





# Architecture Micro-services avec Spring Cloud

---

David THIBAU – 2021

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- **Introduction**

- Architecture Micro-services
- Services techniques : frameworks vs infra
- L'offre Spring-Cloud

- **Discovery et Config**

- Service Eureka
- Spring Cloud Config

- **Support pour les clients**

- Répartition de charge
- Le pattern Circuit breaker
- Clients déclaratifs avec Feign
- API Gateway
- *Spring Cloud Contract*

- **Monitoring**

- Agrégateur de flux Hystrix
- Tracing avec Spring Sleuth
- Centralisation des logs

- **Messaging**

- Spring Cloud Stream : Concepts et introduction
- Spring Cloud Data Flow
- Saga Pattern

- **Spring Cloud Security**

- Rappels OAuth2
- Spring Boot et OAuth2
- Spring Cloud Security

- **Déploiement**

- Alternatives
- Support pour les conteneurs
- Spring Cloud et Kubernetes



# 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.

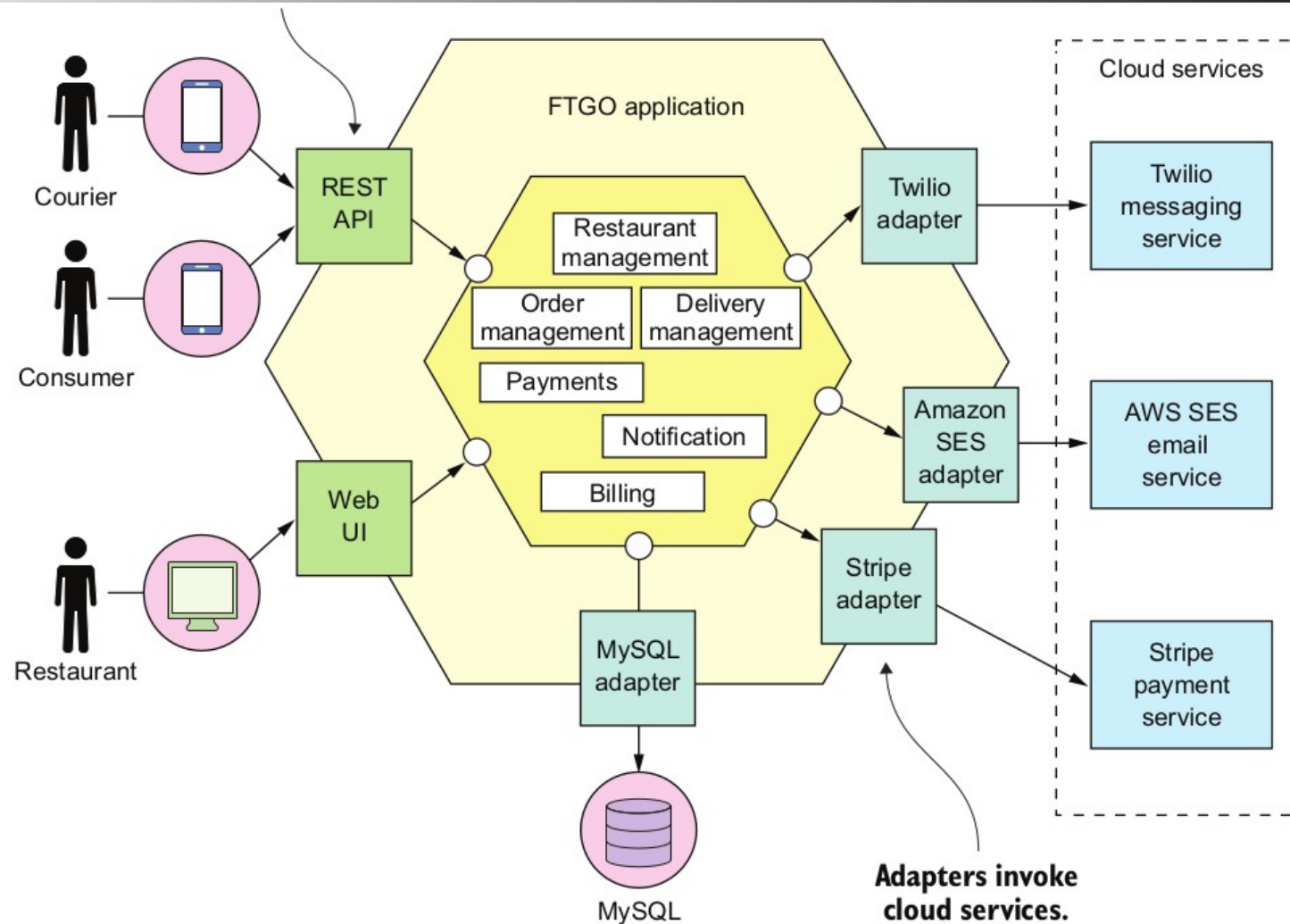


# Inconvénients et difficultés

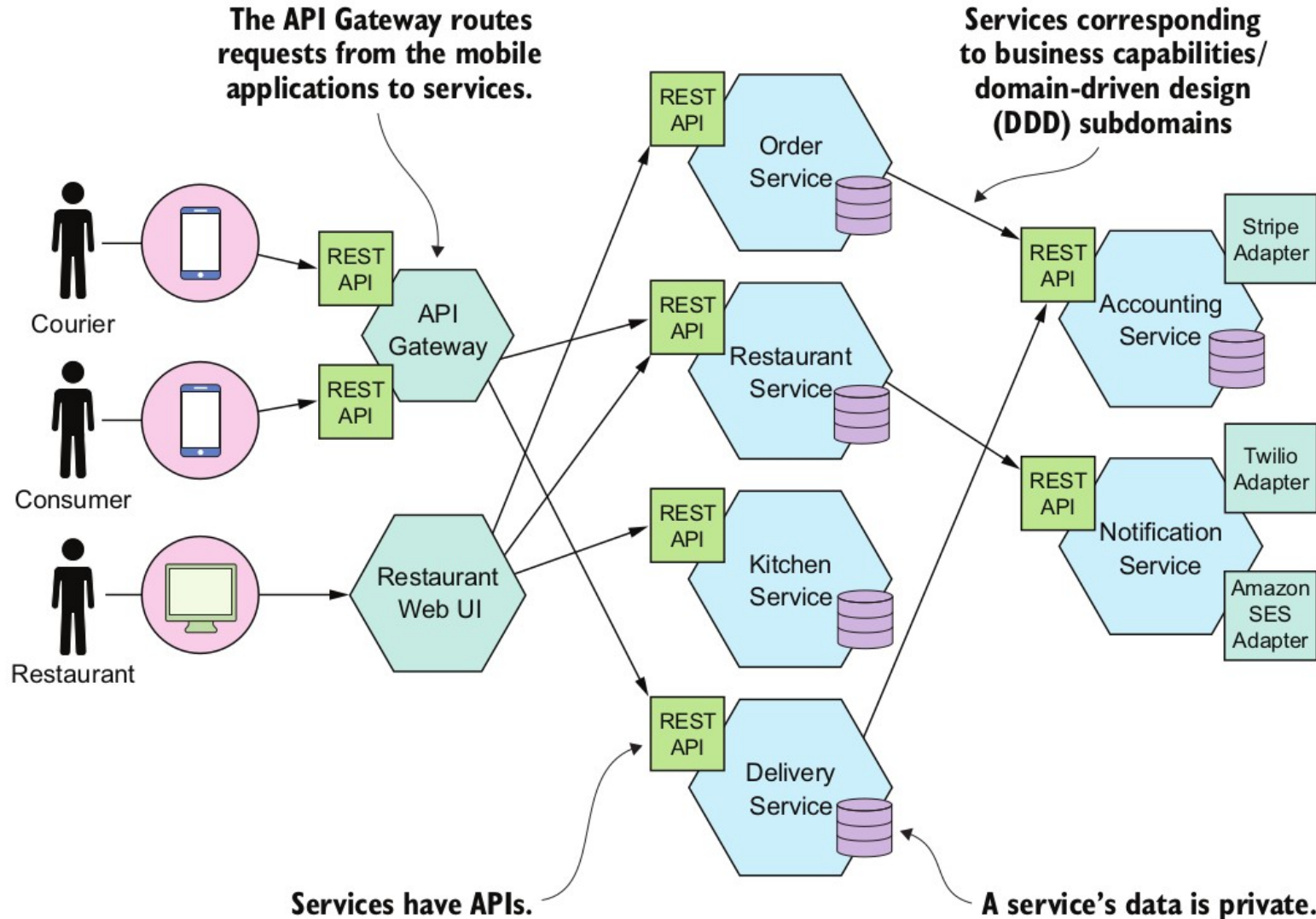
---

- 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

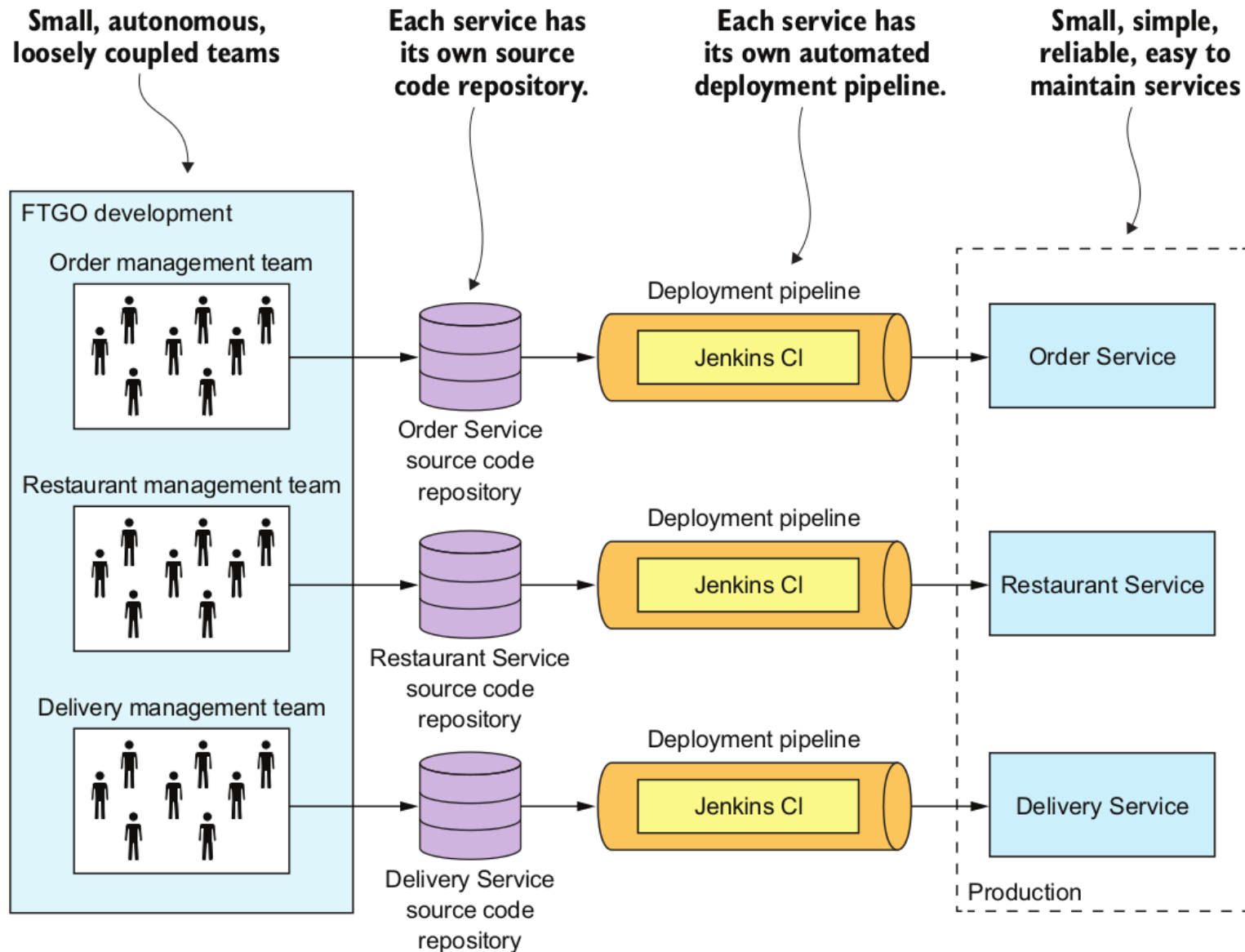
# Architecture monolithique Hexagonale

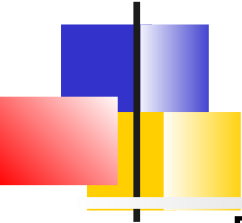


# Une architecture micro-service



# 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 Transverses

---

De nombreux services transverses peuvent être fournis par un framework ou une infrastructure

- 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, de génération et de relais de jeton
- 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



