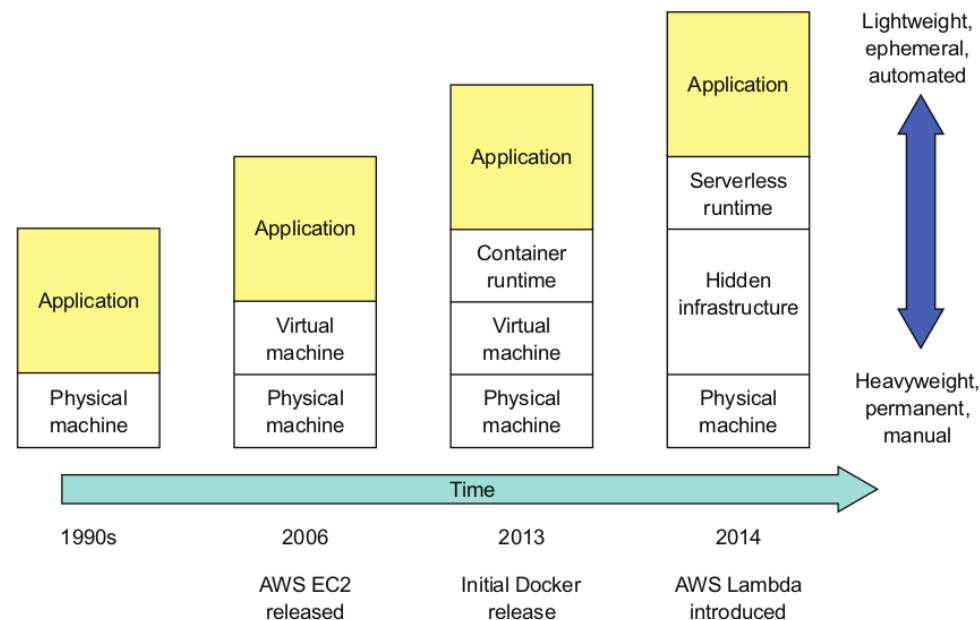
Annexes

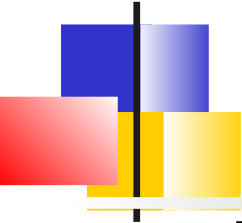
Tests
Déploiement



Infrastructure de déploiement

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



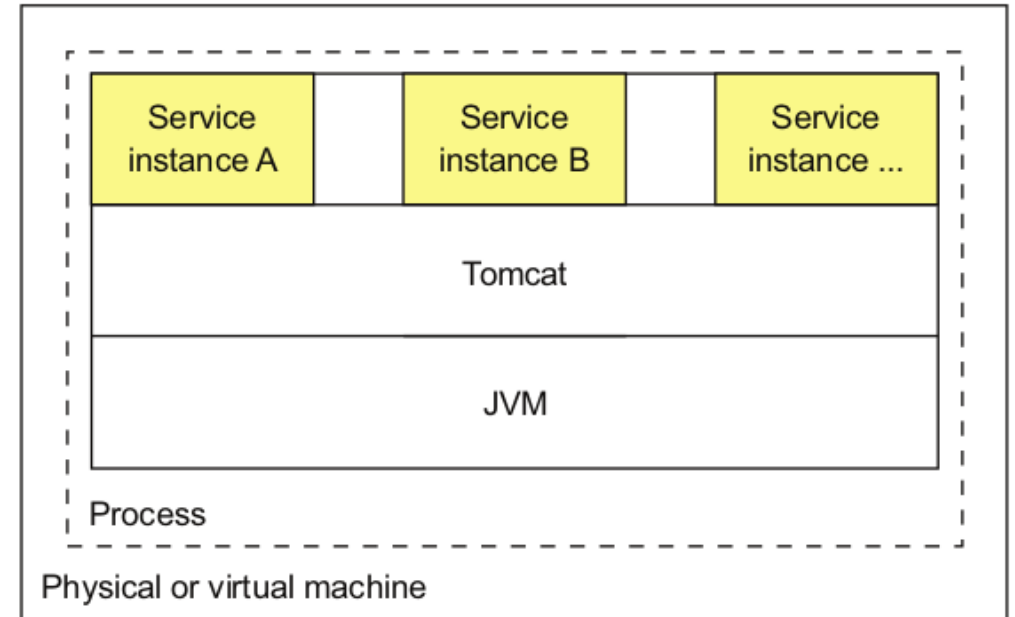
