



Micro-services, Spring Cloud, GraphQL

David THIBAU – 2021

david.thibau@gmail.com



Agenda

- **Introduction**

- Architecture Micro-services
- Services techniques : frameworks vs infra
- L'offre Spring-Cloud
- Service de discovery et de configuration

- **GraphQL micro-services**

- Introduction
- Répartition de charge
- Circuit Breaker
- Gateway



Introduction

Architectures micro-services
Services techniques : frameworks vs
infra
L'offre Spring Cloud



Introduction

Le terme « ***micro-services*** » décrit un nouveau pattern architectural visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel

C'est le même objectif que les méthodes agiles ou les approches *DevOps* :
« *Déployer plus souvent* »



Architecture

L'architecture implique la décomposition des applications en très petit services

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

Le but étant de livrer et maintenir des systèmes complexes avec la rapidité et la qualité demandées par le business digital actuel

On l'a appelée également *SOA 2.0* ou *SOA For Hipsters*



Caractéristiques

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

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

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

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

Communication légère : REST sur HTTP, STOMP sur WebSocket,



Bénéfices

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

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

Maintenance facilitée : Le code d'un micro-service est limité à une seule fonctionnalité
=> Corrections, évolutions plus rapide

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

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

Communication inter-équipe renforcée : Full-stack team
=> Favorise le CD des applications complexes



Contraintes

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

Découverte automatique : Les services sont typiquement distribués dans l'environnement d'exécution, le scaling peut être automatisé. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

Monitoring : Les points de surveillances sont distribués. Les traces et les métriques doivent être agrégés en un point central

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

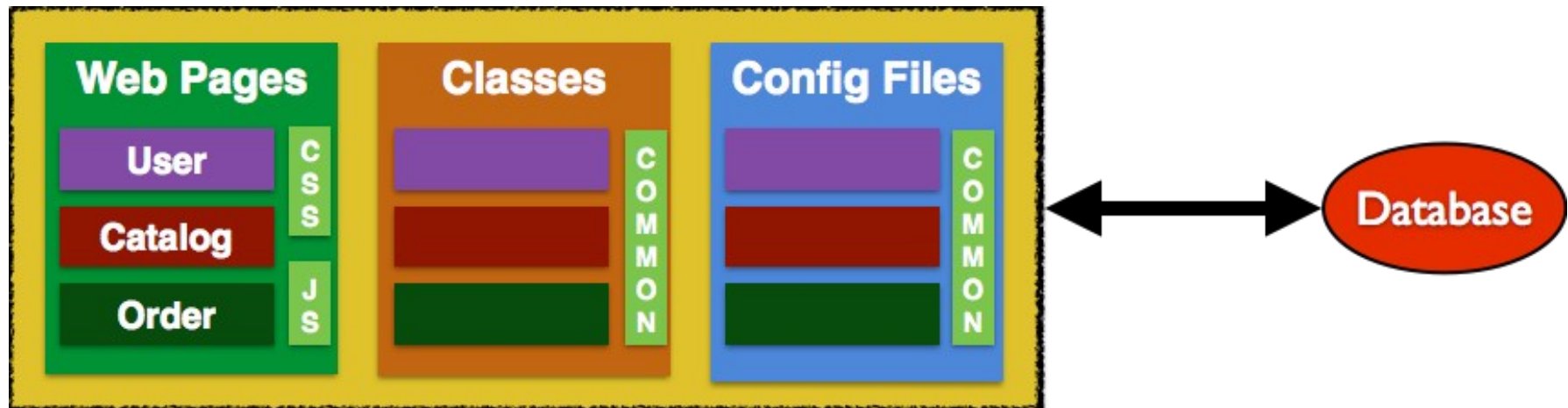
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.



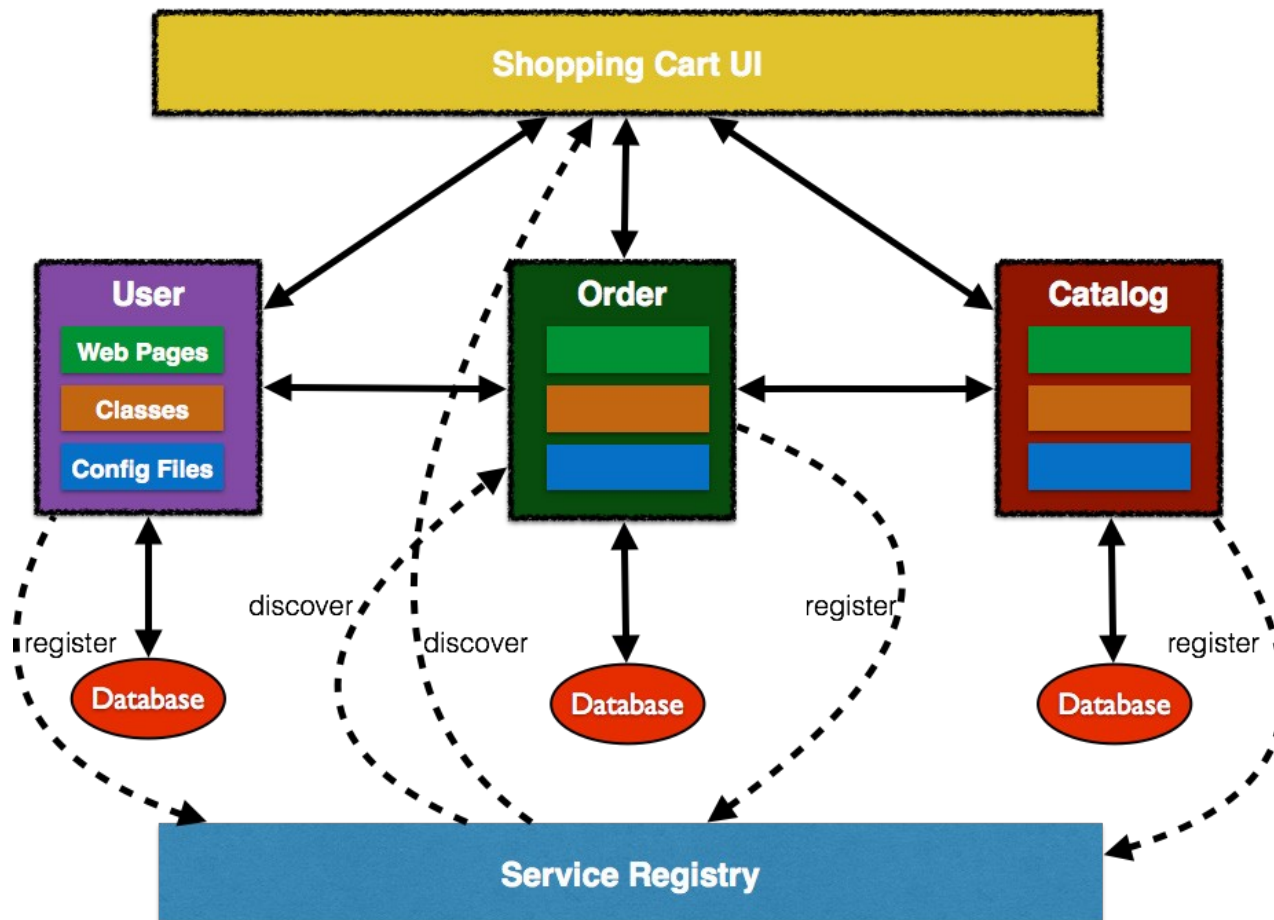
Inconvénients et difficultés

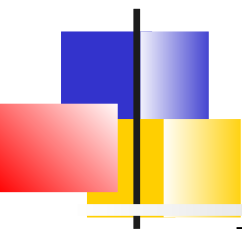
- Trouver la bonne décomposition est difficile.
Une mauvaise décomposition peut entraîner des couplages entre les micro-services
- Le côté distribué fait que le système complet est plus difficile à tester, déployer
- Le déploiement de fonctionnalités qui touche plusieurs services est plus délicat
- La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