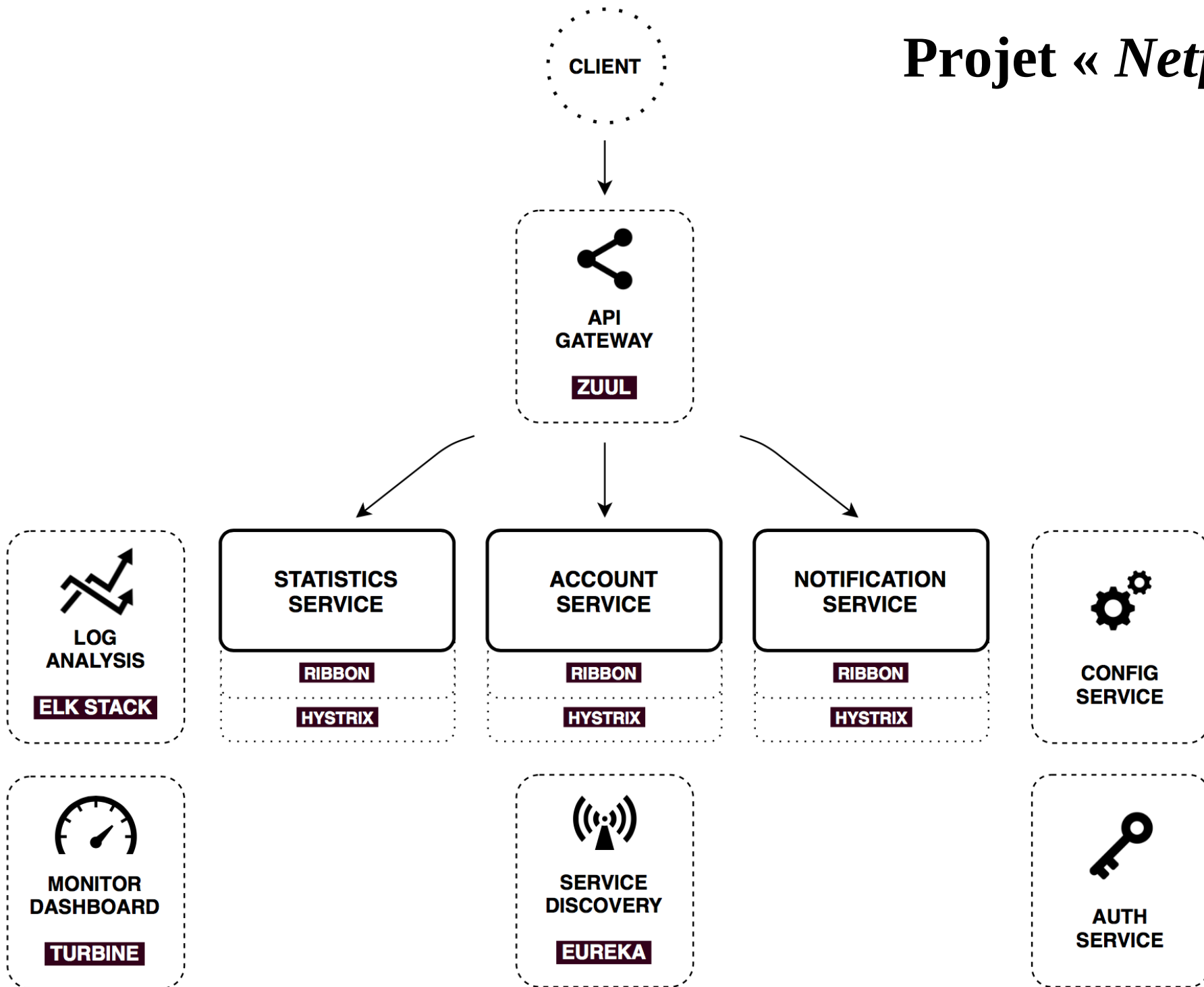
# Services techniques vs Infra

---

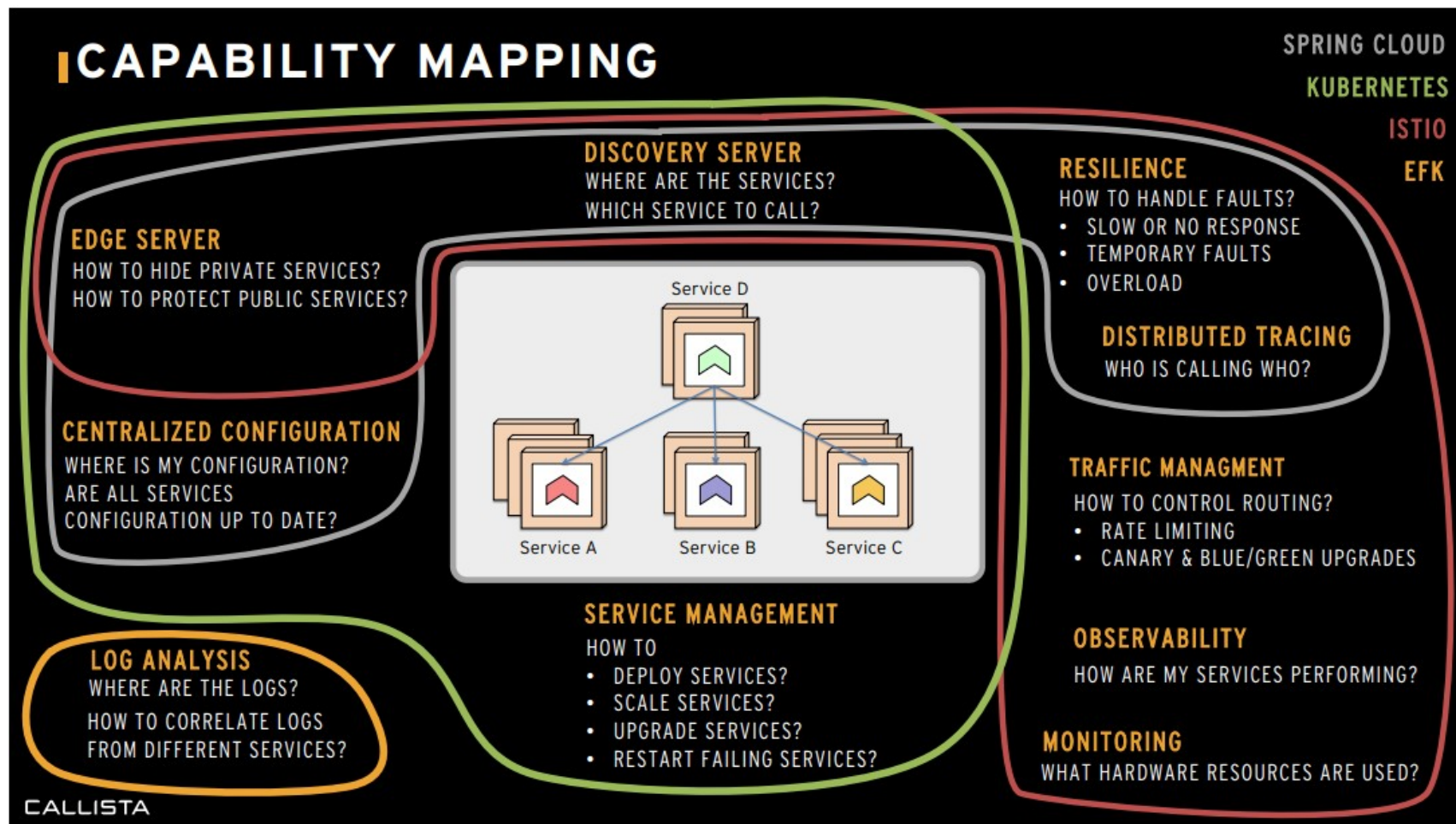
Qui fournit les services techniques ?

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

# Projet « Netflix »



# Capability Mapping





# Introduction

---

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



# Offre Spring Cloud

---

Spring Cloud a pour vocation de faciliter l'implémentation des patterns des architectures distribuées

Spring Cloud nécessite Spring Boot

Les solutions Spring Cloud s'exécutent dans différents environnements distribués :

- Local au développeur
- Réseau d'entreprise (Bare Metal ou autres)
- Plateformes gérés (Kubernetes, Amazon Web Services, Cloud Foundry, Heroku Paas)



# 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



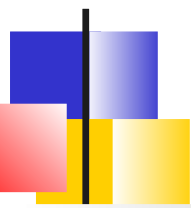


# Offre Spring Cloud

---

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), ...)

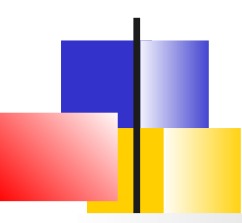


# Starter de projet Spring Cloud

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** : Resilience4J, Sentinel, Spring Retry, Hystrix

**Routing** : Gateway, Zuul



# Spring et les Patterns Micro-services

---

Communication entre service, Aspects et patterns:

- Spring Cloud OpenFeign
- Spring Cloud Discovery
- Spring Cloud CircuitBreaker
- Spring Cloud Gateway

Distribution des données, Aspects

- Spring Cloud Messaging (Saga Pattern)
- Spring Cloud Kafka Stream (KTable)





# Spring et les Patterns Micro-services (2)

---

## Déploiement des services, Patterns :

- Spring Boot Plugin : spring-boot:build-image
- Spring Cloud Kubernetes
- Spring Native (ServerLess)

## Observabilité afin de fournir des *insights* applicatifs :

- Actuator
- Sleuth, Zipkin

## Tests automatisés :

- Spring Cloud Contract

## Patterns transverses :

- Spring Cloud Config

## Sécurité :

- Spring Security 5,

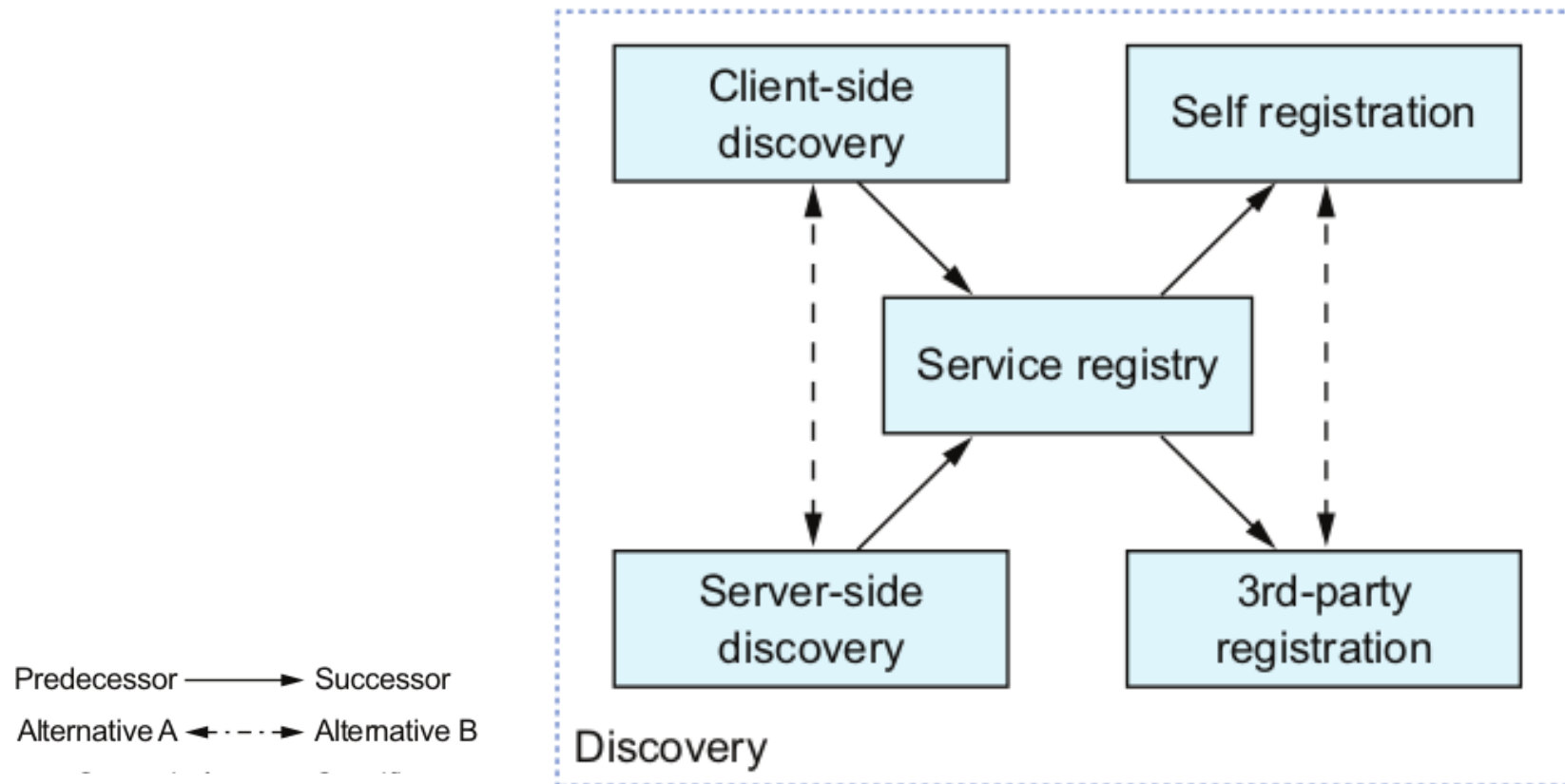


# Serveur de Discovery et Configuration

---

**Service de Discovery**  
Spring Cloud Config

# Types d'implémentation du service discovery





# Patterns

---

***Self registration*** : Une instance de service s'enregistre au démarrage auprès du service de discovery

Exemple : Eureka

***Client-side discovery*** : Un service client récupère la liste des services disponibles auprès du service de discovery et équilibre la charge

Exemple : Eureka, Consul

***3rd party registration*** : Les instances de services sont automatiquement enregistrés via un agent tiers

Exemple : Consul, Kubernetes

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

Exemple : Kubernetes



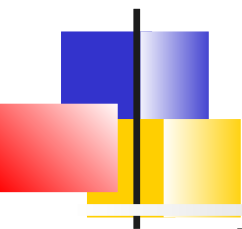
# *@EnableDiscoveryClient*

---

L'annotation **@EnableDiscoveryClient** recherche des implémentations de l'interface **DiscoveryClient** :

- Spring Cloud Netflix **Eureka**,
- Spring Cloud **Consul** Discovery
- Spring Cloud **Zookeeper** Discovery.
- Spring Cloud **Kubernetes** Discovery

L'application s'enregistre alors auprès d'un serveur de discovery distant.



# Service Discovery Eureka

---

La société Netflix a mis a disposition de la communauté Open Source **Eureka** un service de découverte pour les micro-services

Eureka offre :

- Une API Rest, et une page Web
- Simple et rapide
- Peut fonctionner en cluster (1 par data centre)



# Fonctionnement

---

Quand un client s'enregistre avec *Eureka*, il fournit des méta-données comme le hôte, le port, l'indicateur de santé, la page d'accueil

*Eureka* reçoit périodiquement des messages (heartbeat) de chaque instance des services.

Si il ne reçoit pas de messages au bout d'un *timeout* configurable, l'instance est supprimée du registre.



# Mise en place du serveur Eureka

---

L'application Eureka Server doit déclarer le starter

***spring-cloud-starter-eureka-server***

```
@SpringBootApplication
```

```
@EnableEurekaServer
```

```
public class AnnuaireApplication {
```

```
    public static void main(String[] args) {
```

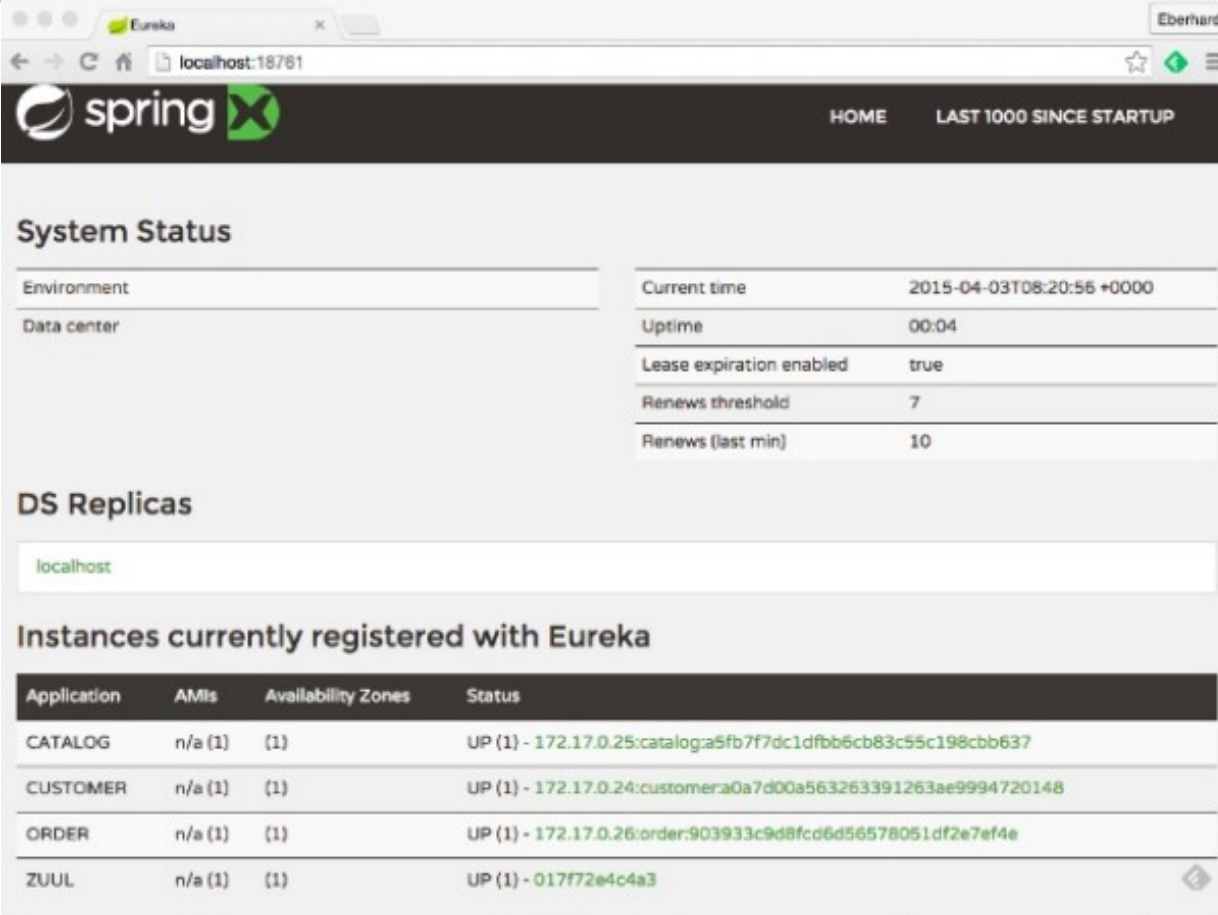
```
        SpringApplication.run(AnnuaireApplication.class,  
                                args);
```

```
    }
```

```
}
```



# Page Eureka



A screenshot of the Spring Eureka web interface running in a browser. The browser's address bar shows 'localhost:18761'. The page has a dark header with the Spring logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into three sections: 'System Status', 'DS Replicas', and 'Instances currently registered with Eureka'. The 'System Status' section contains two tables. The first table lists 'Environment' and 'Data center'. The second table lists 'Current time' (2015-04-03T08:20:56 +0000), 'Uptime' (00:04), 'Lease expiration enabled' (true), 'Renews threshold' (7), and 'Renews (last min)' (10). The 'DS Replicas' section shows 'localhost' in a search box. The 'Instances currently registered with Eureka' section contains a table with four columns: 'Application', 'AMIs', 'Availability Zones', and 'Status'. It lists four applications: CATALOG, CUSTOMER, ORDER, and ZUUL, all with status 'UP (1)' and their respective instance IDs.