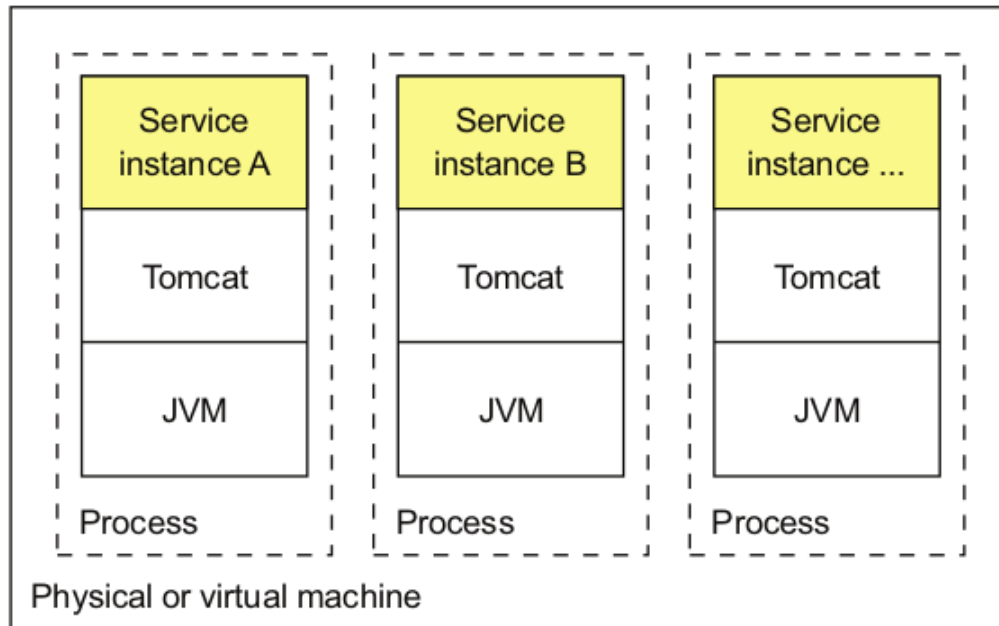


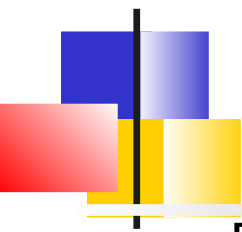
Format de packaging spécifique au langage

Une alternative est de packager les services dans un format spécifique au langage (ex : .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)