Architecture monolithique



Version micro-service





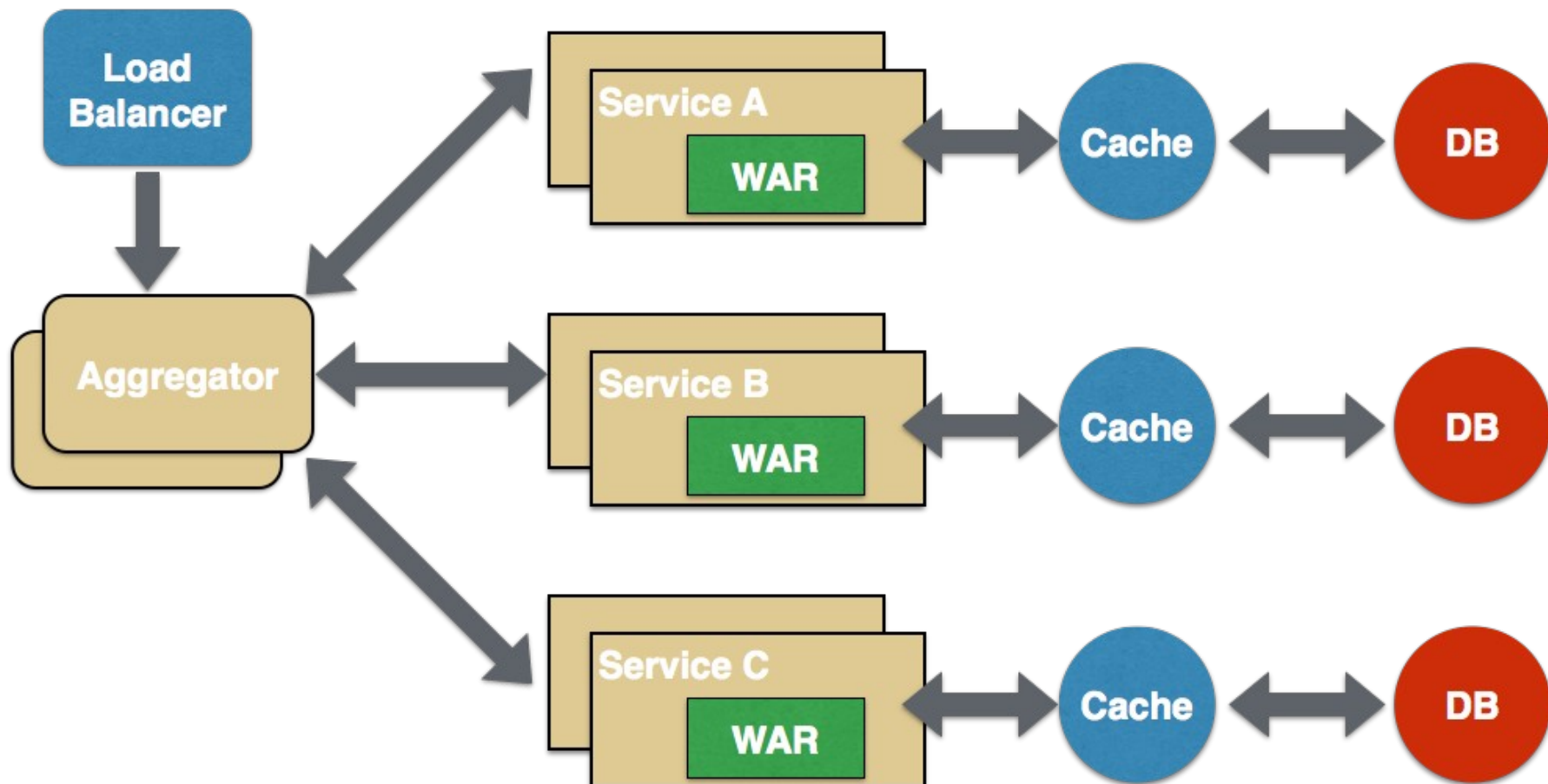
Patterns de composition

Les micro-services peuvent être combinés afin de fournir un micro-service composite.

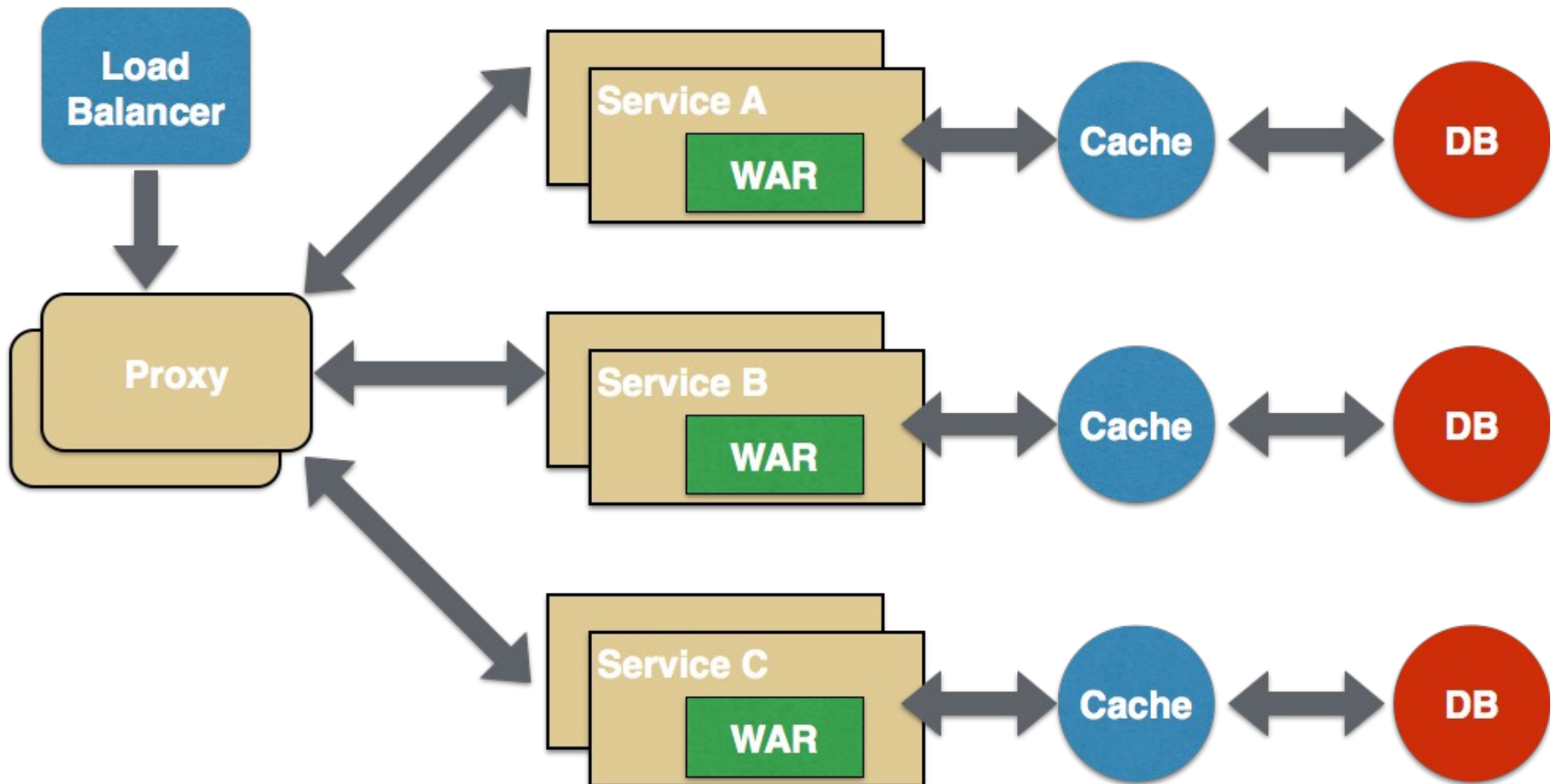
Comme design patterns de composition citons :