**System Status**

Environment	Current time	2015-04-03T08:20:56 +0000
Data center	Uptime	00:04
	Lease expiration enabled	true
	Renews threshold	7
	Renews (last min)	10

**DS Replicas**

localhost

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones	Status
CATALOG	n/a (1)	(1)	UP (1) - 172.17.0.25:catalog:a5fb7f7dc1dfbb6cb83c55c198cbb637
CUSTOMER	n/a (1)	(1)	UP (1) - 172.17.0.24:customer:a0a7d00a563263391263ae9994720148
ORDER	n/a (1)	(1)	UP (1) - 172.17.0.26:order:903933c9d8fcd6d56578051df2e7ef4e
ZUUL	n/a (1)	(1)	UP (1) - 017f72e4c4a3



# Client Eureka

---

Pour s'enregistrer sur un service Eureka, un client doit simplement déclarer le starter<sup>1</sup>  
**`spring-cloud-starter-netflix-eureka-client`**

L'enregistrement a alors 2 conséquences :

- Le serveur surveille périodiquement le statut du service
- Le client interroge régulièrement le serveur pour localiser les autres services

1. Sans SpringBoot, il faut activer la l'auto-configuration avec `@EnableDiscoveryClient`



# Utilisation du client Eureka

---

La principale API de *EurekaClient* est de localiser une instance d'un service :

```
@Autowired
private EurekaClient discoveryClient;

public String serviceUrl() {
    InstanceInfo instance =
        discoveryClient.getNextServerFromEureka("STORES", false);
    return instance.getHomePageUrl();
}
```



# Configuration

---

Pour localiser le serveur Eureka

`eureka.client.serviceUrl.defaultZone`

Les valeurs par défaut pour  
l'enregistrement du nom de l'application,  
de l'hôte virtuel et du port sont :

- `${spring.application.name}`
- `${spring.application.name}`
- `${server.port}`



# Page de statut et indicateur de santé

---

La page d'accueil d'*Eureka* propose des liens vers différentes ressources du service enregistré.

Par défaut ce sont les liens */actuator* mais cela peut être changé

eureka:

instance:

statusPageUrlPath: `${management.context-path}/info`

healthCheckUrlPath: `${management.context-path}/health`

homePageUrl: `https://${eureka.hostname}/`



# Méta-données additionnelles

---

Des méta-données additionnelles peuvent être ajoutées dans

*eureka.instance.metadataMap*

Elles sont alors accessibles des clients distants qui peuvent en faire bon usage

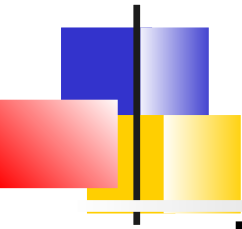
Ces méta-données spécifiques sont utilisées par les solutions de Cloud supportées :  
Cloudfoundry, AWS par exemple



# Exemple

---

```
spring:
  application:
    name: members-service # Service registers under this name
eureka:
  client:
    serviceUrl:
      defaultZone: http://annuaire:1111/eureka/
  instance:
    LeaseRenewalIntervalInSeconds: 5 # 5s au lieu de 30s (dev)
    MetadataMap :
      instanceId=${spring.application.name}:${random.value}
```



# Heartbeat

---

Les services envoient un heartbeat toutes les 30 secondes.

=> Un service n'est pas disponible pour la découverte par un autre client tant que l'instance, le serveur et le client n'ont pas tous les mêmes données dans leur cache local (cela peut donc prendre 3 pulsations)

La valeur des 30s peut être modifiée via la configuration :  
*eureka.instance.leaseRenewalIntervalInSeconds*

Cependant, il n'est pas recommandé de le modifier en production





# Configurations serveur

---

Par défaut, chaque serveur Eureka est également un client et nécessite au moins un serveur pair pour s'enregistrer.

=> 2 configurations typiques :

- Standalone : Désactiver l'enregistrement du seul serveur Eureka
- Peer : déclarer au minimum un serveur peer pour chaque instance du serveur Eureka



# Configuration *standalone*

---

La configuration *standalone* est utile lorsque l'on ne démarre qu'un seul service Eureka

```
eureka:  
  instance:  
    hostname: localhost  
  client:  
    register-with-eureka: false  
    fetch-registry: false  
    serviceUrl:  
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```



# Configuration en paire

---

```
---
spring:
  profiles: peer1
eureka:
  instance:
    hostname: peer1
  client:
    serviceUrl:
      defaultZone: http://peer2/eureka/
```

```
---
spring:
  profiles: peer2
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1/eureka/
```



# Zones

---

Si les clients Eureka sont dans des zones différentes, il est préférable qu'ils accèdent au serveur Eureka le + proche.

Il faut alors indiquer dans quelle zone se trouve le client

```
eureka.instance.metadataMap.zone = zone1  
eureka.client.preferSameZoneEureka = true
```



# Serveur de Discovery et Configuration

---

Service de Discovery, l'exemple Eureka  
**Spring Cloud Config**



# Configuration centralisée

---

***spring-cloud-config*** offre une alternative pour mettre en place une gestion centralisée de toutes les configurations des micro-services.

- Les mises à jour peuvent être dynamiques
- Le serveur peut être associé à un dépôt Git
  - => Gestion de branches, des releases



# Serveur de configuration

---

Le serveur de configuration est une application Spring Boot annotée par **@EnableConfigServer**

La stratégie par défaut pour localiser les sources des propriétés est de cloner un dépôt git (*spring.cloud.config.server.git.uri*)

Le serveur offre différents endpoints permettant de récupérer les valeurs de configuration :

`/ {application} / {profile} [ / {label} ]`

`/ {application} - {profile} . yml`

`...`



# Résolution des valeurs de configuration

---

Les valeurs de configurations retournées par les endpoints dépendent de 3 paramètres :

- ***{application}*** qui correspond à la propriété "*spring.application.name*" du côté du client
- ***{profile}*** qui correspond à "*spring.profiles.active*" du côté du client
- ***{label}*** qui correspond à une révision ou tag du côté du serveur





# Mise en place

---

**@SpringBootApplication**

**@EnableConfigServer**

```
public class ConfigServer {  
    public static void main(String[] args) {  
        SpringApplication.run(ConfigServer.class, args);  
    }  
}
```

Propriétés principales (*application.properties*)

server.port: 8888

spring.cloud.config.server.git.uri: [file://\\${user.home}/config-repo](#)

Dépendance

```
<dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-config-server</artifactId>  
</dependency>
```



# Backend Git

---

L'implémentation par défaut pour stocker les valeurs de configuration est un dépôt Git spécifié par :

***spring.cloud.config.server.git.uri***

Le répertoire de clonage est spécifié par :

***spring.cloud.config.server.git.basedir***

Le paramètre *{label}* est alors associé à une référence *git* (id de commit, branche ou tag)



# Sécurité

---

Pour utiliser l'authentification basique HTTP, il suffit de préciser les propriétés :

- *spring.cloud.config.server.git.username*
- *spring.cloud.config.server.git.password*

Pour utiliser *ssh*, le plus simple est d'installer une clé publique sur le serveur



# Backend de fichiers

---

Le serveur peut également charger les configurations à partir du classpath ou du système de fichiers

Il faut alors

- activer le profil ***native***
- et d'indiquer la propriété ***spring.cloud.config.server.native.searchLocations*** qui spécifie l'emplacement racine
  - soit à partir du classpath  
*classpath:/config*
  - Soit à partir du système de fichiers  
*file:./config*



# Exemple Classpath

---

```
@SpringBootApplication
```

```
@EnableConfigServer
```

```
public class ConfigApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(ConfigApplication.class, args);  
    }  
}
```

```
---
```

```
spring:  
  cloud:  
    config:  
      server:  
        native:  
          search-locations: classpath:/shared  
profiles:  
  active: native
```



# Arborescence serveur

---

Le répertoire racine sur le serveur contient alors :

- ***application.yml*** : Valeurs de configuration partagées par tous les micro-services
- Des fichiers ***<application-name>.yml*** : Valeurs de configuration pour le micro-service *<application-name>*
- ***<application-name>-<profil>.yml*** : Des fichiers spécifiques à un profil<sup>1</sup>.

1. Les profils peuvent également être spécifiés dans *<application-name>.yml*



# Endpoints proposés

---

Une fois démarré le serveur propose les endpoints suivants :

- ***/application/default*** : Les propriétés partagées par tous les micro-services
- ***/application-name/default*** : Les propriétés d'un service donné dans le profil par défaut
- ***/application-name/profil*** : Les propriétés d'un service donné dans un profil donné.



# Usage client

---

Les clients dépendent de :

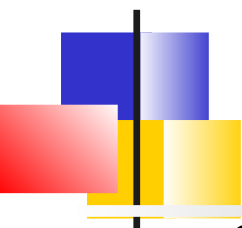
***spring-cloud-config-client***

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-config</artifactId>  
</dependency>
```

Ils définissent l'adresse du serveur de configuration  
(par défaut localhost:8888) via la propriété :

*spring.cloud.config.uri: http://myconfigserver.com*





# Configurations optionnelles

---

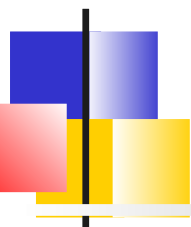
Si l'on veut interdire le démarrage en cas de non contact du serveur de configuration :

```
spring.cloud.config.failFast=true
```

Si on veut autoriser plusieurs tentatives de connexions :

- spring-retry et spring-boot-starter-aop dans le classpath
- puis les propriétés `spring.cloud.config.retry.*`

L'emplacement par défaut des configurations  
(/{name}/{profile}/{label}) peut être surchargé par  
`spring.cloud.config.*`



# Propriétés des micro-services

---

Les propriétés de configuration des micro-services sont donc présents à 2 endroits :

- ***bootstrap.yml*** : Présent en local, il ne contient en général que :
  - *spring.application.name*
  - La configuration du serveur de config
- Fichiers sur le serveur de config :
  - propriétés communes des micro-services,
  - propriétés spécifiques application
  - spécifique profil



# Changement dynamique d'environnement

---

Si la configuration est modifiée (fichier de config modifié ou *push git*), celle-ci peut être répercutée dynamiquement sur les clients du serveur de config

Lors d'une modification, un événement de type *EnvironmentChangeEvent* est généré

- Il contient la liste des clés de configuration qui ont changées.
- Les clients ne détectent pas automatiquement les changements. On peut alors utiliser :
  - **Spring Cloud Bus** pour pousser les événements automatiquement vers les services
  - **Actuator** et une requête POST pour demander le rafraîchissement d'un micro-service



# Utilisation d'Actuator

---

Le endpoint refresh d'actuator doit être activée.

Une requête POST sur ***/actuator/refresh*** permet de à certains beans Spring de rechercher leur configuration.

- Les beans applicatifs voulant se recharger doivent être annotés par ***@RefreshScope***



# Support pour les clients

---

## **Répartition de charge**

Pattern Disjoncteur

Clients déclaratifs avec Feign

Gateway et routing

*Consumer Driven Contract*



# Introduction

---

Les micro-services pouvant être répliqués, la charge doit être répartie sur les différentes répliques.

2 patterns sont possibles pour la répartition de charge :

- Client-Side : Le client connaît le emplacements des répliques et peut ainsi répartir la charge
- Server-side : Le client s'adresse à un point unique : le load balancer qui lui connaît l'emplacement des répliques



# Client-side et Discovery-Client

---

Avec l'API *DiscoveryClient* de SpringCloud, la répartition client-side est naturelle.

Elle peut être implémentéeE

- En utilisant l'API *DiscoveryClient*
- En utilisant l'abstraction *spring-cloud-loadbalancer* de SC Commons



# Implémentation via *DiscoveryClient*

---

```
@Autowired
```

```
DiscoveryClient discoveryClient;
```

```
...
```

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

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

```
restTemplate.rootUri(url);
```

```
...
```



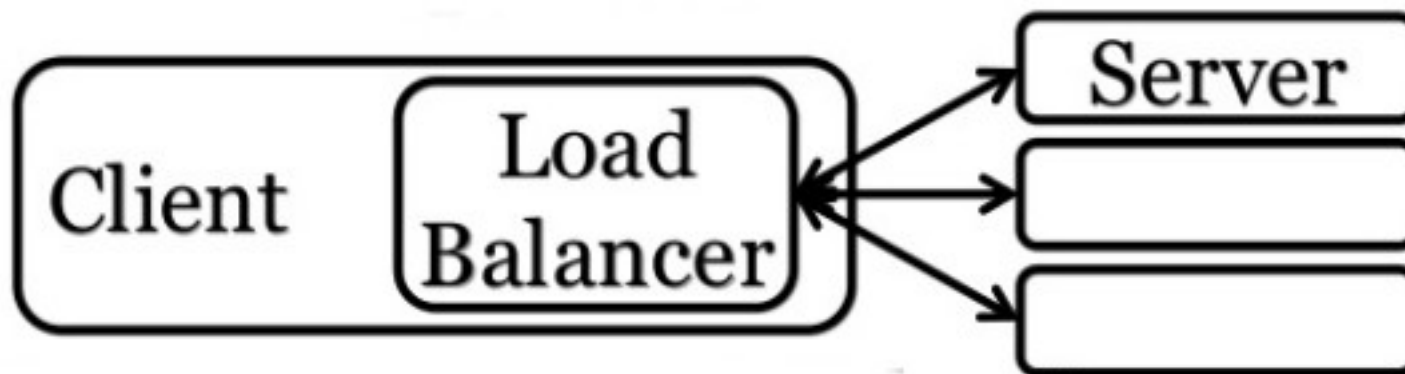


# Spring Cloud LoadBalancer

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

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

- Il utilise le bean *DiscoveryClient* disponible dans le classpath
- Il peut également utiliser une librairie de cache





# 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*

---

Dans un contexte non-réactif, 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* .

```
@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()));
    }
}
```



# Support pour les clients

---

Répartition de charge

**Pattern Disjoncteur**

Clients déclaratifs avec Feign

Gateway et routing

*Consumer Driven Contract*



# 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.<sup>1</sup>*





# 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



# Example

---

```
@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.*



# Support pour les clients

---

Répartition de charge

Pattern Disjoncteur

**Clients déclaratifs avec Feign**

Gateway et routing

*Consumer Driven Contract*



# Introduction

---

***Feign*** est un client de service REST déclaratif, facilitant le développement.

Pour utiliser *Feign*, juste créer une interface et l'annoter !

Il offre un support pour des annotations pluggable, les annotations JAX-RS, des dés/encodeurs pluggable



# Mise en place

---

**spring-cloud-starter-feign**

```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
public class Application {
```

```
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
```

```
}
```





# Client

---

```
// « stores » : client nommé correspondant au service
// Eureka
@FeignClient("stores")
public interface StoreClient {
    @RequestMapping(method = RequestMethod.GET, value = "/stores")
    List<Store> getStores();

    @RequestMapping(method = RequestMethod.POST, value =
"/stores/{storeId}", consumes = "application/json")
    Store update(@PathVariable("storeId") Long storeId, Store store);
}
```



# CircuitBreaker avec *Feign*

---

Si Spring Cloud CircuitBreaker est dans le classpath et que ***feign.circuitbreaker.enabled=true***, les appels sont encapsulés dans un circuit breaker

Le nom du circuit-breaker est alors :  
*<feignClientClassName>#<calledMethod>(<parameterTypes>)*

Pour configurer la commande de fallback il faut renseigner, l'attribut ***fallback*** en indiquant un bean implémentant l'interface



# Exemple avec *Feign* et *fallback*

---

```
@FeignClient(name = "hello", fallback = ClientFallback.class)
protected interface Client {
    @RequestMapping(method = RequestMethod.GET, value = "/hello")
    Hello iFailSometimes();
}
```

## **@Service**

```
class ClientFallback implements Client {
    @Override
    public Hello iFailSometimes() {
        return new Hello("fallback");
    }
}
```



# Support pour les clients

---

Répartition de charge  
Pattern Disjoncteur  
Clients déclaratifs avec Feign  
**Gateway et routing**  
*Consumer Driven Contract*



# 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 Gateway

---

Spring Cloud Gateway, construit sur un modèle réactif, a pour vocation de fournir toutes les fonctionnalités d'une Gateway :

- Routage en fonction de n'importe quel attribut de requête.
- Les filtres spécifiques aux routes.
- Intégration de CircuitBreaker
- Intégration de 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*

---

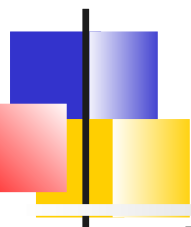
Pour créer des routes vis à vis des services enregistrés dans le serveur de discovery :

```
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





# Définition explicite des routes

---

En général, on ne veut pas exposer tous les micro-services à l'extérieur ; il faut donc configurer les routes explicitement.