Examples





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

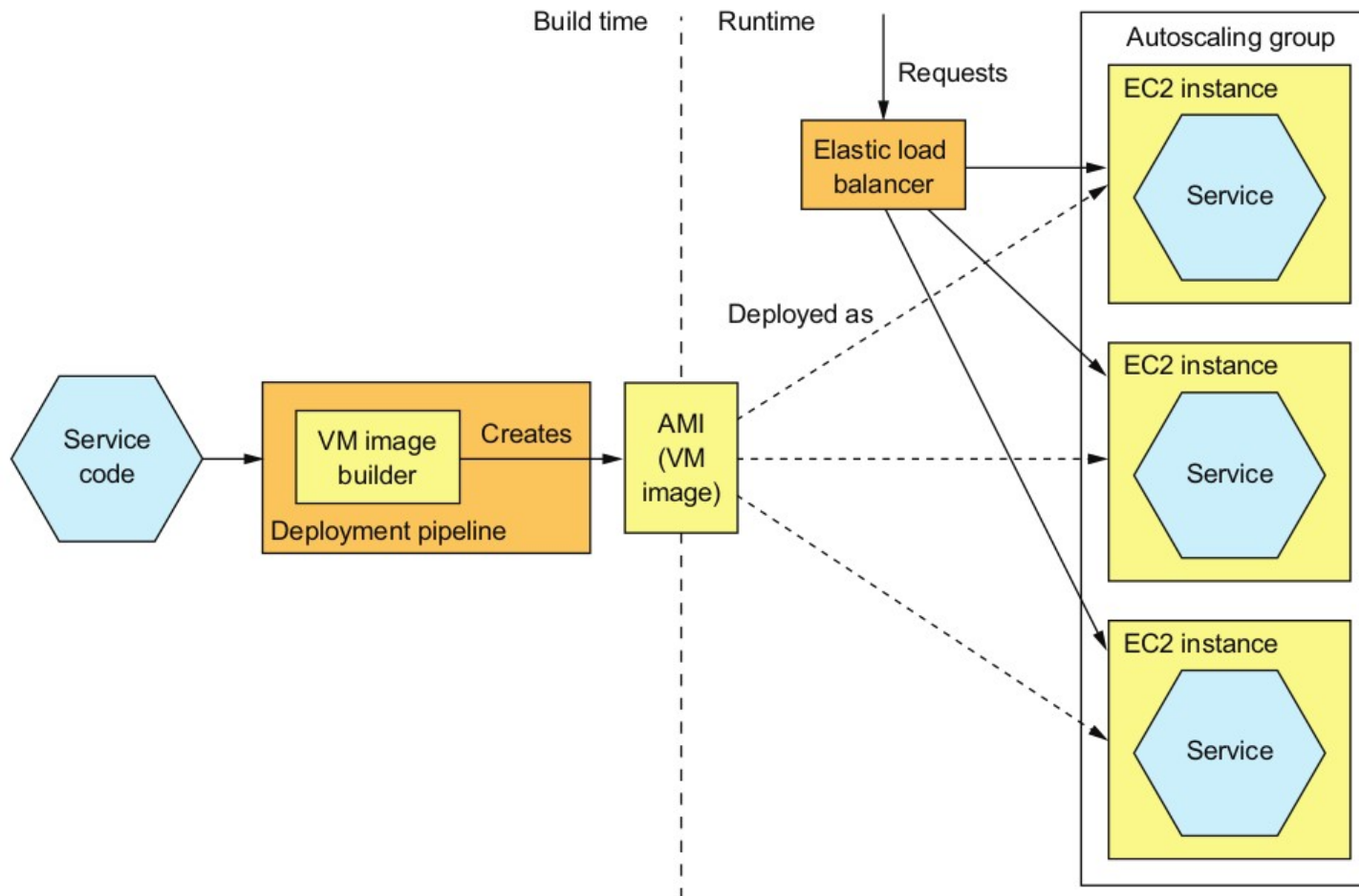
Deploy a service as a VM Pattern¹ :

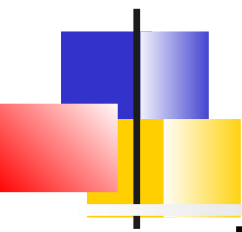
Déployer les services packagés comme des images VM. Chaque service est une VM

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

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

Example





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

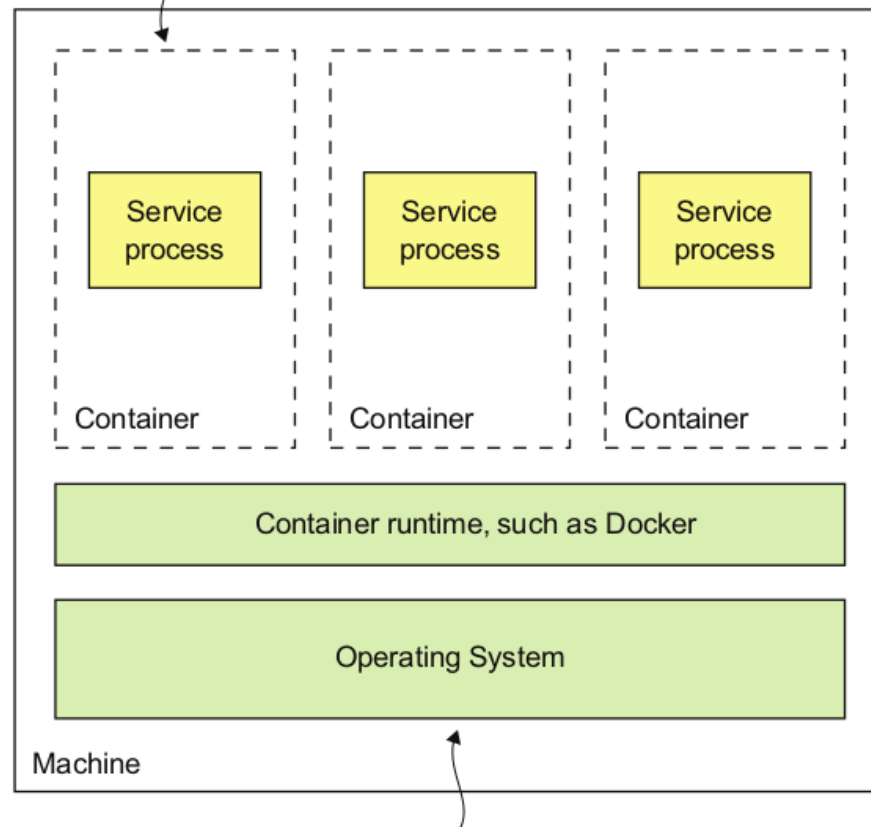
Pattern¹ : Déployé les services packagés comme des images de conteneur. Chaque service est un conteneur