- **L'agrégateur** : Agrégation de plusieurs micro-service et fourniture d'une autre API REST
- **Proxy** : Délégation à un service caché avec éventuellement une transformation
- **Chaîne** : Réponse consolidée à partir de plusieurs sous-services
- **Branche** : Idem agrégateur avec le parallélisme

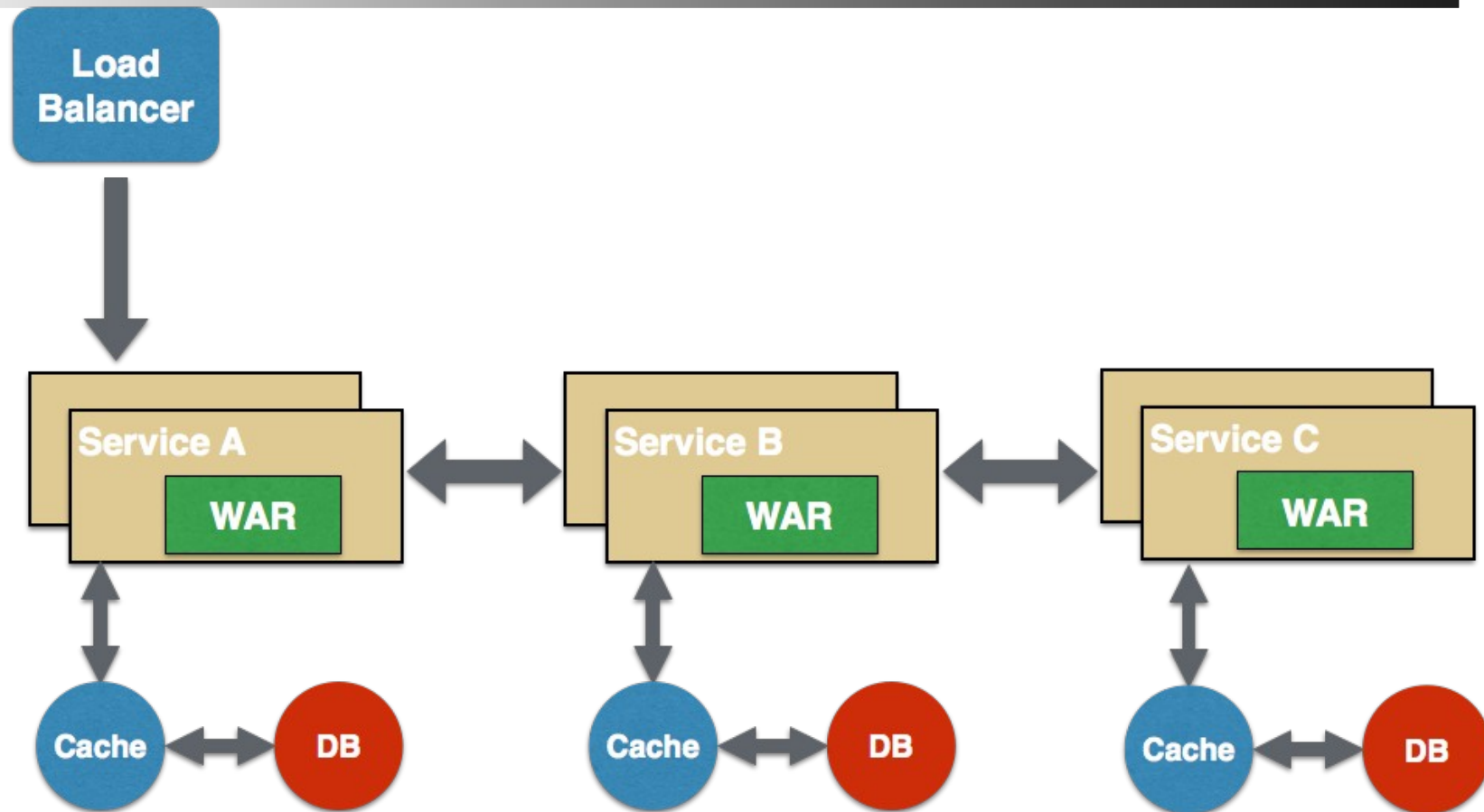