Ex avec *DiscoveryClient* :

```
spring:
  cloud:
    gateway:
      locator:
        enabled : false
      routes:
        - id: route-order
          uri: lb://ORDERSERVICE
          predicates:
            - Path= /order/**
          filters:
            - RewritePath=/order/(?<remaining>.*), /${remaining}
```



# Exemple CircuitBreaker

---

Dans le classpath :

`spring-cloud-starter-circuitbreaker-reactor-resilience4j`

Config :

`spring:`

`cloud:`

`gateway:`

`routes:`

- `id: circuitbreaker_route`

`uri: https://example.org`

`filters:`

- `name: CircuitBreaker`

`args:`

`name: myCircuitBreaker`

`fallbackUri: forward:/inCaseOfFailureUseThis`

- `RewritePath=/consumingSE, /backingSE`



# Actuator

---

Pour avoir un endpoint actuator :

```
management.endpoint.gateway.enabled=true # default value  
management.endpoints.web.exposure.include=gateway
```

Ensuite, les configurations de routes sont  
accessibles à l'URL  
`/actuator/gateway/routes`



# Support pour les clients

---

Répartition de charge  
Pattern Disjoncteur  
Clients déclaratifs avec Feign  
Gateway et routing  
***Consumer Driven Contract***



# Spring Cloud Contract

---

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

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

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



# DSL

---

*Spring Cloud Contract* et son projet principal **Verifier** fournit un **DSL** (*Groovy* ou *.yaml*) qui permet de spécifier le contrat

Le contrat permet alors de produire :

- Les tests d'acceptations complets côté serveurs (*Spock* ou *JUnit*)
- Les définitions de stub JSON à utiliser par *WireMock* lors des tests d'intégration sur le code client.  
Le code de test doit toujours être écrit à la main mais les données de test sont produites.

SCC permet également les tests de routing de message lors de l'utilisation de *Spring Cloud Stream*



# 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
    }
}
```



# Tests générés côté producteur

---

```
public class ContractVerifierTest extends BaseTestClass {

    @Test
    public void validate_shouldReturnEvenWhenRequestParamIsEven() throws Exception {
        // given:
        MockMvcRequestSpecification request = given();

        // when:
        ResponseOptions response = given().spec(request)
            .queryParams("number", "2")
            .get("/validate/prime-number");

        // then:
        assertThat(response.statusCode()).isEqualTo(200);

        // and:
        String responseBody = response.getBody().asString();
        assertThat(responseBody).isEqualTo("Even");
    }
}
```





# Tests Consommateur

---

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

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void given_WhenPassEvenNumberInQueryParam_ThenReturnEven()
        throws Exception {

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



# Monitoring

---

## **Metriques CircuitBreaker**

Tracing avec *Spring Cloud Sleuth*

Centralisation des logs



# Introduction

---

Pour autoriser la collecte de métrique de Resilience4J, il faut les starters :

`org.springframework.boot:spring-boot-starter-actuator`  
`io.github.resilience4j:resilience4j-micrometer`

Les métriques sont typiquement envoyés via *Micrometer* sur des systèmes comme *InfluxDB* ou *Prometheus* puis visualisés avec des outils de visualisation tels que *Grafana*



# Métriques exportés

---

***resilience4j.circuitbreaker.state*** : L'état du circuit

***resilience4j.circuitbreaker.failure.rate*** : La cadence d'erreur

***resilience4j.circuitbreaker.slow.call.rate*** : La cadence d'appels lents

***resilience4j.circuitbreaker.calls*** : Nombre total d'appels réussis, échoués ou ignorés

***resilience4j.circuitbreaker.not.permitted.calls*** : Le nombre total d'appels non autorisés



# Envoi vers prometheus

---

Pour publier les métriques vers prometheus, il suffit d'ajouter la dépendance :

`io.micrometer:micrometer-registry-prometheus`



# Monitoring

---

Metriques CircuitBreaker  
**Tracing avec Spring Cloud Sleuth**  
Centralisation des logs



# Introduction

---

***Spring Cloud Sleuth*** permet d'ajouter dans les messages de logs des *ids* permettant de tracer un séquence d'appels de méthodes correspondant à une requête HTTP, le démarrage d'un job, une thread ...

- Il s'intègre sans effort avec *Logback* et *SLF4J* et utilise la librairie de tracing *Brave* .

Dans le cadre d'une architecture micro-services, *Sleuth* sert à tracer le cheminement d'une requête à travers tous les micro-services qu'elle a invoqué.



# Vocabulaire

---

Sleuth définit 2 notions :

- **Traces** identifiant une requête unique ou le démarrage d'un job. Toutes les différentes étapes de traitement partageront le même *traceld*.
- **Spans**, identifie une étape de traitement. Une trace peut donc être composée de plusieurs spans. *Sleuth* ajoute un *spanId* pour chaque étape de traitement

En utilisant ces Ids on peut facilement isoler les traces relatives à une requête particulière.





# Mise en place

---

Dépendance :

`org.springframework.cloud:spring-cloud-starter-sleuth`

Impact sur les traces de l'appli :

2017-01-10 22:36:38.254 INFO

`[SpringApplicationName, 4e30f7340b3fb631, 4e30f7340b3fb631, false]` 12516

--- [nio-8080-exec-1] c.b.spring.session.SleuthController : Hello Sleuth

2017-01-10 22:51:48.664 INFO

`[SpringApplicationName, 4e30f7340b3fb631, 4e30f7340b3fb631, false]` 12516

--- [nio-8080-exec-1] c.baeldung.spring.session.SleuthService : Doing some work

*[nom application, traceId, spanId, export]*

- Nom application : Propriété *spring.application.name*.
- TraceId : id affecté à une requête, un job.
- SpanId : Identifie une unité de travail. Pour la première étape *TraceId=SpanId*.
- Export : Booléen qui indique si cette trace a été exporté vers un agrégateur comme *Zipkin*.



# Ajout manuel de spans

---

Il est possible de manuellement créer des nouveaux spans dans une application.

```
@Autowired
private Tracer tracer;
// ...
public void doSomeWorkNewSpan() throws InterruptedException {
    logger.info("I'm in the original span");

    Span newSpan = tracer.newTrace().name("newSpan").start();
    try (SpanInScope ws = tracer.withSpanInScope(newSpan.start())) {
        Thread.sleep(1000L);
        logger.info("I'm in the new span doing some cool work that needs its own span");
    } finally {
        newSpan.finish();
    }
    logger.info("I'm in the original span");
}
```



# Ajout manuel de spans (2)

---

Les traces deviennent.

```
2017-01-11 21:07:54.924
```

```
INFO [SpringApplicationName,9cdebbffe8bbbade,9cdebbffe8bbbade,false] 12516
```

```
--- [nio-8080-exec-6] c.b.spring.session.SleuthController      : New Span
```

```
2017-01-11 21:07:54.924
```

```
INFO [SpringApplicationName,9cdebbffe8bbbade,9cdebbffe8bbbade,false] 12516
```

```
--- [nio-8080-exec-6] c.baeldung.spring.session.SleuthService :
```

```
I'm in the original span
```

```
2017-01-11 21:07:55.924
```

```
INFO [SpringApplicationName,9cdebbffe8bbbade,1e706f252a0ee9c2,false] 12516
```

```
--- [nio-8080-exec-6] c.baeldung.spring.session.SleuthService :
```

```
I'm in the new span doing some cool work that needs its own span
```

```
2017-01-11 21:07:55.924
```

```
INFO [SpringApplicationName,9cdebbffe8bbbade,9cdebbffe8bbbade,false] 12516
```

```
--- [nio-8080-exec-6] c.baeldung.spring.session.SleuthService :
```

```
I'm in the original span
```



# Zipkin

---

**Zipkin** est un projet open-source qui permet de recevoir et visualiser des traces.

Il est disponible :

- Sous forme de jar
- Ou d'image docker  
*openzipkin/zipkin*



# Architecture

---

Chaque micro-service exporte ses spans vers le serveur *Zipkin*

Le serveur les stocke et propose une interface permettant de consulter les traces

# Interface Zipkin

localhost:9411

Zipkin Investigate system behavior Find a trace Dependencies Go to trace

Service Name all Start time 02-08-2017 20:53

End time 02-15-2017 20:53 Duration (µs) >= Limit 10 Find Traces ?

Annotations Query (e.g. "finagle.timeout", "http.path=/foo/bar/ and cluster=foo and cache.miss")

Please select the criteria for your trace lookup.

Service Name all Start time 02-08-2017 20:53

End time 02-15-2017 20:53 Duration (µs) >= Limit 10 Find Traces ?

Annotations Query (e.g. "finagle.timeout", "http.path=/foo/bar/ and cluster=foo and cache.miss")

Please select the criteria for your trace lookup.



# Configuration des services

---

Dans un contexte SpringCloud, chaque service doit ajouter la dépendance :

`org.springframework.cloud:spring-cloud-starter-zipkin`

Et doit déclarer dans sa configuration l'URL Zipkin