Le packaging en image fait partie de la pipeline de déploiement

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

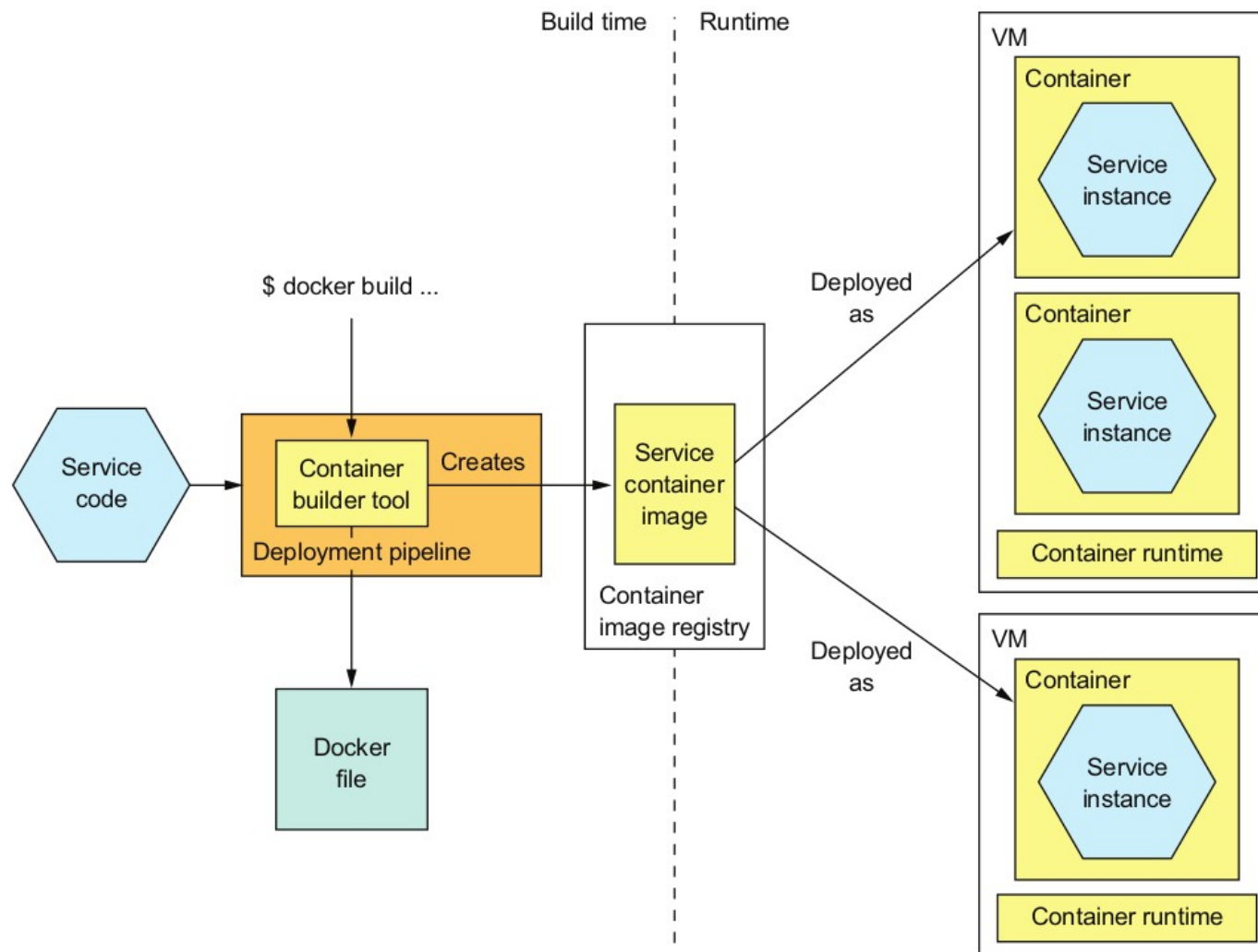
Exécution

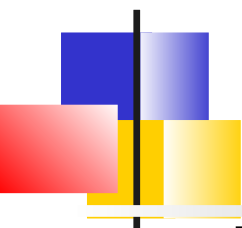
Each container is a sandbox that isolates the processes.