Agrégateur



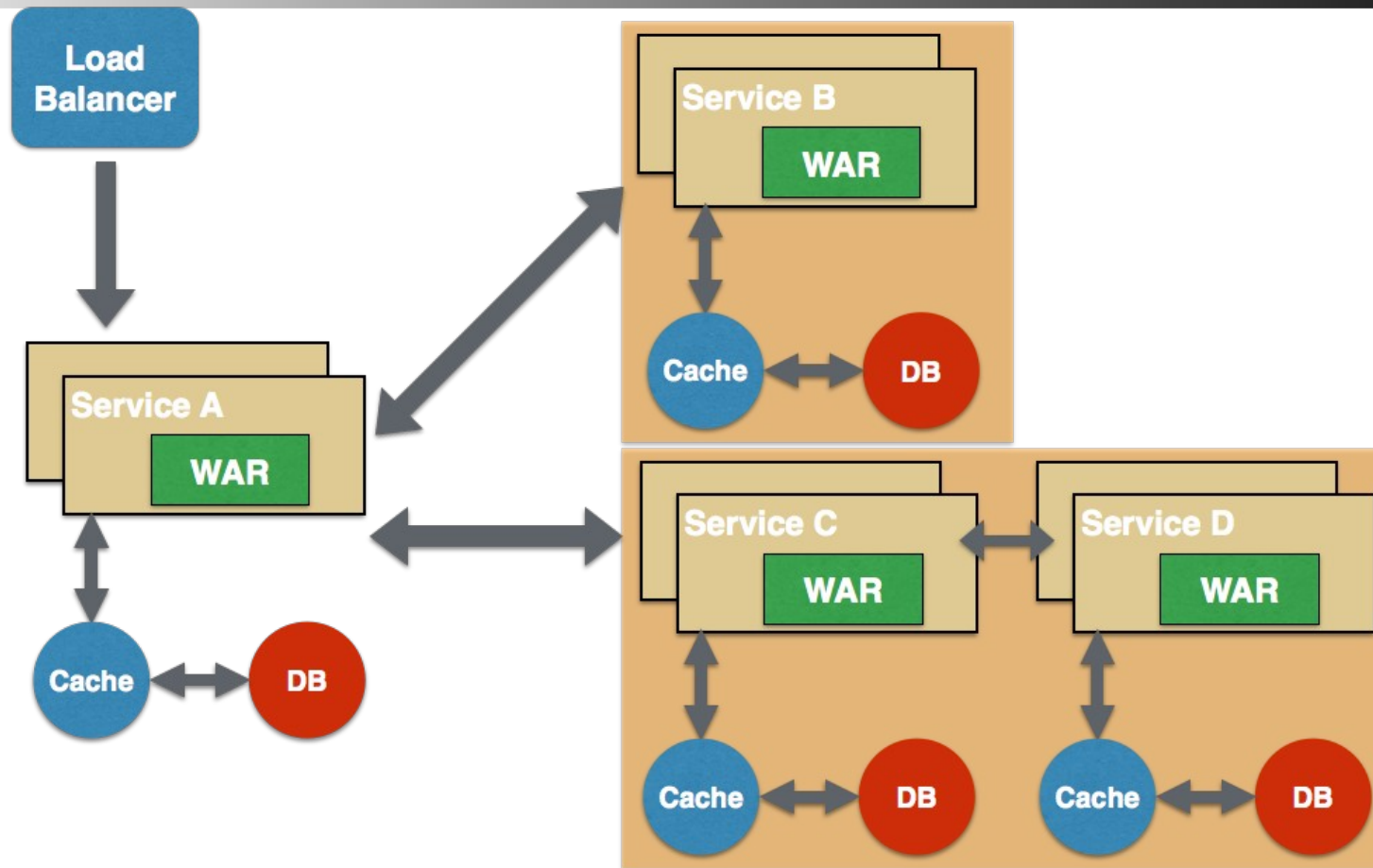
Proxy



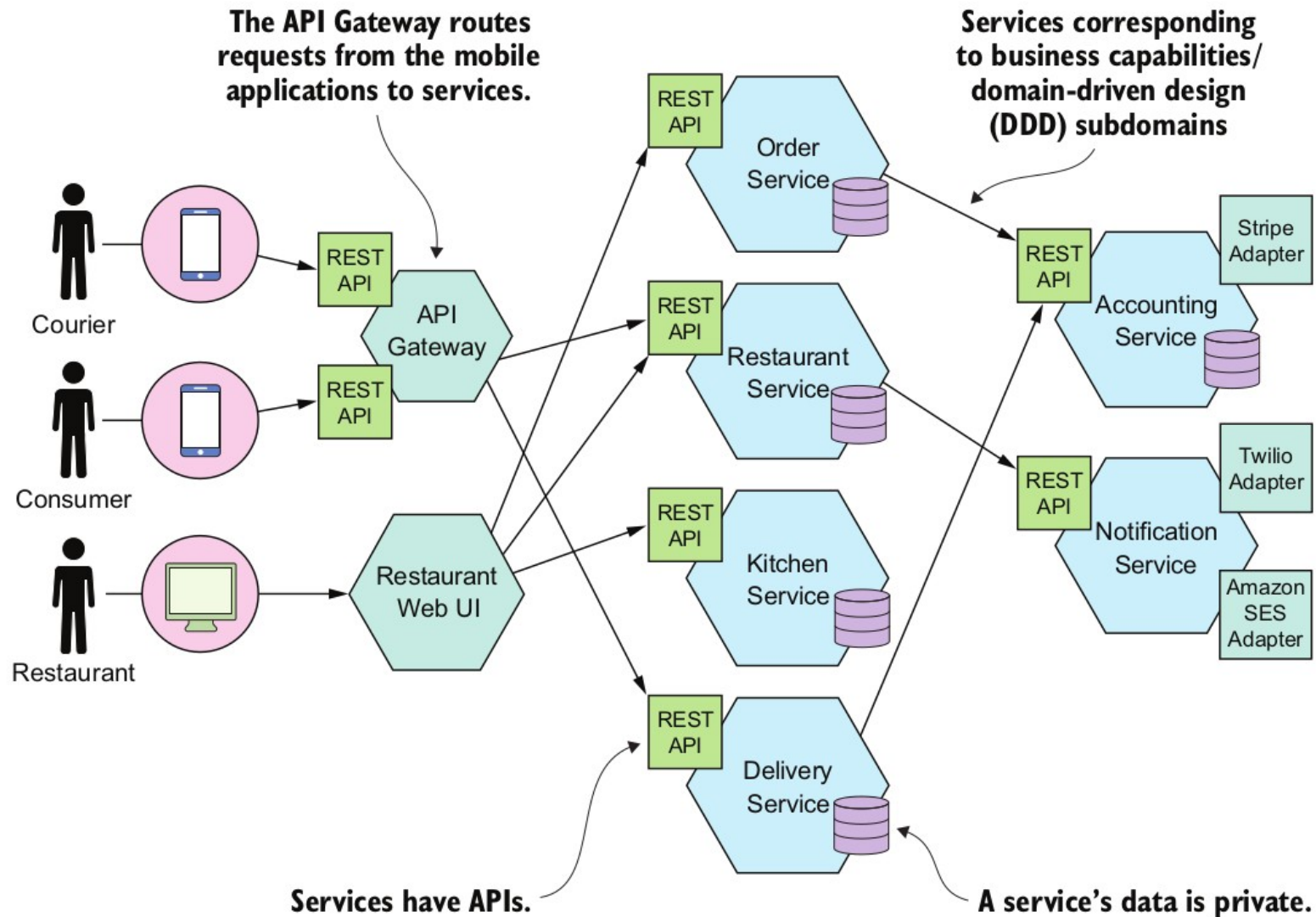
Chaîne



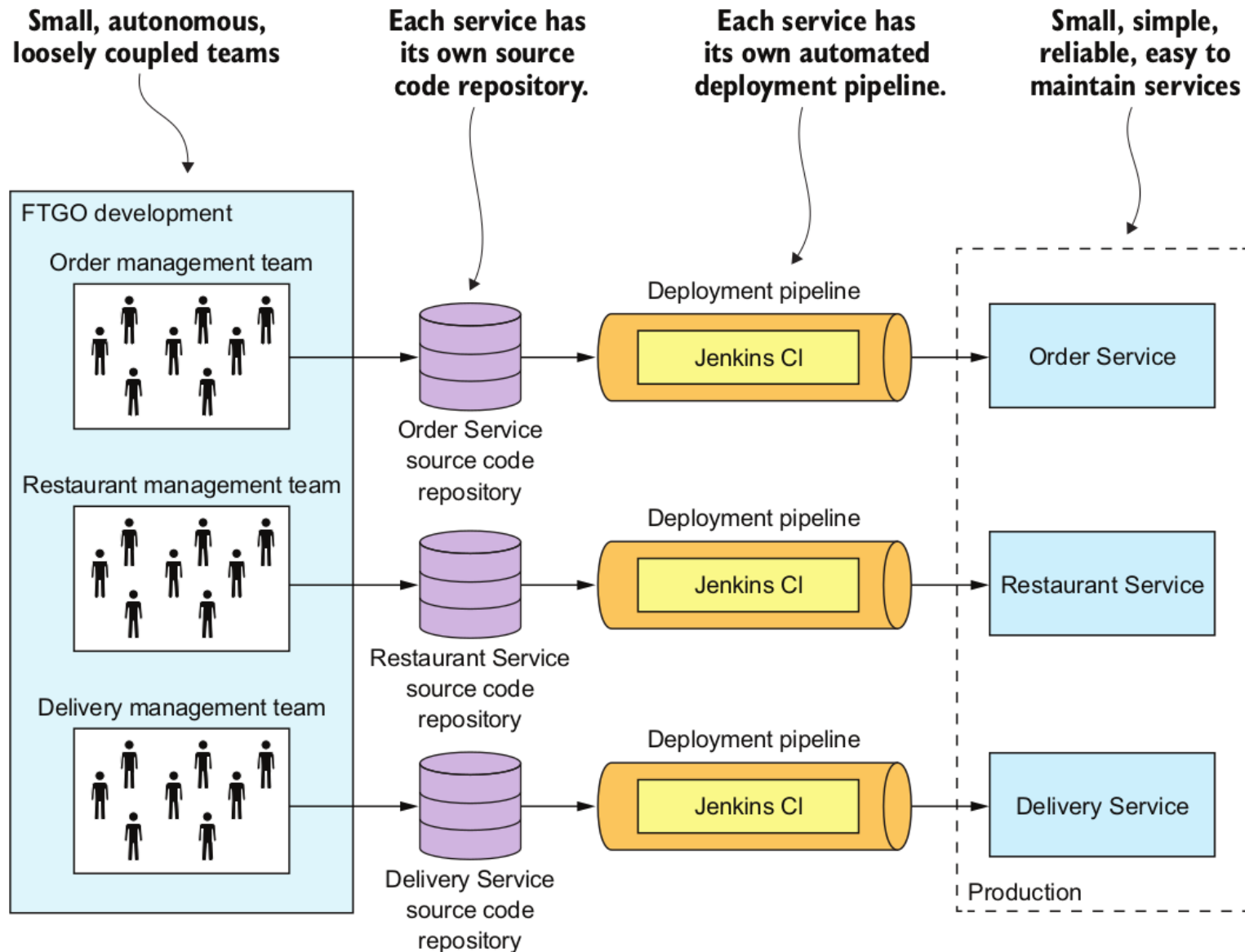
Branche

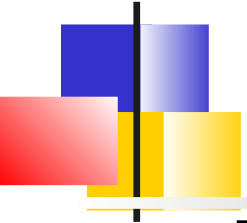


Une architecture micro-service



Organisation DevOps





Problèmes à résoudre et design patterns

Décomposition en services, Patterns :