```
spring:
  zipkin:
    base-url: http://zipkin:9411/
  sleuth:
    sampler:
      probability: 1
```



# Monitoring

---

Metriques CircuitBreaker  
Tracing avec Spring Cloud Sleuth  
**Centralisation des logs**





# Gestion des logs

---

Une préconisation pour le développement des micro-services est qu'il ne se préoccupe jamais du routage ou du stockage de son flux de sortie.  
=> Ne doit pas tenter d'écrire ou de gérer des fichiers journaux.

Au lieu de cela, chaque micro-service écrit son flux d'événements sur stdout.

En production, chaque flux est capturé par l'environnement d'exécution, assemblé avec tous les autres flux de la stack et acheminé vers une ou plusieurs destinations finales pour visualisation et archivage à long terme.



# Elastic Stack

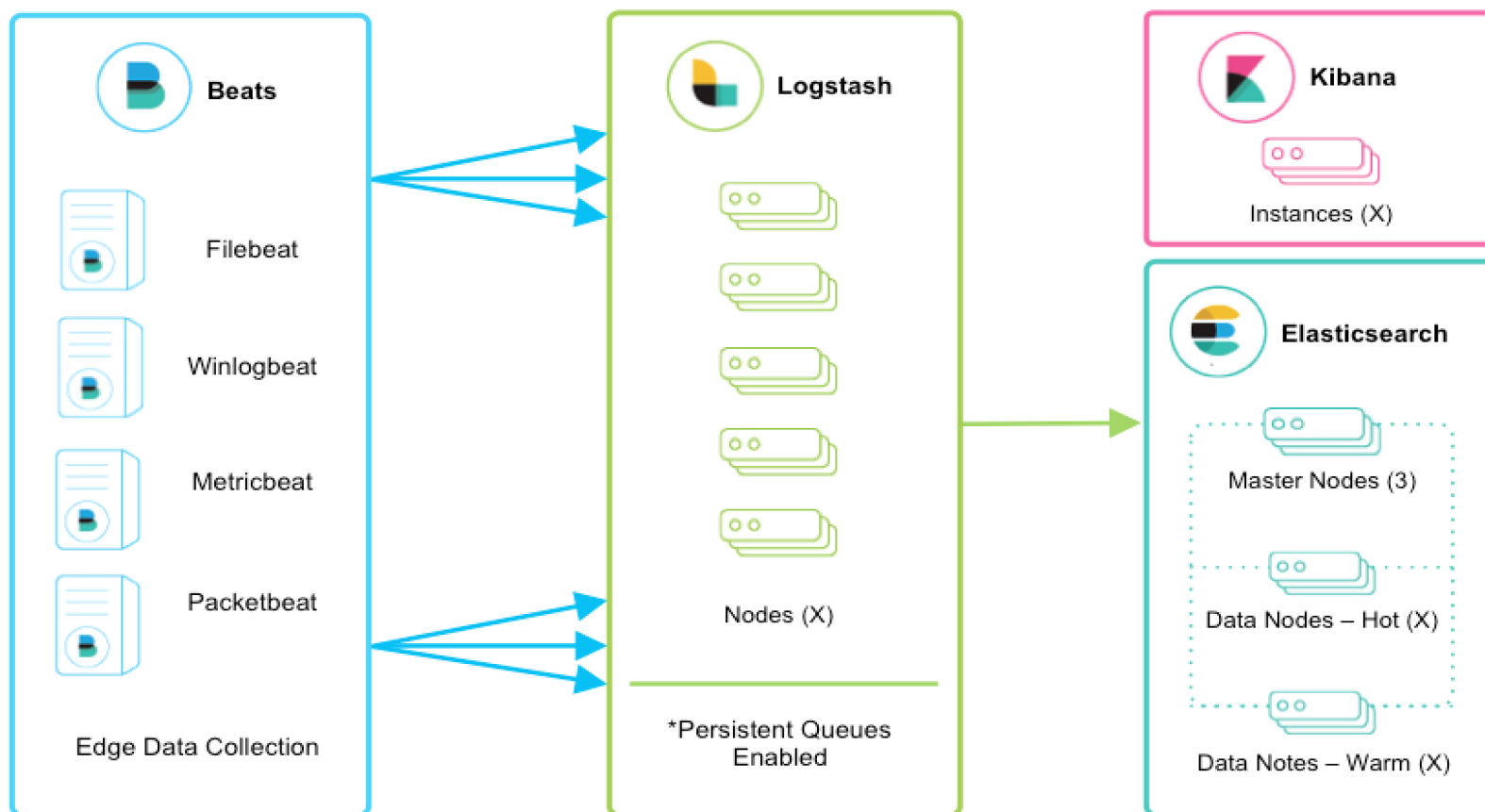
---

Une des solutions les plus courantes pour centraliser, visualiser et analyser les traces est la solution *ElasticStack*

Cette pile est constituée de :

- **Beats** : convoyeurs de données installés avec les applications SpringBoot
- **Logstash** : Micro-service chargé de normaliser les différentes traces et de les soumettre à un ElasticSearch
- **ElasticSearch** : Micro-service permettant d'indexer les traces et offrant des fonctionnalités de recherche et d'agrégation
- **Kibana** : Application front-end offrant une UI pour l'analyse

# Architecture en cluster





# Messaging

---

## **Communication asynchrone**

Exemple : Saga Pattern

Spring Cloud Stream

Spring Cloud Data Flow



# Introduction

---

Les communications asynchrones entre les micro-services apportent plusieurs avantages :

- Découplage du producteur et consommateur de message
- Scaling et montée en charge
- Implémentation de patterns de micro-services  
Saga<sup>1</sup>, Event-sourcing Pattern<sup>2</sup>

Des difficultés :

- Gestion de l'asynchronisme
- Mise en place et exploitation d'un message broker

1. <https://microservices.io/patterns/data/saga.html>

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



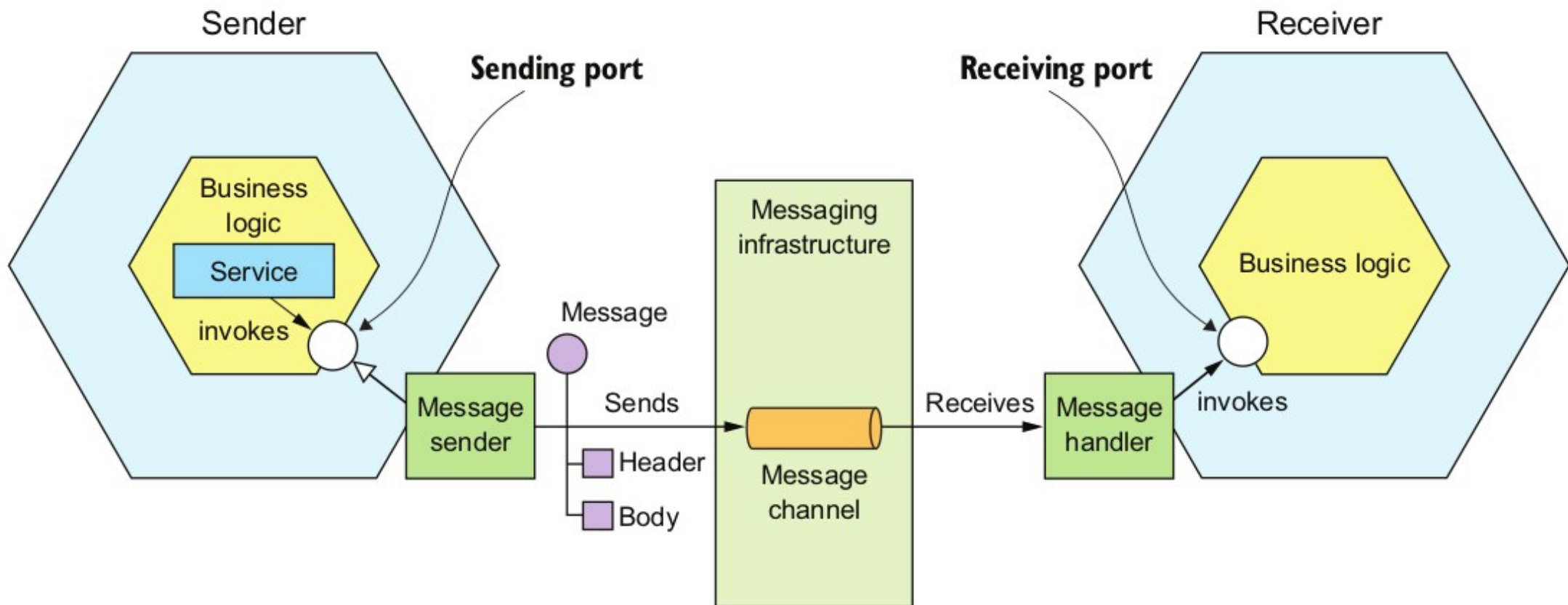
# *Messaging Pattern*

---

***Messaging Pattern***<sup>1</sup> : 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

# Architecture





# Message

---

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

On distingue 3 types de messages :

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





# Canaux de messages

---

2 types de canaux :

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



# Styles d'interaction

---

Tous les styles d'interactions sont supportés :

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



# Spécification de l'API

---

La spécification consiste à définir

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

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



# Message Broker

---

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

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

Implémentations courantes :

- ActiveMQ
- RabbitMQ
- Kafka
- AWS Kinesis



# Facteurs de choix (1°)

---

## Langages de programmation supportés

C'est mieux si il en supporte plusieurs

## Standard de messaging supportés

AMQP, STOMP ou propriétaire

## Ordre des messages

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

## Garanties de livraison

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

## Persistance

Les messages survivent-ils au crash ?



# Facteurs de choix (2)

---

## Durabilité

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

## Scalabilité

Le broker est-il scalable ?

## Latence

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

## Consommation concurrente

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



# Offre Spring

---

Starter messaging pur :

- RabbitMQ, ActiveMQ, Kafka, ActiveMQ Artemis, Solace PubSub

Pipeline de traitement d'évènements :

- Kafka Stream

Architecture micro-services *event-driven*

- Spring Cloud Stream
- Spring Data Flow



# Messaging

---

Communication asynchrone

**Example : Saga Pattern**

Spring Cloud Stream

Spring Cloud Data Flow





# Exemple : Gestion des transactions avec Saga

---

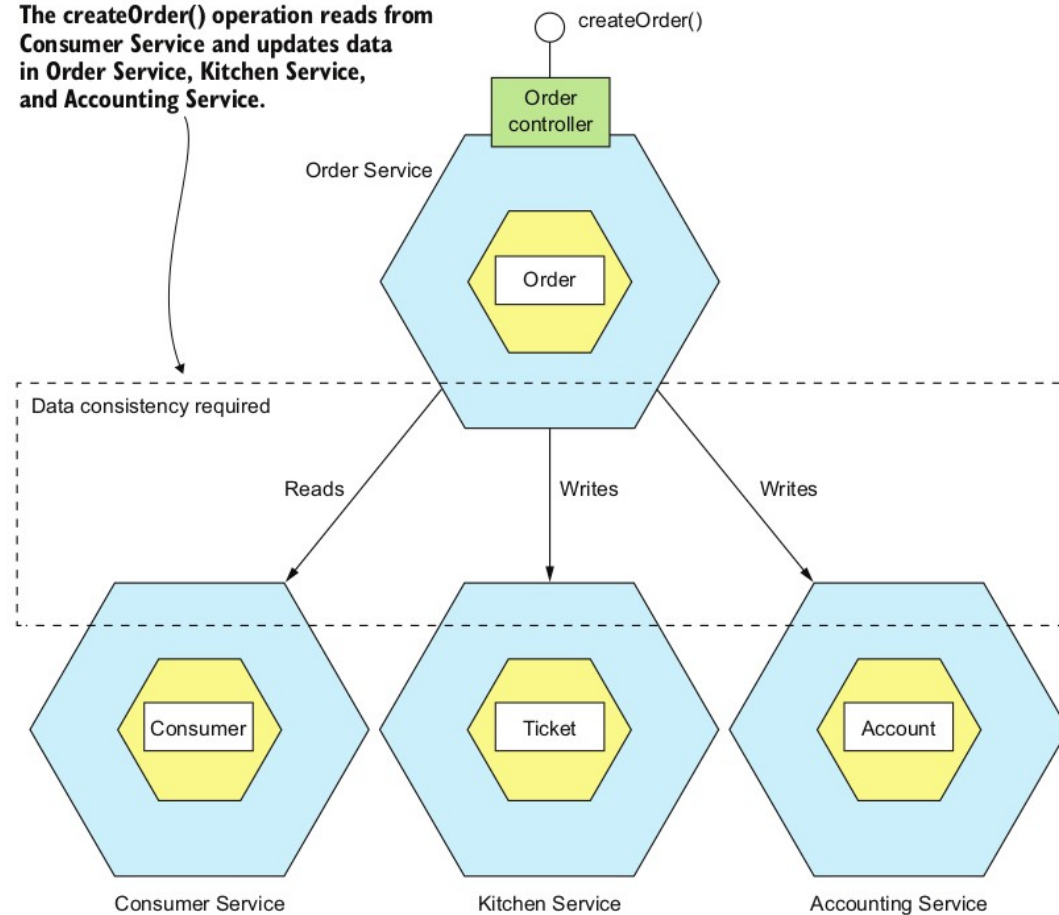
Problème : Comment implémenter des transactions qui englobent plusieurs services, i.e des opérations qui mettent à jour des données dispersées dans plusieurs services

Solution : Une **saga**, séquence de transactions locales basée sur des messages. Cette séquence présente les propriétés ACD (Atomicité, Cohérence, Durabilité) mais la propriété d'isolation n'est pas respectée<sup>1</sup>.

1. En conséquence, l'application doit utiliser des contre-mesures (opération de compensation) 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 transaction et sont assurés de réussir



# Exemple

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

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



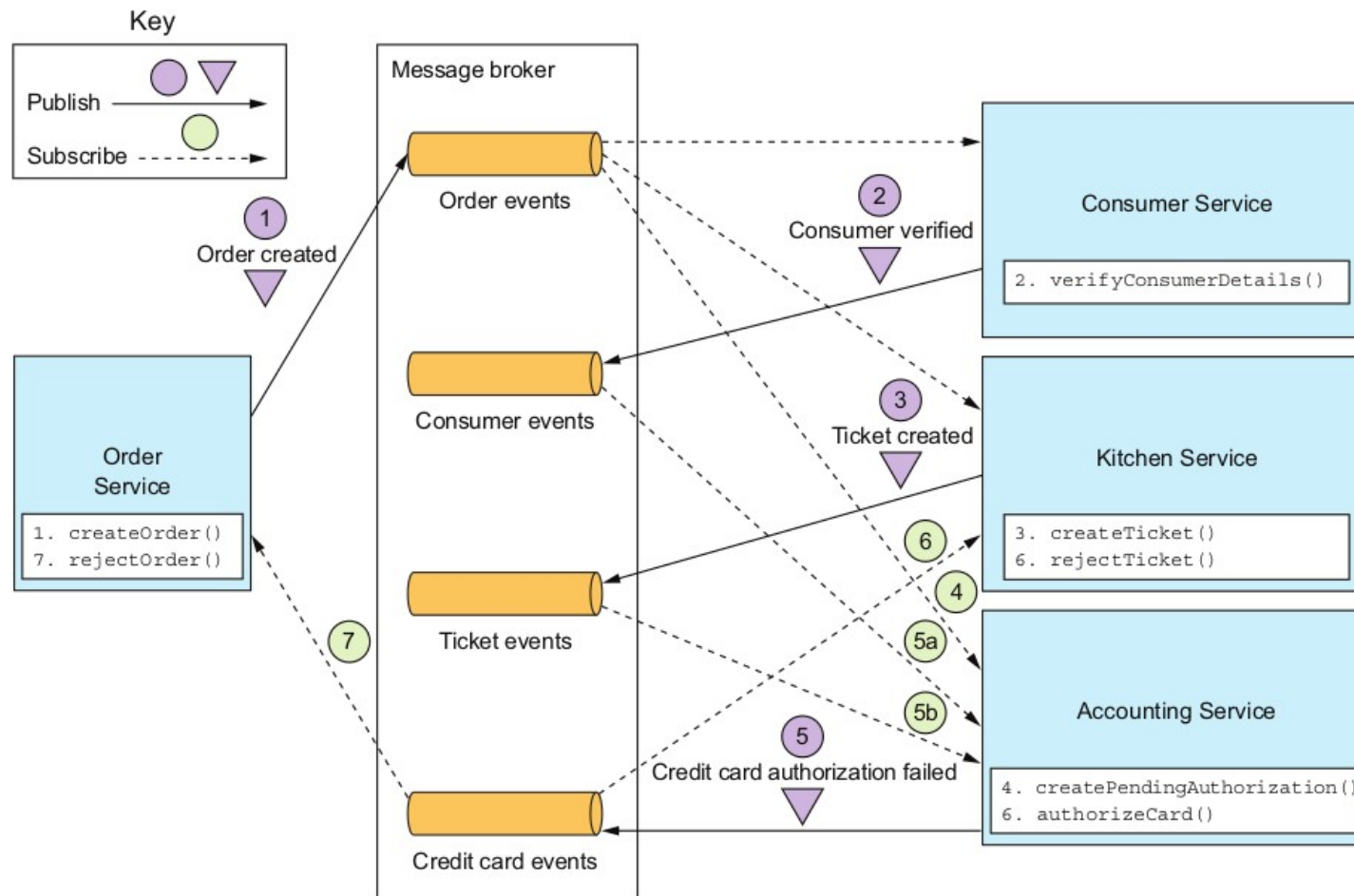
# Implémentation

---

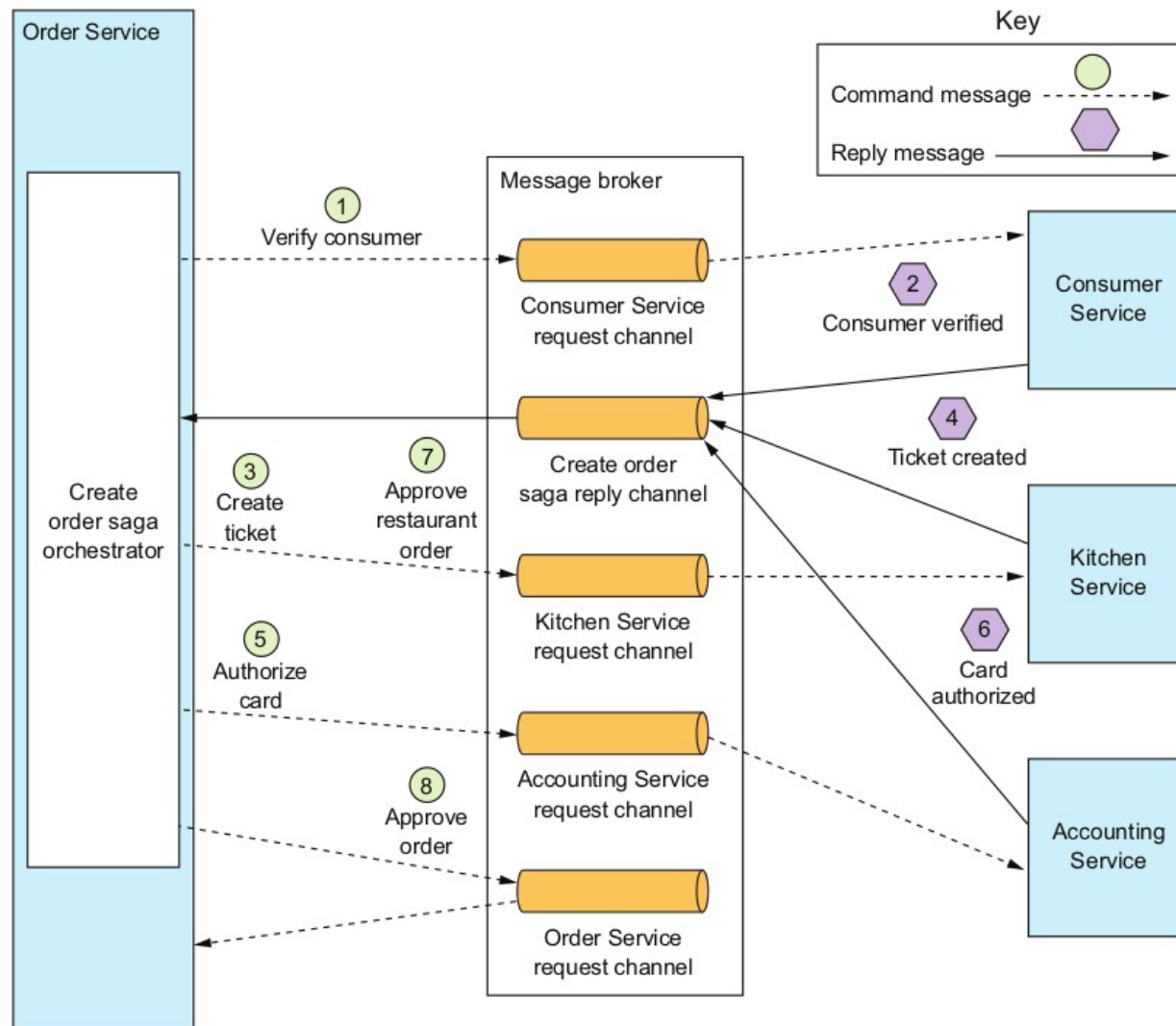
Il existe 2 façons de coordonner une saga:

- **chorégraphie** - les participants à la saga échangent des événements
- **orchestration** - un orchestrateur centralisé utilise une messagerie asynchrone pour piloter les participants

# Chorégraphie



# Orchestration





# Exemple *spring-kafka*

---

## Envoi de message

```
@Value("${app.my-channel}")
String PAYMENT_REQUEST_CHANNEL;

@Autowired
KafkaTemplate<Long, DomainEvent> kafkaOrderTemplate;

public Order doService(Domain model) {
    ...
    DomainEvent event = new DomainEvent(model);
    kafkaOrderTemplate.send(ORDER_STATUS_CHANNEL, event);
    ...
}
```

## Réception de message :