Shared by all of the containers

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

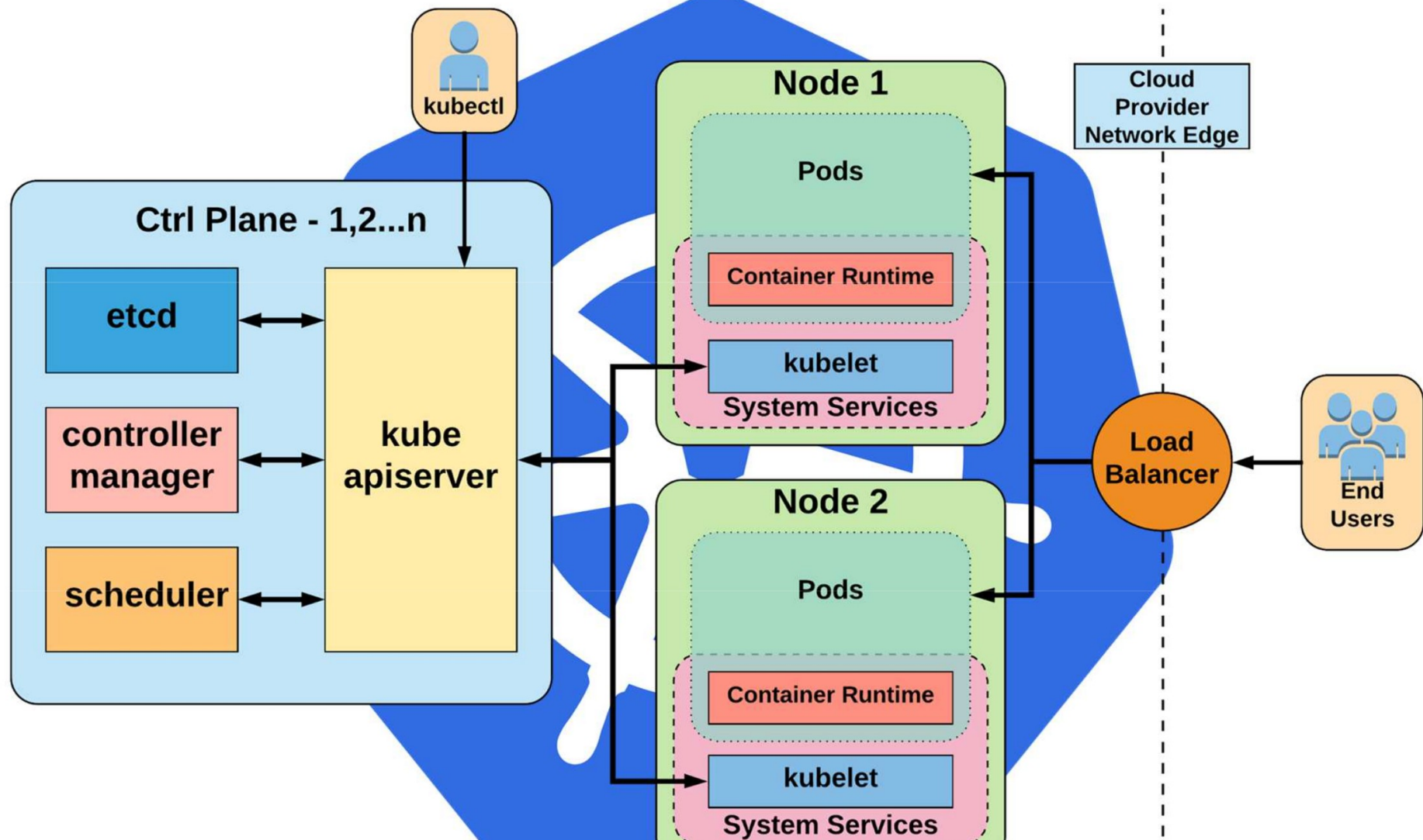


Fonctionnalités applicatives d'un orchestrateur de conteneur

- Méthodes natives pour la découverte de services et la répartition de charge
- Déploiements Blue/Green
- Scaling automatique
- Gestion d'application Stateless et Stateful
- Démarrage de jobs planifiés
- Stockage des valeurs de configuration et des secrets
- Sécurité