- DDD ou sous-domaines
- Fonctionnalités métier

Communication entre service, Aspects et patterns:

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

Distribution des données, Aspects

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



Patterns et problèmes à résoudre

Déploiement des services, Patterns :

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

Observabilité afin de fournir des *insights* applicatifs :

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

Tests automatisés :

- Service en isolation, Tests des contrats (APIs)

Patterns transverses :

- Externalisation des configurations, Pipelines CD, ...

Sécurité :

- Jetons d'accès, *oAuth*, ...



Les 12 facteurs de réussite

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



Introduction

Architectures micro-services
Services techniques :
frameworks vs infrastructure
L'offre Spring Cloud

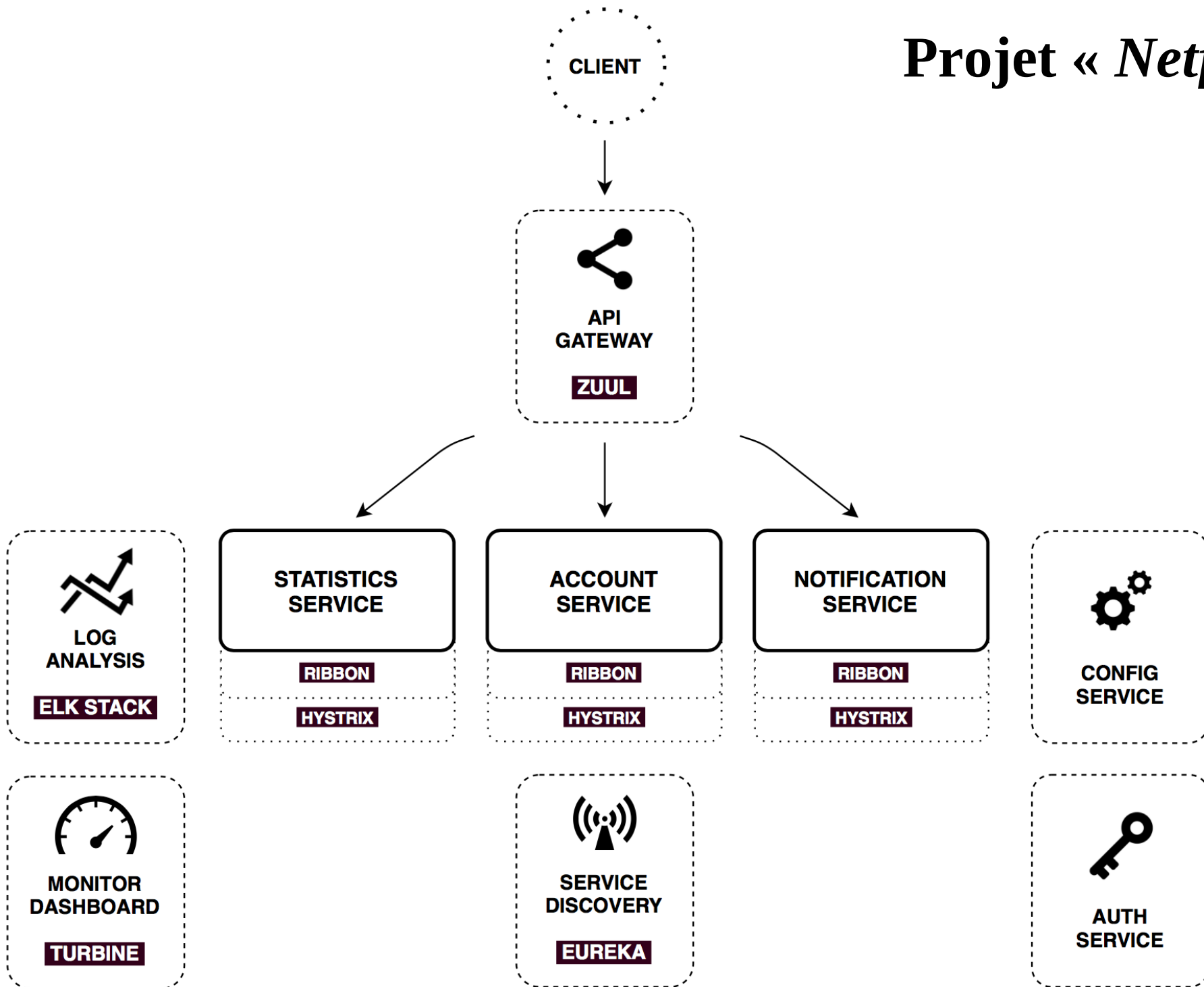


Services techniques

Les architectures micro-services nécessitent des services technique :

- Service **de discovery** permettant à un micro-service de s'enregistrer et de localiser ses micro-services dépendants
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Services **d'authentification** offrant une fonctionnalité de SSO parmi l'ensemble des micro-services
- Service de **monitoring** agrégeant les métriques de surveillance en un point central
- Support pour la répartition de charge, le fail-over, la résilience aux fautes
- Service de session applicative, horloge synchronisée,

Projet « Netflix »





Infrastructure de déploiement

Infrastructure de déploiement pour ce type d'architecture :

- Serveur matériel provisionné : Pas imaginable
- Virtualisation + outils de gestion de conf (Puppet, Chef, Ansible) : Peu adapté
- Orchestrateur de Container (Kubernetes) :
Fait pour
- Offre Cloud (AWS, Google, ...) : Economie ?
- Serverless : Nécessite des démarrages ultra-rapides