```
@KafkaListener(topics = "#{ '${app.my-channel}' }", id = "oneHandler")
public void handleEvent(DomainEvent domainEvent) {
    ...
}
```





# Messaging

---

Communication asynchrone

Exemple : Saga Pattern

**Spring Cloud Stream**

Spring Cloud Data Flow

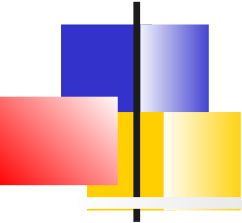


# Introduction

---

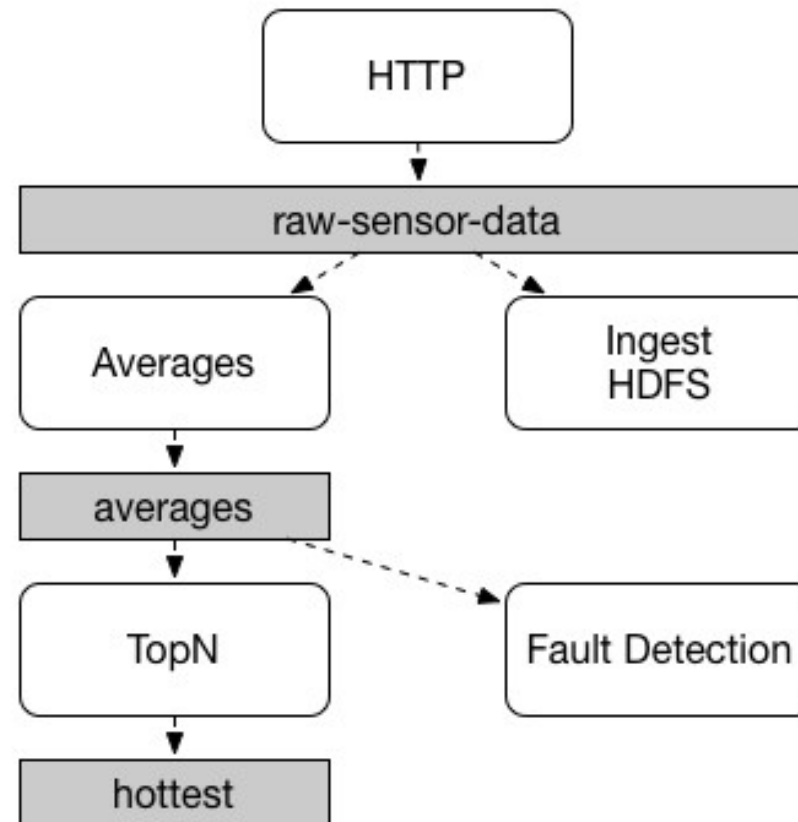
***Spring Cloud Stream*** est le framework pour construire des applications micro-service pilotées par des flux de messages.

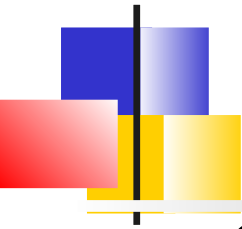
- Utilise *Spring Boot* et *Spring Integration* pour la connectivité avec les messages brokers
- Repose sur les concepts de
  - *topic persistant (publish/subscribe)*,
  - groupe de consommateurs
  - et de partitions



# Exemple d'architecture

---





# Utilisation

---

Starter : ***spring-cloud-stream***

- Une ou plusieurs **interfaces** déclarant les canaux d'entrée et/ou de sortie
- **@EnableBinding** pour auto-configurer la connectivité des canaux avec le système de messagerie sous-jacent
- **@StreamListener** sur les méthodes voulant réagir à un flux
- Utilisation de l'API **MessageChannel** pour envoyer des messages



# Exemple

---

```
@SpringBootApplication
@EnableBinding(Sink.class)
public class VoteRecordingSinkApplication {

    public static void main(String[] args) {
        SpringApplication.run(VoteRecordingSinkApplication.class, args);
    }

    @StreamListener(Sink.INPUT)
    public void processVote(Vote vote) {
        votingService.recordVote(vote);
    }
}

-----
public interface Sink {
    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();
}
```



# Concepts principaux

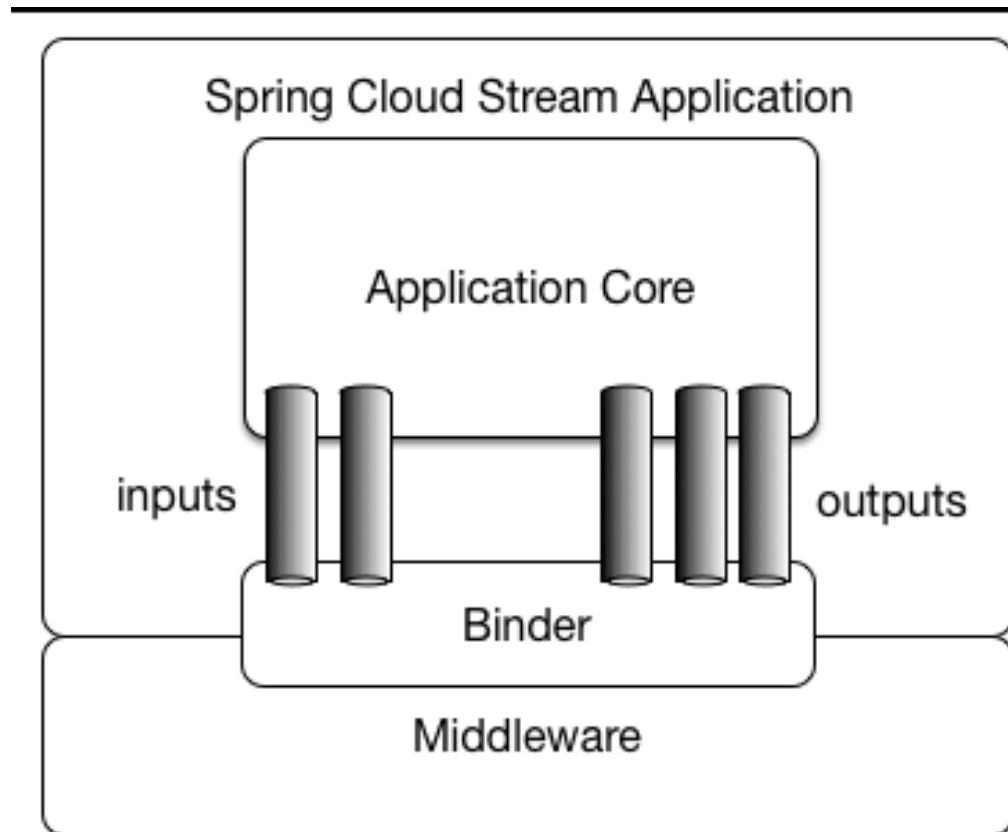
---

Les applications Cloud Stream se basent sur la présence d'un **middleware neutre**

Elles communiquent avec l'extérieur à travers des **canaux** (channels) d'entrée /sortie

Les canaux sont associés à des systèmes de messagerie (Kafka, RabbitMQ, ...) à travers des **binders spécifiques** à l'implémentation

# Architecture





# Publish & Subscribe

---

La communication entre applications suit un modèle ***publish-subscribe***, les données sont alors diffusées à travers des topics partagés

La notion de **groupe de consommateurs** permet la scalabilité

- Un message n'est reçu que par un seul membre du groupe

Par défaut, les topics sont **durables**.

Ce qui garantit la livraison des messages mêmes si les consommateurs sont arrêtés au moment de l'émission





# Partitionnement

---

Spring Cloud Stream supporte le **partitionnement** des messages

Cela garantit que les messages identifiés par une caractéristique commune seront traités par la même instance de consommateur (Modèle *stateful*)



# Modèle de programmation

---

**@EnableBinding** configure la connectivité

**@Input** et **@Output** dans des interfaces définissent les canaux d'entrée.

Spring fournit 3 interfaces. *Source* : 1 seul canal de sortie, *Sink* 1 seul canal d'entrée et *Processor* les 2 types de canaux

Les implémentations de ces interfaces ou directement les canaux peuvent être **injectées** permettant l'envoi de message par exemple

Les annotations de Spring Integration ou **@StreamListener** (plus simple) peuvent être utilisées pour traiter les messages

Également du support pour des API réactive, ou l'agrégation d'application (plus de messagerie)



# Messaging

---

Saga Pattern  
Spring Cloud Stream  
**Spring Cloud Data Flow**



# Introduction

---

Domaine d'usage : Traitement de données par flux et traitement par lots basé sur des Microservices pour Cloud Foundry et Kubernetes.

Spring Cloud Data Flow fournit des outils permettant de créer des pipelines des traitement de données par lots.

- Les pipelines sont constitués d'applications Spring Boot, construites à l'aide de Spring Cloud Stream ou Spring Cloud Task.

Cas d'utilisation : ETL, import/export, traitement en continu d'évènements, analyse prédictive, ...



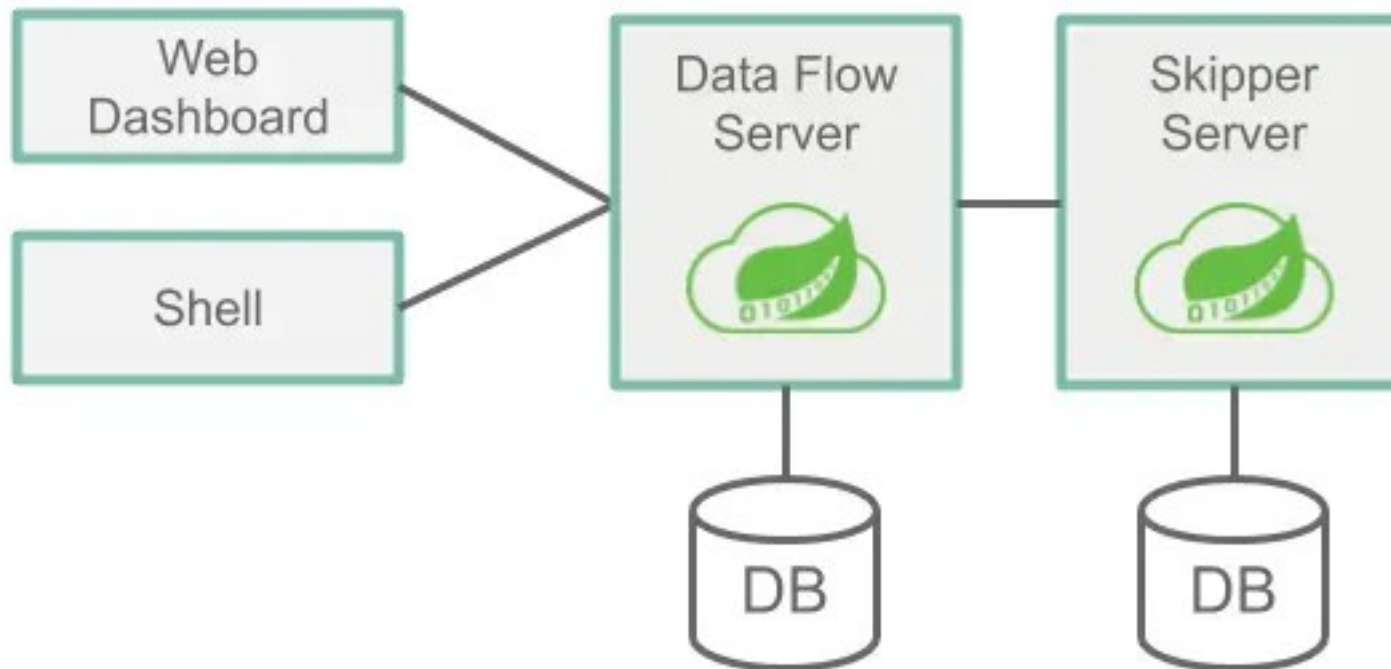
# Architecture

---

L'architecture de l'offre est composée de 2 composants :

- ***Data Flow Serveur*** : Contient la bibliothèque des micro-services de base et les pipelines de traitement
- ***Skipper Serveur*** : S'occupe des déploiements dans les environnements Cloud à partir de repository d'artefact

# Architecture





# DataFlow Serveur

---

- Parse, valide, aide à l'édition et stocke les définitions de flux et de travaux batch exprimées via un DSL
- Stocke les artefacts .jar ou images docker
- Définit les propriétés de configuration des artefacts (flux d'entrées/sorties flux, propriétés de déploiements telles que le nombre initial d'instances, les besoins en mémoire et le partitionnement des données).
- Démarre les déploiements de travaux batch et délègue la planification des travaux
- Délégation du déploiement de flux à Skipper.
- Offre une historique d'exécution et un audit



# Skipper serveur

---

Déploiement de flux sur une ou plusieurs plates-formes.

Mise à niveau et restauration du flux sur une ou plusieurs plates-formes à l'aide d'une stratégie de mise à jour blue/green basée sur une machine d'état.

Stockage de l'historique du fichier manifeste de chaque flux (qui représente la description finale des applications déployées).





# Cloud Security

---

## **Introduction**

Rappels OAuth2  
Spring Boot et OAuth2  
Spring Cloud Security



# Introduction

---

Plusieurs approches pour sécuriser une architecture micro-services :

- N'implémenter la sécurité qu'au niveau du proxy. Les micro-services back-end ne sont pas protégés
- Chaque micro-service a sa propre politique de sécurité. Un jeton d'accès relayé permet de vérifier les ACLs
- Chaque micro-service a sa propre politique de sécurité et chaque micro-service demande son propre jeton pour effectuer ses appels REST

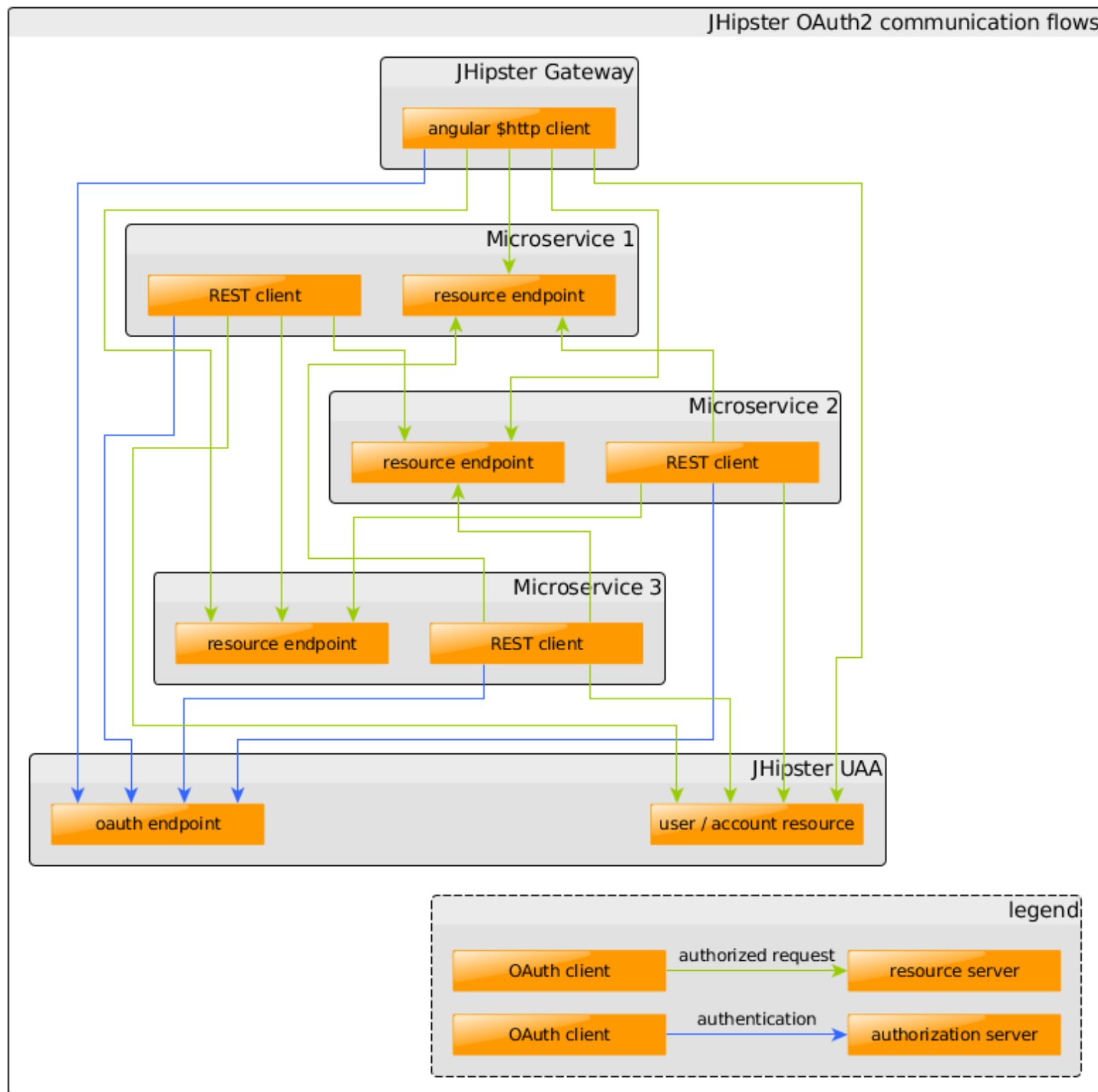


# Modèle micro-service

---

Dans un modèle micro-service, les différents micro-services peuvent être clients et/ou serveur de ressources *oAuth*.

L'identité d'une requête et ses permissions peuvent varier en fonction des micro-services qu'elle traverse





# Cloud Security

---

Introduction

**Rappels OAuth2/OpenID**

Spring Boot et OAuth2

Spring Cloud Security



# Rôles du protocole

---

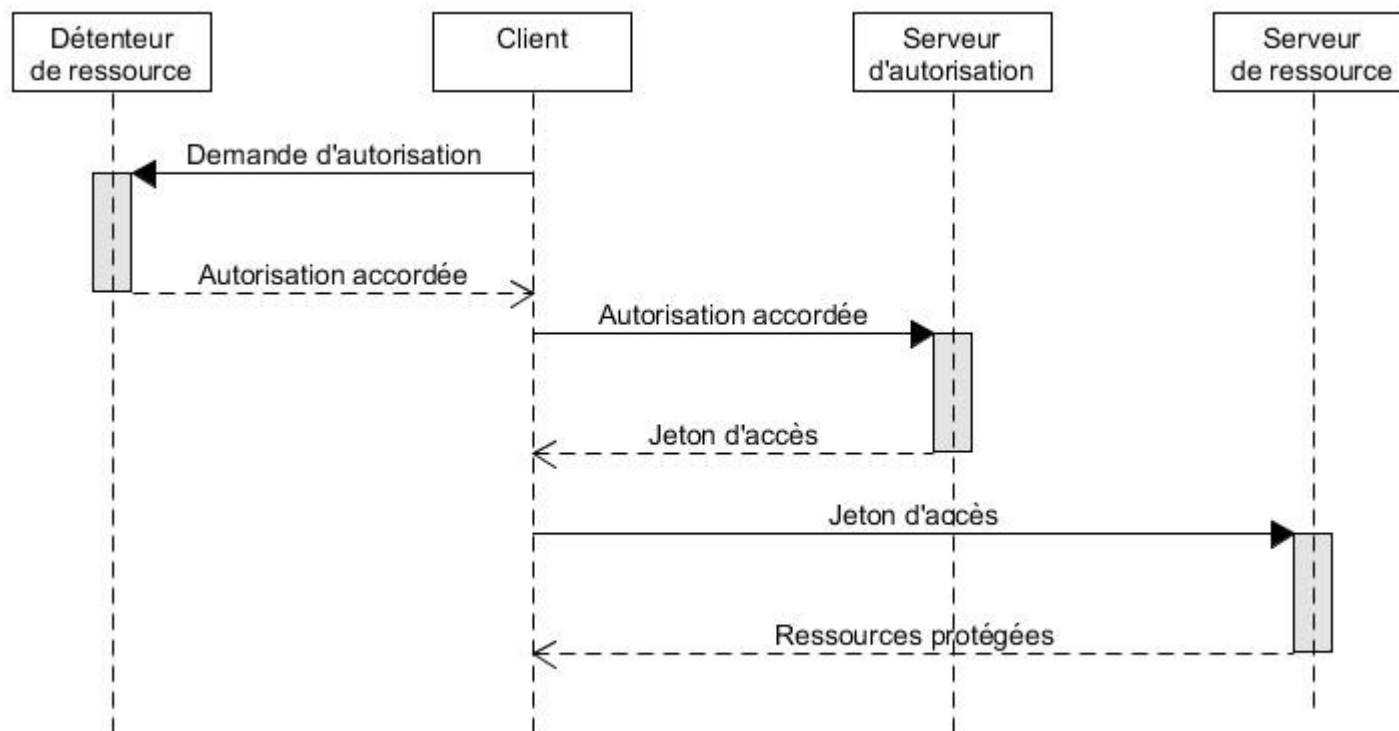
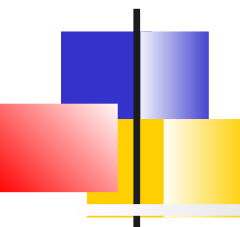
Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

Le **serveur de ressources** est l'API utilisée pour accéder aux ressources protégées

Le **serveur d'autorisation** est le serveur qui autorise un client à accéder aux ressources en lui fournissant un jeton. Il peut demander l'approbation de l'utilisateur

**L'utilisateur** est la personne qui donne accès à certaines parties de son compte

Rq: Un participant du protocole peut jouer plusieurs rôles





# Scénario

---

1. Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)
2. Obtenir l'autorisation de l'utilisateur.  
(4 types de grant)
3. Obtention du token (date d'expiration)
4. Appel de l'API pour obtenir les informations voulues en utilisant le token
5. Validation du token par le serveur de ressource





# Tokens

---

Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation

Les jetons sont ensuite présents dans les requêtes HTTP et contiennent des informations sensibles => HTTPS

Il y a 2 types de token

- Le **jeton d'accès**: Il a une durée de vie limité.
- Le **Refresh Token**: Délivré avec le jeton d'accès. Il est renvoyer au serveur d'autorisation pour renouveler le jeton d'accès lorsqu'il a expiré



# Périmètre d'accès

---

Le **scope** est un paramètre utilisé pour limiter les droits d'accès d'un client

Le serveur d'autorisation définit les *scopes* disponibles

Le client peut préciser le *scope* qu'il veut utiliser lors de l'accès au serveur d'autorisation



# Enregistrement du client

---

Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.

Le client doit fournir :

- **Application Name**: Le nom de l'application
- **Redirect URLs**: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès
- **Grant Types** : Les types d'autorisations utilisables par le client
- **Javascript Origin** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*

Le serveur répond avec :

- **Client Id**:
- **Client Secret**: Clé devant rester confidentielle



# *OAuth2 Grant Type*

---

Différents moyens afin que l'utilisateur donne son accord : les **grant types**

- ***authorization code*** :
  - L'utilisateur est dirigé vers le serveur d'autorisation
  - L'utilisateur consent sur le serveur d'autorisation
  - Il est redirigé vers le client avec un code d'autorisation
  - Le client utilise le code pour obtenir le jeton
- ***implicit*** : Jeton fourni directement. Certains serveurs interdisent de mode
- ***password*** : Le client fournit les crédeniels de l'utilisateur
- ***client credentials*** : Le client est l'utilisateur
- ***device code*** :



# Usage du jeton

---

Le jeton est passé à travers 2 moyens :

- Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)
- ***L'entête d'Authorization***