pod = 1 ou plusieurs conteneurs co-localisés

Architecture cluster





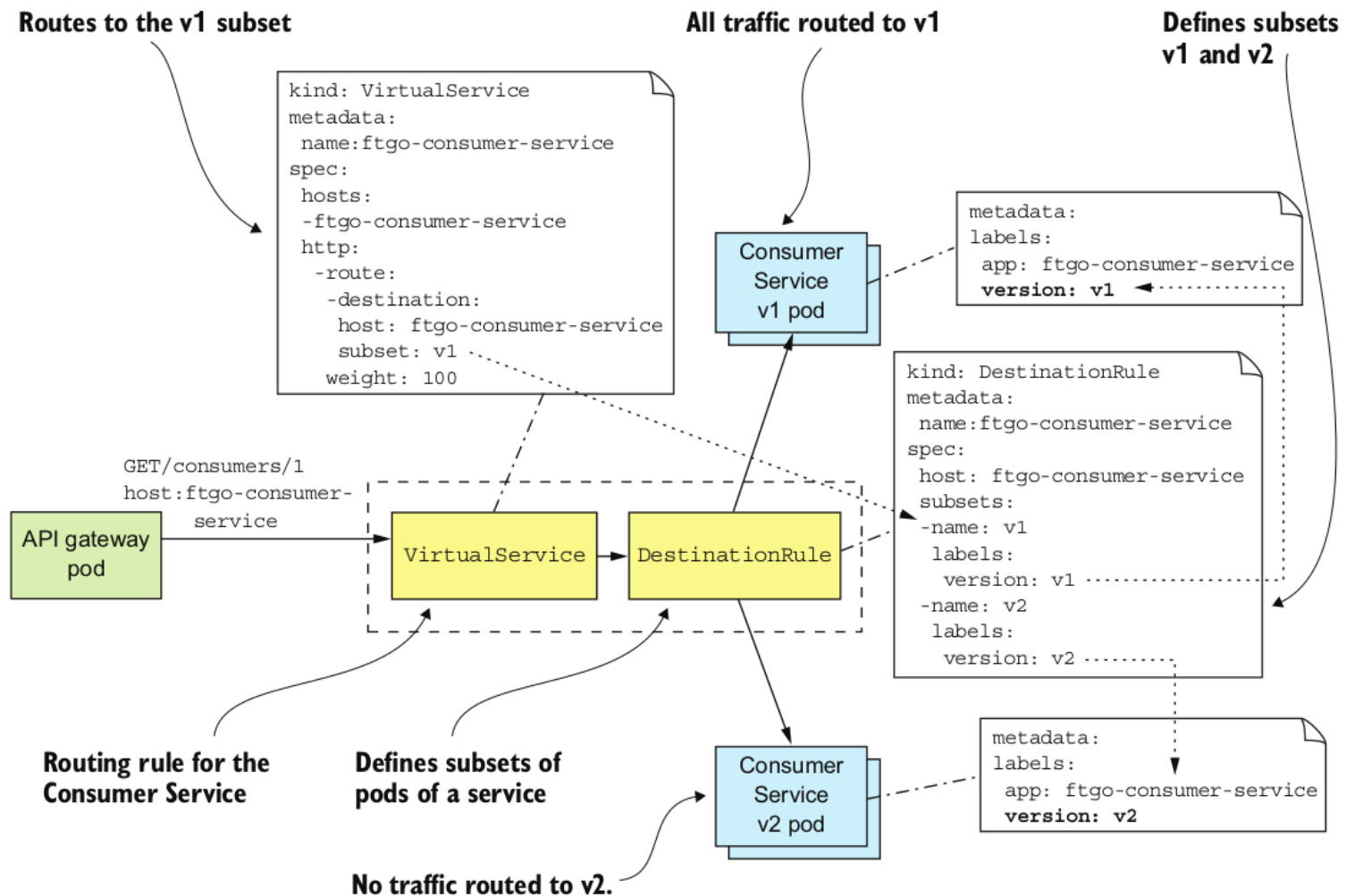
Service Mesh

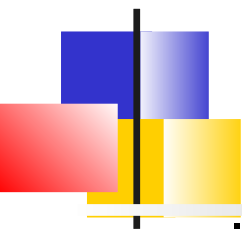
Service Mesh Pattern¹ : Acheminer tout le trafic réseau entrant et sortant des services via une couche réseau qui implémentent diverses services techniques, comme les circuit-breaker, le traçage distribué, la découverte, l'équilibrage de charge et le routage du trafic basé sur des règles