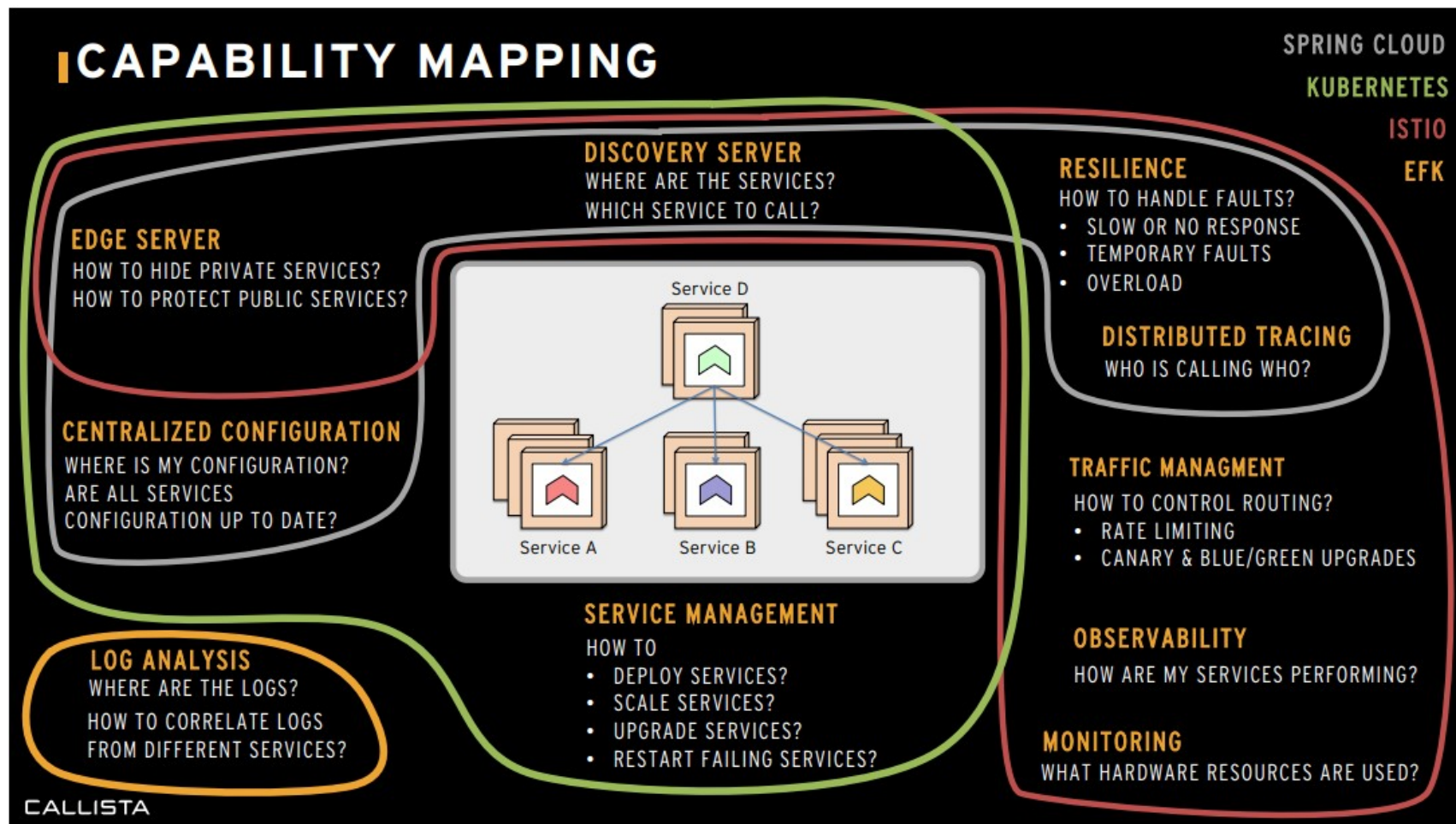


Services techniques

Qui fournit les services techniques ?

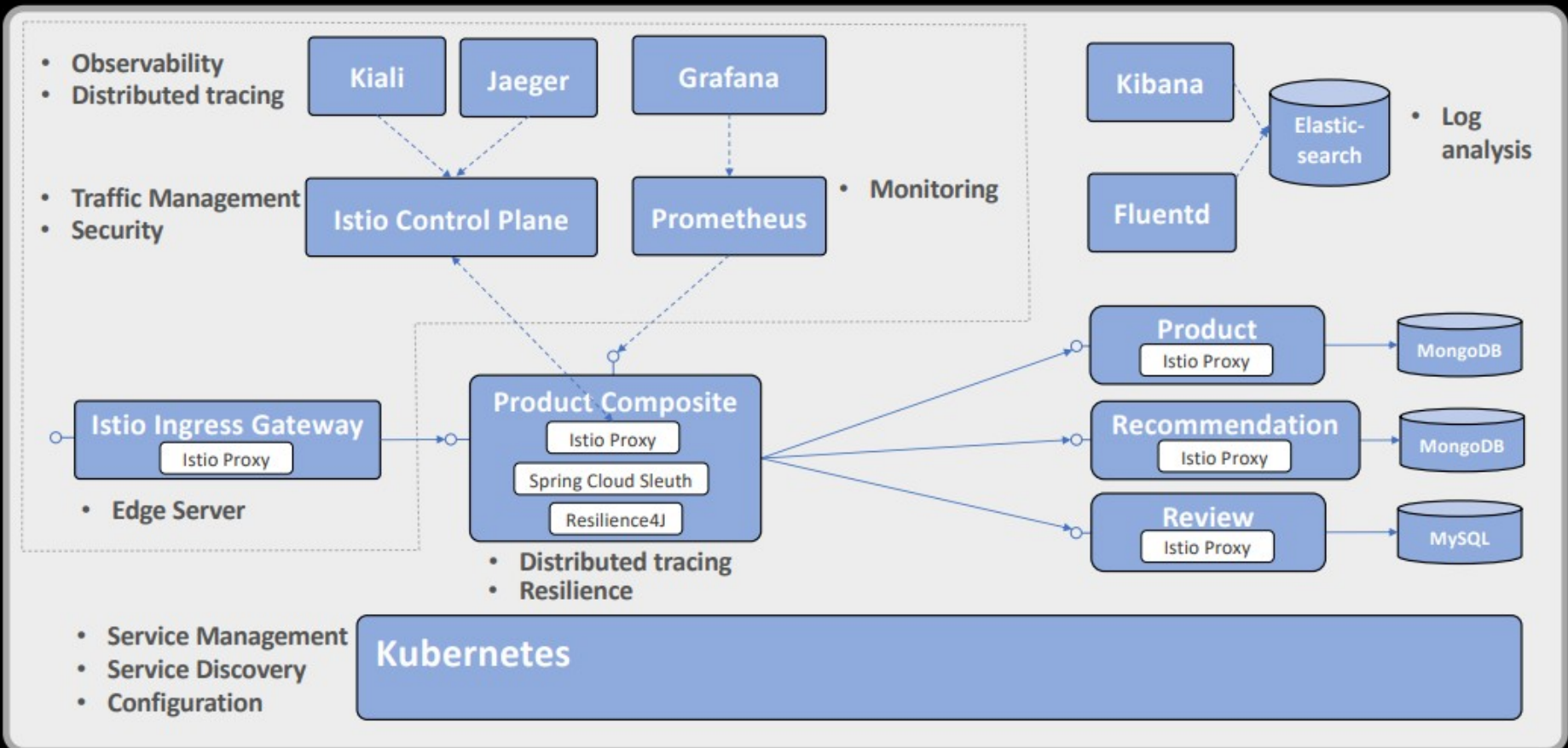
- Dans les premières architectures, c'est le software => framework Netflix
- Actuellement, de nombreux services techniques migre vers l'infrastructure :
 - Services, Config, Répartition de charge Kubernetes
 - Résilience, Sécurité, Monitoring : Service mesh de type Istio

Service requis



Un exemple de choix

SPRING CLOUD + KUBERNETES + ISTIO





Introduction

Architectures micro-services
Services techniques :
frameworks vs infrastructure
L'offre Spring Cloud



Offre Spring Cloud

Spring Cloud a donc vocation d'offrir tous les outils et services techniques nécessaire aux architectures fortement distribuées

Spring Cloud nécessite Spring Boot

Il offre :

- des facilités de déploiement vers des environnements Cloud (Amazon Web Services, Cloud Foundry, Heroku Paas)
- Des facilités pour la mise en place d'architecture Micro-services et plus généralement de systèmes distribués



Applications visées

Les applications visées sont de type SAAS software-as-a-service basées sur les technologies WEB et des méthodologies de développement DevOps

Les objectifs sont :

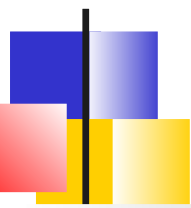
- Utiliser des formats déclaratifs pour l'**automatisation de la mise en place**. Cela pour minimiser les temps et les coûts pour intégrer de nouveaux développeurs au projet
- Avoir un contrat clair avec le système d'exploitation sous-jacent afin d'offrir le **maximum de portabilité** : La JVM
- Être adapté au déploiement sur les plate-formes de cloud afin d'**éviter le besoin de serveurs** et d'administration système
- Minimiser les divergence entre le développement et la production => **déploiement continu et agilité maximale**
- Qui puisse **se scaler sans de gros impacts** sur les outils, l'architecture ou les pratiques de développement



Offre Spring Cloud

Sprint Cloud basé sur Spring Boot fournit un framework de développement de micro-services qui :

- Apporte des abstractions des micro-services techniques nécessaires : Permettant d'adapter rapidement son code à une implémentation spécifique
- Du support pour les clients REST incluant la répartition de charge et la résilience
- Du support pour l'intégration des micro-services aux middleware de messagerie
- Bénéficie de l'environnement Spring Boot et de l'écosystème Spring (Testabilité, Spring MVC / REST, Spring Data (SQL ou NoSQL), ...)