```
GET /profile HTTP/1.1
```

```
Host: api.example.com
```

```
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>



# Validation du jeton

---

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation d'un support persistant partagé (ex. JdbcStore)
- Utilisation de JWT et validation via clé privé ou clé publique



# JWT

**JSON Web Token (JWT)** est un standard ouvert défini dans la RFC 75191.

Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties.

La sécurité consiste en la vérification de l'intégrité des données à l'aide d'une signature numérique. (HMAC ou RSA).

Dans le cadre d'une application REST SpringBoot, le jeton contient les informations d'authentification d'un user :  
Subject + Rôles

Différentes implémentations existent en Java (*io.jsonwebtoken*, ...) ou le starter ***spring-security-oauth2-jose***



# Cloud Security

---

Introduction  
Rappels OAuth2  
**Spring Boot et OAuth2**  
Spring Cloud Security





# Apport de SpringBoot

---

Le support de OAuth via Spring a été revu :

- Le projet **spring-security-oauth2** a été déprécié et remplacé par SpringSecurity 5.

Voir :

<https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide>

- Il n'y a plus de support pour un serveur d'autorisation

3 starters sont désormais fourni :

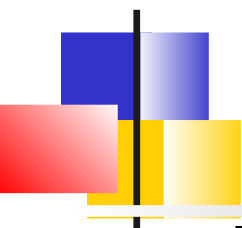
- **OAuth2 Client** : Intégration pour utiliser un login OAuth2 fournit par Google, Github, Facebook, ...
- **OAuth2 Resource server** : Application permettant de définir des ACLs par rapport aux scopes client et aux rôles contenu dans des jetons OAuth
- **Okta** : Pour travailler avec le fournisseur OAuth Okta



# Solutions pour un serveur d'autorisation

---

- Utiliser un produit autonome
- Un projet Spring pour un serveur d'autorisation est en cours :  
<https://github.com/spring-projects-experimental/spring-authorization-server>
- Une autre alternative est d'embarquer une solution *oAuth* comme *KeyCloak* dans un application SpringBoot  
Voir par exemple :  
<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app>



# Serveur de ressources

---

Dépendance :

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
</dependency>
```

Le serveur de ressources doit vérifier la signature du jeton pour s'assurer que les données n'ont pas été modifiées.

- ***jwk-set-uri*** contient la clé publique que le serveur peut utiliser pour la vérification
- ***issuer-uri*** pointe vers l'URI du serveur d'autorisation de base, qui peut également être utilisé pour localiser le endpoint fournissant la clé publique



# Exemple *application.yml*

---

```
server:
  port: 8081
  servlet:
    context-path: /resource-server

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://keycloak:8083/auth/realms/myRealm
          jwk-set-uri: http://keycloak:8083/auth/realms/myRealm/protocol/openid-connect/certs
```



# Configuration typique *SpringBoot*

---

## @Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/user/info", "/api/foos/**")
                    .hasAuthority("SCOPE_read")
                .antMatchers(HttpMethod.POST, "/api/foos")
                    .hasAuthority("SCOPE_write")
                .anyRequest()
                    .authenticated()
            .and()
                .oauth2ResourceServer()
                    .jwt();
    }
}
```



# Personnalisations

---

Différents aspects de la configuration par défaut peuvent être personnalisées :

- Revendications personnalisées dans le jeton
- Charger la clé à partir d'un KeyStore



# Cloud Security

---

Introduction  
Rappels OAuth2  
Spring Boot et OAuth2  
**Spring Cloud Security**



# Apports de Spring Cloud Security

---

Construit sur **Spring Boot** et **Spring Security OAuth2**

*Spring Cloud Security* facilite la mise en place de pattern comme :

- Le relais des jetons *oAuth* à partir d'un front-end jusqu'aux services back-end avec Zuul
- Le relai de jetons entre des ressources serveurs
- Un intercepteur afin qu'un client *Feign* se comporte comme *OAuth2RestTemplate* (fetching tokens etc.)
- Configure l'authentification dans un proxy Zuul





# Relais entre serveur de ressources (1)

---

```
@Configuration
@EnableOAuth2Sso
public class SiteSecurityConfigurer
    extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // ...
    }
}
```

Configuration serveur de ressource :

```
security:
  oauth2:
    client:
      accessTokenUri: http://localhost:7070/authserver/oauth/token
      userAuthorizationUri: http://localhost:7070/authserver/oauth/authorize
      clientId: authserver
      clientSecret: passwordforauthserver
  resource:
    userInfoUri: http://localhost:9000/user
```



# Relais entre serveur de ressources (2)

---

@Bean

```
public OAuth2RestOperations restOperations(  
    OAuth2ProtectedResourceDetails resource, OAuth2ClientContext context) {  
    return new OAuth2RestTemplate(resource, context);  
}
```

-----

@Autowired

```
private RestOperations restOperations;
```

```
public String callRemote() {  
    return restOperations.getForObject("http://remote-service/api",  
        String.class);  
}
```



# *Zuul Proxy*

---

```
@Controller
```

```
@EnableOAuth2Sso
```

```
@EnableZuulProxy
```

```
class Application {
```

```
}
```

L'annotation **@EnableOAuth2Sso** a pour effet que *Zuul* obtienne et valide des jetons auprès d'un serveur d'autorisation

De plus ces jetons sont transférés vers les services backend gérés par Zuul



# Exemple de configuration Zuul

---

```
# customers obtient le jeton d'accès  
# stores n'a que l'entête authorization  
# recommendation n'a rien
```

```
proxy:
```

```
  auth:
```

```
    routes:
```

```
      customers: oauth2
```

```
      stores: passthru
```

```
      recommendations: none
```