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>

Routage vers 2 versions





Istio

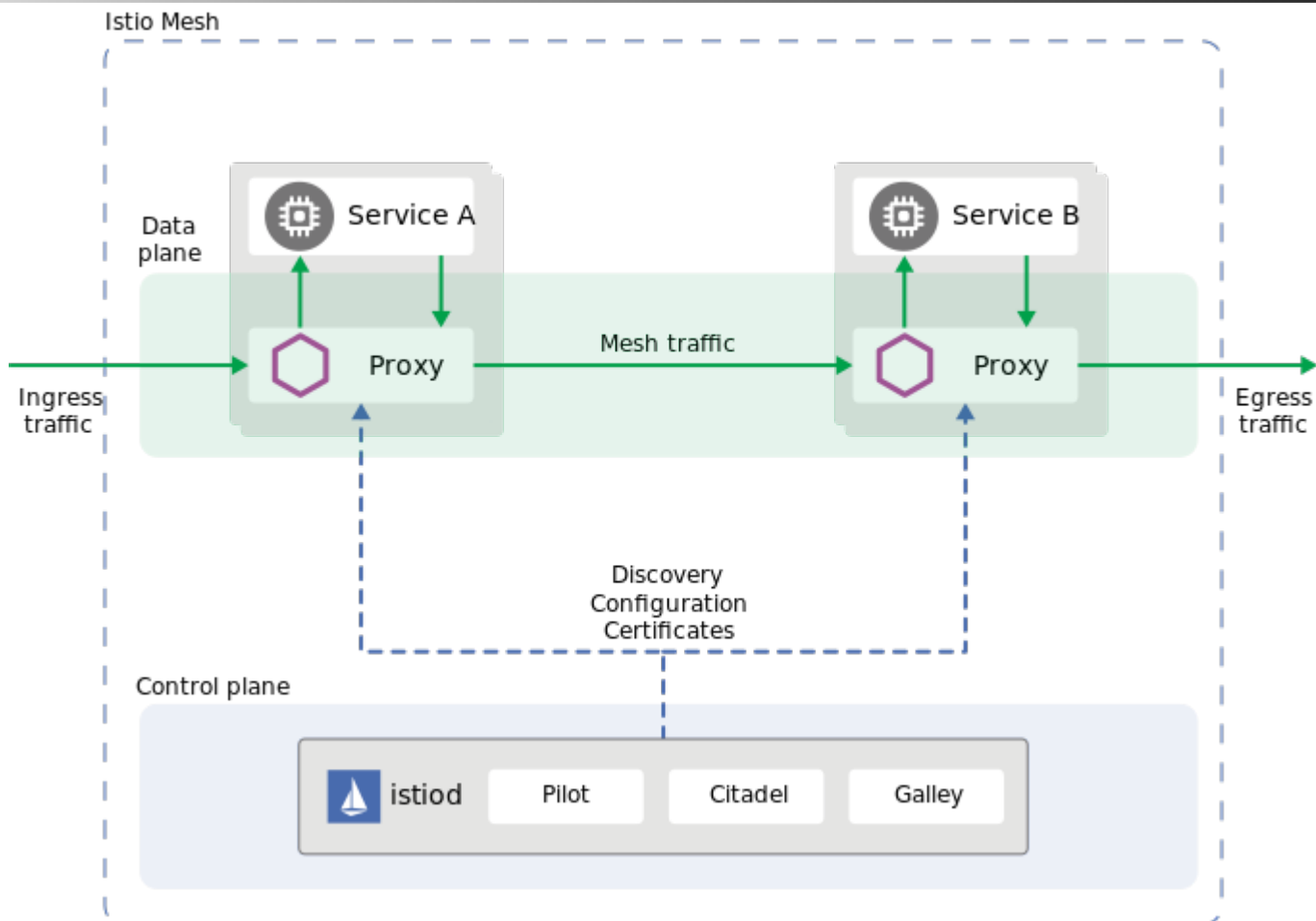
Istio est un ***service mesh*** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

- 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 politique 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 fournis