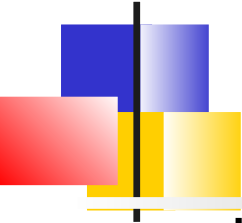


Starter de projet Spring Cloud

Les fichiers de dépendances des projets SpringCloud font intervenir 2 versions :

- La version Spring Boot
- La release Spring Cloud

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.5.10.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Edgware.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```



Spring Boot / Spring Cloud

La plupart des fonctionnalités sont offertes par Spring Boot et l'auto-configuration

Les fonctionnalités supplémentaires de Spring Cloud sont offertes via 2 librairies :

- **Spring Cloud Context** : Utilitaires et services spécifiques pour le chargement de l'*ApplicationContext* d'une application Spring Cloud (bootstrap, cryptage, rafraîchissement, endpoints)
- **Spring Cloud Commons** est un ensemble de classes et d'abstraction utilisées dans les différentes implémentations des services techniques (Par exemple : Spring Cloud Netflix vs. Spring Cloud Consul).



Contexte de bootstrap

Une application Spring Cloud crée un contexte Spring de "**bootstrap**" à partir du fichier **bootstrap.yml**.

Ce contexte est responsable de charger les propriétés de configuration à partir de ressources externes

Typiquement, un serveur de configuration distant.



Contexte bootstrap et contexte local

Les deux contextes (*bootstrap* et *local*) contribuent au contexte applicatif final de l'application

- Les propriétés de *bootstrap* (distant) ont la priorité la plus haute et ne peuvent donc pas être surchargées par une configuration locale.
- En général, le fichier de configuration local *bootstrap.yml* ne contient que l'identification de l'application et la localisation du service externe de configuration.



Exemple : *bootstrap.yml*

```
spring:
  application:
    name: members-service
  cloud:
    config:
      uri: http://config:8888
      fail-fast: true
      password: ${CONFIG_SERVICE_PASSWORD}
      username: user
```



Spring Cloud Commons

Abstractions et implémentations

Discovery (Client et serveur) : Eureka, Consul, Zookeeper

LoadBalancer : Ribbon, SpringRestTemplate, Reactive Web Client

Circuit Breaker : Hystrix, Resilience4J, Sentinel, Spring Retry



GraphQL microservices

Introduction

Répartition de charge
Circuit breaker
Gateway



Clients micro-service

Dans cette architecture un micro-service est souvent le client REST d'un autre micro-service

Les clients doivent avoir des capacités :

- De répartition dynamique de charge
- De résilience (code alternatif en cas de dysfonctionnement du serveur)



Alternatives GraphQL

Peu de client GraphQL Java disponible :

- Apollo GraphQL Client pour Android
- Netflix Domain Graph Service Framework (DGS)
- GraphQL-Kotlin

Via un client standard REST, et
implémenter soi-même l'API GraphQL



Inconvénients

Dans un contexte *SpringCloud*, les fonctionnalités de répartition de charge, de gateway s'appuient sur le service de découverte et principalement sur REST

En utilisant GraphQL, nous sommes obligés d'implémenter manuellement certaines fonctionnalités



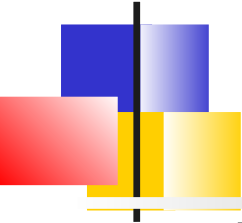
Utilisation de RestTemplate

```
HttpHeaders headers = new HttpHeaders();
headers.add("content-type", "application/graphql");

// query is a graphql query wrapped into a String
String query1 = "{\n" +
    "    locationTypes: {\n" +
    "        edges: \n" +
    "            {\n" +
    "                node: \n" +
    "            }\n" +
    "        }\n" +
    "    }";

String URL = "http://localhost:8080/graphql";

ResponseEntity<String> response = restTemplate.postForEntity(URL, new
HttpEntity<>(query1, headers), String.class);
System.out.println("The response======" + response);
```



Apollo Client

Plugin gradle permettant de récupérer le schéma du service cible

Mise au point des query dans la syntaxe GraphQL

Génération des classes modèles via le plugin gradle

Builder le client et l'utiliser



Exemple

// Création de `ApolloClient` avec l'URL du serveur cible

```
ApolloClient apolloClient = ApolloClient.builder()  
    .serverUrl("https://your.domain/graphql/endpoint")  
    .build();
```

// Requête

```
apolloClient.query(new LaunchDetailsQuery("83"))  
    .enqueue(new ApolloCall.Callback<LaunchDetailsQuery.Data>() {  
        @Override  
        public void onResponse(@NotNull Response<LaunchDetailsQuery.Data>  
response) {  
            Log.e("Apollo", "Launch site: " + response.getData().launch.site);  
        }  
  
        @Override  
        public void onFailure(@NotNull ApolloException e) {  
            Log.e("Apollo", "Error", e);  
        }  
    });
```



GraphQL microservices

Introduction

Répartition de charge

Circuit breaker

Gateway



Introduction

La répartition de charge côté client consiste à interroger le service de *Discovery* pour connaître les instances disponibles et appliquer un algorithme de répartition (ex. Round Robin) pour répartir les requêtes.

- Cela peut être fait directement en utilisant l'API *DiscoveryClient*
- Ou en utilisant l'API *spring-cloud-loadbalancer* de SC Commons basée sur *RestTemplate*
- Ou se reposer sur l'infrastructure (Kubernetes par exemple)



Implémentation via *DiscoveryClient*

```
@Autowired
DiscoveryClient discoveryClient;

...

List<ServiceInstance> instances =
    discoveryClient.getInstances("another-service");

Instance instance = _chooseOneInstance(instances);
String url = "http://" + instance.getHost() + ":" + instance.getPort();

// Le client est typiquement un RestTemplate ou un apolloClient
client.rootUri(url);

...

```

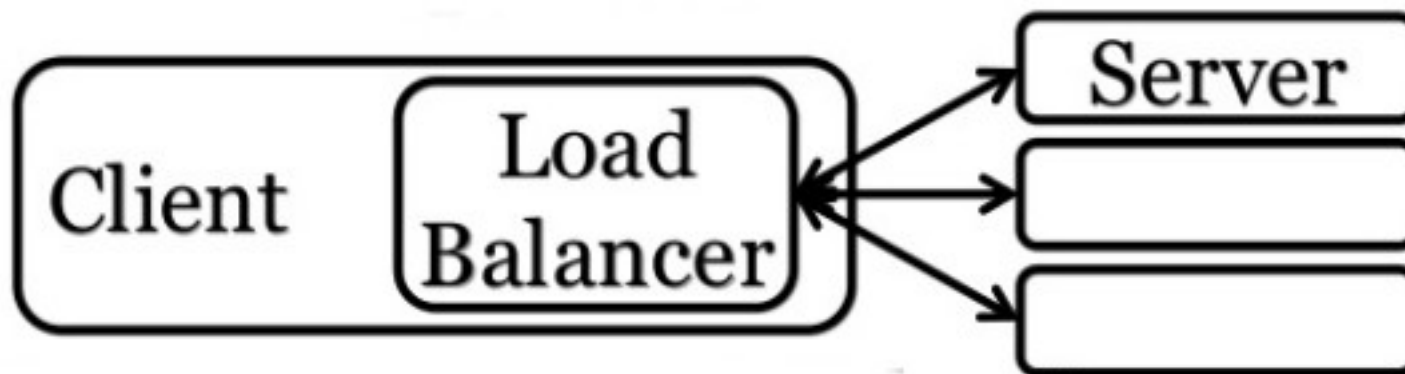



Spring Cloud LoadBalancer

Spring Cloud fournit sa propre abstraction et implémentation d'équilibreur de charge côté client.

Le client Rest découvre les instances disponibles à partir du service de discovery.

- Il utilise le bean *DiscoveryClient* disponible dans le classpath
- Il peut également utiliser une librairie de cache (par défaut *Caffeine*)





Avantages Client-side

Avec l'équilibrage de charge côté client, c'est le client qui décide quel nœud reçoit la requête car il connaît la topologie.

Théoriquement, Il peut en amont éviter de surcharger un service qui est trop chargé ou inaccessible .



Dépendances

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
<!-- Plus discovery client, par exemple -->
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```



@LoadBalanced

Il suffit d'annoter un bean de type *RestTemplate* avec **@LoadBalanced** pour bénéficier de l'implémentation sous-jacente de l'interface de *Spring Cloud Commons* .

Par défaut, l'implémentation legacy Ribbon sauf si :
spring.cloud.loadbalancer.ribbon.enabled: false

```
@Configuration
public class ClientConfiguration {

    @Autowired
    RestTemplateBuilder builder;

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return builder.build();
    }
}
```



@LoadBalanced

Pile réactive

Dans un contexte réactive la même annotation ***@LoadBalanced*** est utilisée

- L'annotation se place sur une méthode fournissant un ***WebClient.Builder***.
- Le builder permet de construire un ***WebClient***
- Le *WebClient* est utilisé pour effectuer les requêtes Rest



Exemple

```
@Bean
```

```
@LoadBalanced
```

```
WebClient.Builder builder() {  
    return WebClient.builder();  
}
```

```
@Bean
```

```
WebClient webClient(WebClient.Builder builder) {  
    return builder.build();  
}
```

```
// Utilisation du bean
```

```
Flux<Greeting> call(String url) {  
    return  
    webClient.get().uri(url).retrieve().bodyToFlux(Greeting.class);  
}
```



Usage

```
@Log4j2
@Component
class ConfiguredWebClientRunner {

    ConfiguredWebClientRunner(WebClient http) {
        call(http, "http://api/greetings").subscribe(
            greeting -> log.info("configured: " + greeting.toString()));
    }
}
```



GraphQL microservices

Introduction
Répartition de charge
Circuit breaker
Gateway



Contexte du pattern

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



Solution

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

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



Spring Cloud Breaker Pattern

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

- Netflix Hystrix
- Resilience4j
- Sentinel
- Spring Retry



Concepts cœur

Pour utiliser le pattern dans son application, Spring Cloud injecte un ***CircuitBreakerFactory*** en fonction des starters trouvés dans le classpath.

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

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



Exemple

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

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

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



Patterns Resilience4J

Retry : Répéter les exécutions échouées

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

Circuit Breaker : blocages temporaires des défaillances possibles

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

Rate limiter : Limiter les exécutions

Limitez le taux des requêtes

Time Limiter : Limiter le temps d'exécution des requêtes

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

Bulkhead : Limiter les exécutions concurrentes

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

Cache : Mémoriser un résultat réussi

Une partie des demandes peut être similaire.

Fallback : Fournir un résultat alternatif pour les échecs

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



GraphQL microservices

Introduction
Répartition de charge
Circuit breaker
Gateway



Introduction

Le pattern **API Gateway** permet à un client nécessitant plusieurs services de back-end de ne s'adresser qu'à **un seul** service.

La gateway route alors les requêtes vers les back-end implémentant les services demandés

En plus des fonctionnalités de routing, la gateway peut intégrer des services transverses à toute l'architecture micro-service : authentication, CORS, filtrage d'entêtes HTTP, ...



Spring Cloud Routing

Spring Cloud Routing propose 2 starters dédiés aux fonctionnalités de routing

- Netflix Zuul (Projet en maintenance)
- Spring Cloud Gateway



Caractéristiques Spring Cloud Gateway

- Construit sur Spring Framework 5, Project Reactor et Spring Boot 2.0
 - => Netty et modèle réactif exclusivement
- Capable de faire correspondre les routes en fonction de n'importe quel attribut de requête.
- Les filtres sont spécifiques aux routes.
- Intégration du disjoncteur Hystrix.
- Intégration Spring Cloud DiscoveryClient
- Possibilité de limiter la cadence des requêtes
- URL rewriting



Exemple gateway

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
    return builder.routes()
    // Routing + ajout d'entête
        .route(p -> p
            .path("/get")
            .filters(f -> f.addRequestHeader("Hello", "World"))
            .uri("http://httpbin.org:80"))
    // Utilisation de Hystrix
        .route(p -> p
            .host("*.hystrix.com")
            .filters(f -> f.hystrix(config -> config
                .setName("mycmd")
                .setFallbackUri("forward:/fallback")))
            .uri("http://httpbin.org:80"))
        .build();
}
```



Intégration avec *DiscoveryClient*

Pour créer des routes vis à vis des services enregistrés dans le serveur de discovery :

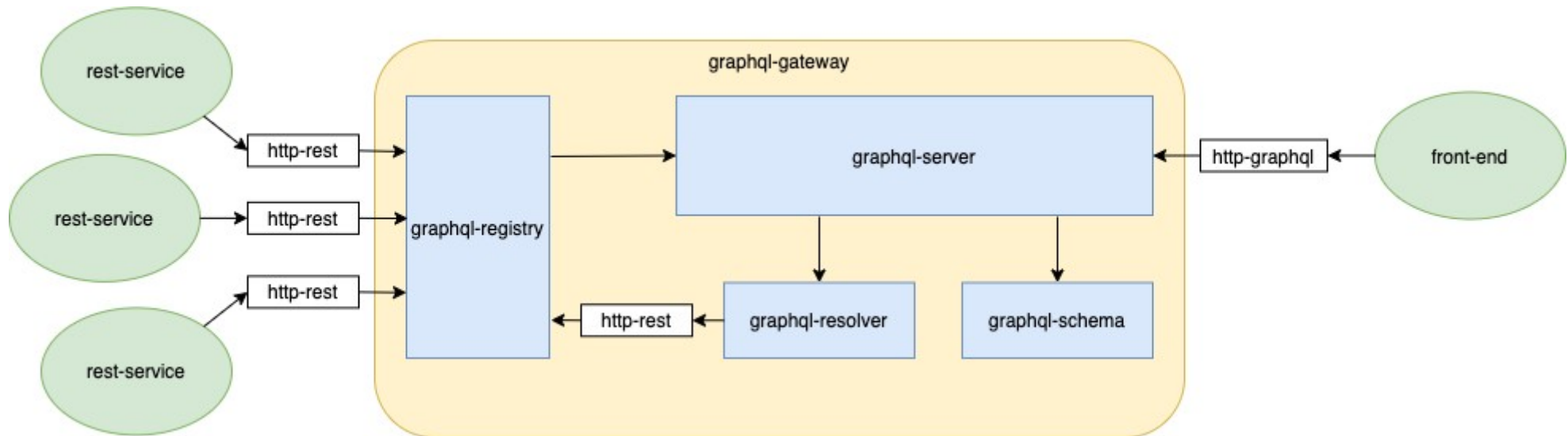
```
spring.cloud.gateway.discovery.locator.enabled=true
```

Par défaut, des routes sont définies pour chaque service enregistrés

Les URLs ***/serviceld/***** sont routées vers le service en enlevant le préfixe

Architecture mixte

graphql-gateway route et enregistre les services back-end !!



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