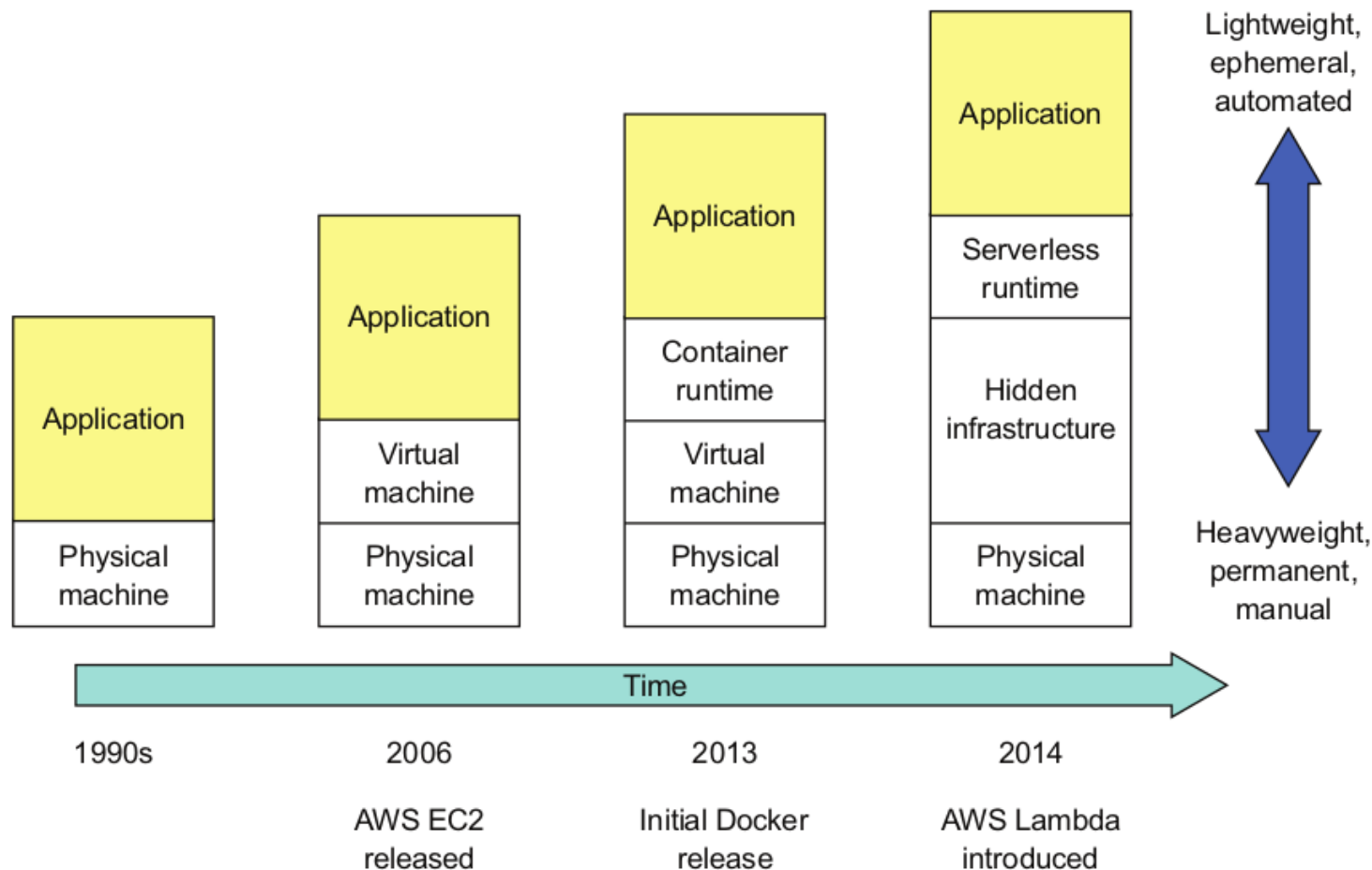
# Déploiement

---

## **Alternatives**

Support pour docker  
Spring Cloud Kubernetes

# Evolution des infrastructures





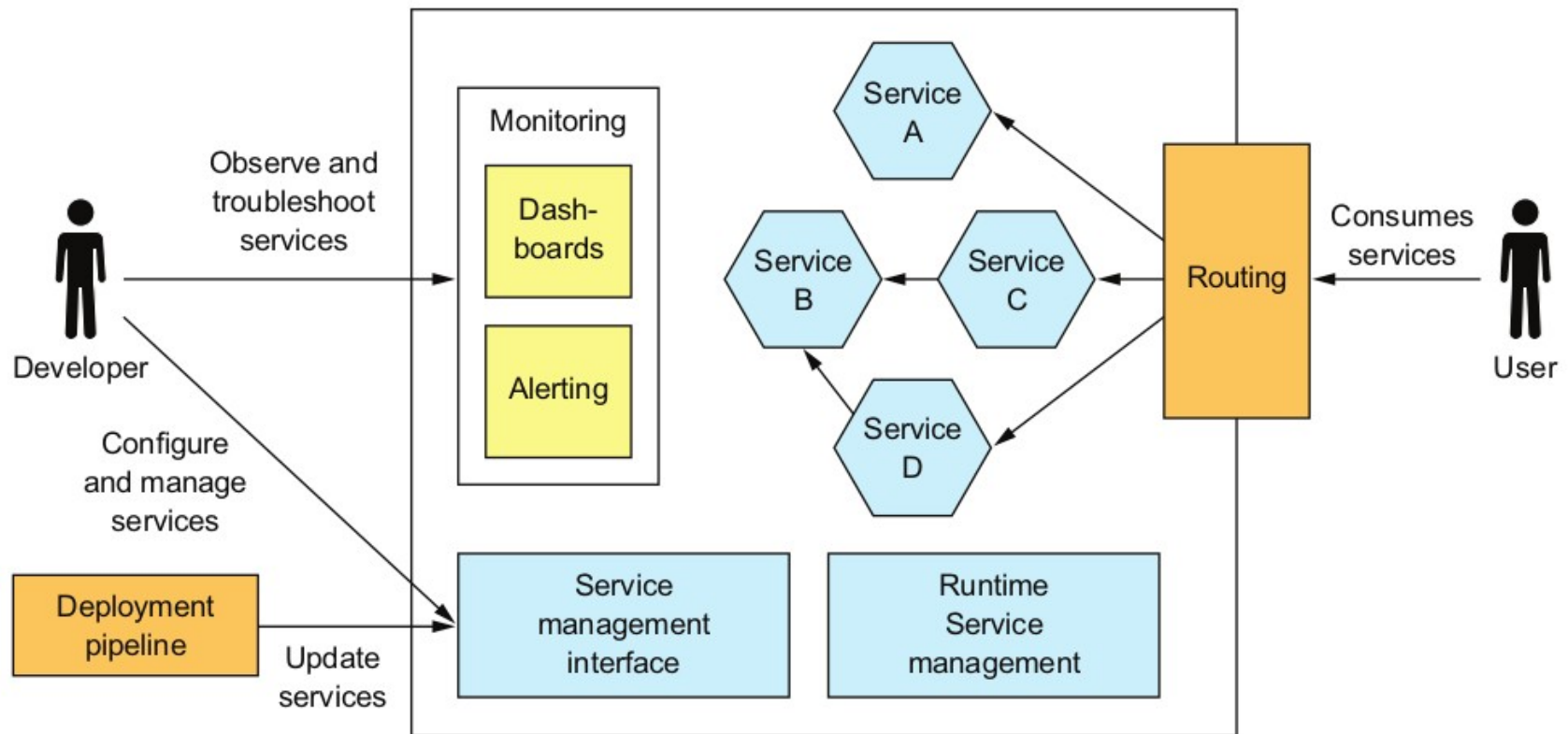
# Contraintes

---

L'environnement de production doit implémenter 4 capacités clés:

- Interface de gestion des services: Créer, mettre à jour et configurer des services. Idéalement une API REST invoquée par des commandes en ligne commande et GUI.
- Gestion des services à l'exécution: tente de garantir que le nombre souhaité d'instances de service s'exécute à tout moment.
- Surveillance/Observabilité : Fournit aux développeurs un aperçu et une vision détaillée aperçu de ce que font leurs services en production
- Routage des requêtes : Les requêtes des utilisateurs sont acheminées vers les services.

# Environnement de production





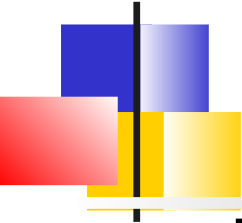


# Alternatives de déploiement

---

Différentes options de packaging pour le déploiement existent :

- Packages spécifiques à un langage : Un jar, war, zip Javascript, ...
- Machines virtuelles qui incluent la pile technologique (OS, JVM, ...)
- Containers
- Serverless Deployment



# Inconvénients des packages spécifique à un langage

---

Manque d'encapsulation de la pile technologique.  
=> l'équipe Ops doit connaître précisément les besoins de provisionnement des machines

Aucune possibilité de limiter les ressources consommées par une instance de service.  
=> Une instance de service peut consommer toutes les ressources CPU, Mémoire de la machine

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

Pas de support pour le placement des instances sur les ressources disponibles



# Inconvénients des VMs

---

Utilisation peu efficace des ressources

=> Chaque service a la surcharge d'une machine entière, souvent surdimensionnée par rapport au service lui-même

Déploiements relativement lents

=> Construction longue car taille d'image très volumineuse, démarrage de service long

Surcharge d'administration système

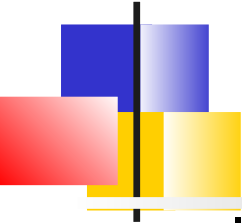
=> Patch systèmes à appliquer



# Déploiement

---

Alternatives  
**Support pour docker**  
Spring Cloud Kubernetes



# Dockerfile basique

---

Il est facile de construire un jar exécutable d'une application Spring Boot via les plugins Maven/Gradle

Un Dockerfile basique est alors :

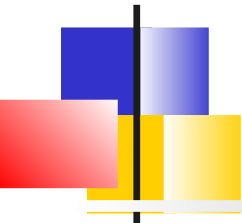
```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Pour le construire :

```
docker build --build-arg JAR_FILE=target/*.jar -t myorg/myapp .
```

Pour l'exécuter :

```
docker run -p 8080:8080 myorg/myapp
```



# Dockerfile + sophistiqué

---

Pour pouvoir passer des variables  
d'environnement et des propriétés  
SpringBoot au démarrage

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -jar /app.jar ${0} $
{@}"]
```

On peut alors démarrer l'image via :

```
docker run -p 9000:9000 -e "JAVA_OPTS=-Ddebug -Xmx128m"
myorg/myapp --server.port=9000
```



# Taille des images

---

Les images alpine sont plus petites que les images standard openjdk de Dockerhub.

20 mB sont économisés en utilisant une jre plutôt qu'une jdk

On peut également essayer d'utiliser jlink (à partir de Java11) pour se créer une image sur mesure en ne sélectionnant que les modules Java utilisés

- Attention pas de cache, si tous les services Java utilisent des JRE personnalisés



# Couches Docker

---

Un jar Spring Boot a naturellement des "couches" Docker en raison de son packaging.

On peut isoler dans une couche Docker les dépendances externes ainsi tous les services utilisant les mêmes dépendances pourront se construire et se lancer plus vite.  
(Cache des container runtime)





# Dockerfile à couche

---

```
$ mkdir target/dependency
$ (cd target/dependency; jar -xf ../*.jar)
$ docker build -t myorg/myapp .
```

—

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "hello.Application"]
```



# Accélérateur de démarrage

---

Quelques astuces pour accélérer le démarrage du service :

- Améliorer le scan du classpath avec *spring-context-indexer*<sup>1</sup>
- Limiter *actuator* à ce qui est vraiment nécessaire
- SB 2.1+ et S 5.1+
- Spécifier le chemin vers le fichier de config :  
*spring.config.location*
- Désactiver JMX `spring.jmx.enabled=false`
- Exécuter la JVM avec `-noverify`
- Pour Java8, également `-XX:+UnlockExperimentalVMOptions`  
`-XX:+UseCGroupMemoryLimitForHeap`

1. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-scanning-index>



# Utilisateur dédié

---

Les services doivent s'exécuter avec un utilisateur non root

```
FROM openjdk:8-jdk-alpine
```

```
RUN addgroup -S demo && adduser -S demo -G demo  
USER demo
```



# Spring-Boot Plugin

---

Depuis la 2.3, Spring boot fournit des plugins Maven/Gradle capables de construire les images

```
$ ./mvnw spring-boot:build-image
```

```
$ ./gradlew bootBuildImage
```

- Plus besoin de Dockerfile
- Nécessite que le démon Docker s'exécute
- Par défaut le nom de l'image est :  
    `docker.io/<group>/<artifact>:latest`
- Utilise les *Cloud Native Buildpacks*



# Autres plugins

---

***Spotify Maven Plugin*** nécessite un Dockerfile et ajoute des objectifs permettant d'automatiser la construction de l'image dans le build Maven

***Palantir Gradle*** est capable de générer un Dockerfile ou d'utiliser celui que l'on fournit

***Jib*** est également un projet Google qui permet de construire des images optimisées (Docker ou OCI) sans Docker installé



# BuildPacks

---

Cloud Foundry utilise les ***buildbacks*** qui transforment automatiquement le code source en conteneur<sup>1</sup>

- Les développeurs n'ont pas besoin de se soucier des détails sur la construction du conteneur.
- Les buildpacks ont de nombreuses fonctionnalités pour la mise en cache des résultats de construction et des dépendances  
=> Souvent un buildpack s'exécute beaucoup plus rapidement qu'un docker natif



# Sortie standard d'un buildpack

---

```
$ pack build myorg/myapp --builder=cloudfoundry/cnb:bionic --path=.
2018/11/07 09:54:48 Pulling builder image 'cloudfoundry/cnb:bionic' (use --no-pull flag to skip this step)
2018/11/07 09:54:49 Selected run image 'packs/run' from stack 'io.buildpacks.stacks.bionic'
2018/11/07 09:54:49 Pulling run image 'packs/run' (use --no-pull flag to skip this step)
*** DETECTING:
2018/11/07 09:54:52 Group: Cloud Foundry OpenJDK Buildpack: pass | Cloud Foundry Build System Buildpack: pass | Cloud
Foundry JVM Application Buildpack: pass
*** ANALYZING: Reading information from previous image for possible re-use
*** BUILDING:
-----> Cloud Foundry OpenJDK Buildpack 1.0.0-BUILD-SNAPSHOT
-----> OpenJDK JDK 1.8.192: Reusing cached dependency
-----> OpenJDK JRE 1.8.192: Reusing cached launch layer

-----> Cloud Foundry Build System Buildpack 1.0.0-BUILD-SNAPSHOT
-----> Using Maven wrapper
      Linking Maven Cache to /home/pack/.m2
-----> Building application
      Running /workspace/app/mvnw -Dmaven.test.skip=true package
...
---> Running in e6c4a94240c2
---> 4f3a96a4f38c
---> 4f3a96a4f38c
Successfully built 4f3a96a4f38c
Successfully tagged myorg/myapp:latest
```



# *KNative*

---

***KNative*** est un projet initié par Google qui a pour but de fournir une infrastructure *ServerLess* au dessus d'un cluster Kubernetes

Comme les *buildpacks*, il est capable de construire les conteneurs à partir du code source.





# Déploiement

---

Alternatives  
Support pour docker  
**Spring Cloud Kubernetes**



# Alternatives

---

Différentes alternatives pour déployer sur un cluster Kubernetes :

- Utiliser les descripteurs natifs et le client ***kubectl***
- Utiliser les packages ***Helm***  
<https://helm.sh/>
- S'équiper d'un outil de IAAS tel que ***Terraform***  
<https://www.terraform.io/>

Toutes ces alternatives nécessitent d'avoir publier les images des conteneurs dans un dépôt.

Ex : *DockerHub*

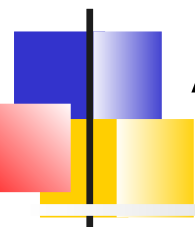


# Exemple descripteur natif

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: delivery-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      name: delivery-service
  template:
    metadata:
      labels:
        name: delivery-service
    spec:
      containers:
        - name: delivery-service
          image: dthibau/delivery-service:1.0.9
          imagePullPolicy: Always
```

-----  
**kubectl apply -f delivery-service.yml**



# Adaptation à l'environnement

---

Pour adapter la configuration à l'environnement de production, on peut s'appuyer :

- Sur des variables d'environnement
- Sur des arguments de démarrage du conteneur
- Sur le service de *ConfigMap* offerte par Kubernetes



# Example

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: delivery-service
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      name: delivery-service
  template:
    metadata:
      labels:
        name: delivery-service
    spec:
      containers:
        - name: delivery-service
          env:
            - name: SPRING_PROFILES_ACTIVE
              value: prod,swagger
            - name: SPRING_DATASOURCE_URL
              valueFrom:
                configMapKeyRef:
                  name: postgres-config
                  key: POSTGRES_URL
          image: dthibau/delivery-service:1.0.9
          imagePullPolicy: Always
```



# Spring Cloud Kubernetes

---

***Spring Cloud Kubernetes*** est un projet dont l'objectif est de faciliter le déploiement de micro-services SpringBoot vers Kubernetes

Il fournit des implémentations des interfaces de *Spring Cloud Commons* qui utilisent les services natifs de Kubernetes :

- *@EnableDiscoveryClient*
- Les objets *PropertySource* configurés via ***ConfigMaps***
- Équilibrage de charge côté client via Netflix Ribbon  
(En cours de dépréciation)



# Annexes



## *Alternative Consul*





# Alternative *Consul*

---

Un autre alternative aux outils Netflix est possible : ***Consul***

Le produit *Consul* permet les 2 services de discovery et de configuration

Il est facilement intégrable dans SpringCloud



# Consul vs Eureka (1)

---

Eureka est uniquement un service de discovery. L'architecture client/server, est en général constituée :

- d'un ensemble de serveurs Eureka (1 par datacenter).
- De clients avec un SDK embarqué pour s'enregistrer et découvrir les services



# Consul vs Eureka (2)

---

Consul propose plus de fonctionnalités :

- Vérification de la santé plus riche (TCP, HTTP, Nagios/Sensu ou TTL comme Eureka)
- Stockage de paires clé/valeurs (config) et réplication d'état via le protocole Raft
- Prise en compte de plusieurs datacenter

Consul nécessite un serveur dans chaque datacenter et un agent sur chaque client.

L'agent permet aux applications d'être indépendantes de Consul. L'enregistrement s'effectue par des fichiers de configuration et la découverte via DNS ou par des load balancer.



# Mise en place - discovery

---

Installation de Consul : Serveurs + agent pour chaque micro-services

Ajout de starter :

*spring-cloud-starter-consul-discovery*

Configuration :

```
spring:
  cloud:
    consul:
      host: localhost
      port: 8500
    discovery:
      healthCheckPath: ${management.server.servlet.context-path}/health
      healthCheckInterval: 15s
```



# Mise en place - configuration

---

Installation de Consul : Serveurs + agent pour chaque micro-services

Ajout de starter :

***spring-cloud-starter-consul-config***

Configuration :

```
spring:
  cloud:
    consul:
      config:
        enabled: true
        prefix: configuration # Surveillance et rafraîchissement automatique
        defaultContext: apps
        profileSeparator: '::'
```



## *Répartition de charge avec Ribbon*



# Introduction Ribbon

---

Ribbon (Netflix) est un répartiteur de charge côté client décentralisé intégré dans SpringCloud LoadBalancer.

Il permet de contrôler facilement les clients HTTP ou TCP.



# Configuration Client Ribbon

---

Chaque répartiteur de charge est un **ensemble de beans** Spring collaborant pour contacter un serveur distant

L'ensemble a un **nom** fourni par le développeur (par exemple en utilisant l'annotation *@FeignClient*).

Spring Cloud crée les beans en utilisant une configuration (*RibbonClientConfiguration*).





# Usage de Ribbon

---

spring-cloud-starter-ribbon

Ensuite, la configuration d'un client peut s'effectuer en utilisant des propriétés externes

`<client>.ribbon.*`

Ou programmatiquement

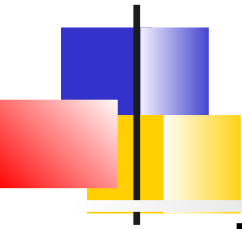
```
@Configuration
@RibbonClient(name = "foo", configuration =
    FooConfiguration.class)
public class TestConfiguration {
}
```



# Exemple configuration

---

```
say-hello:  
  ribbon:  
    eureka:  
      enabled: false  
      listOfServers:  
localhost:8090,localhost:9092,localhost:9999  
      ServerListRefreshInterval: 15000
```



# Beans

---

La configuration crée un *ApplicationContext* pour chaque client nommé comprenant les beans suivants :

- ***IClientConfig***, stocke la configuration
- ***ILoadBalancer*** : Le répartiteur
- ***ServerList*** : définit comment obtenir la liste des serveurs
- ***IRule*** décrivant la stratégie de répartition
- ***IPing*** : Donne la période des ping vers un serveur.



# Répartition de charge et *RestTemplate*

---

```
@RibbonClient(name = "ribbonApp")
```

```
...
```

```
public class RibbonApp {
```

```
    @Bean
```

```
    @LoadBalanced
```

```
    RestTemplate restTemplate () {
```

```
        new RestTemplate()
```

```
    }
```

```
    public void callMicroService() {
```

```
        // ribbonApp est le nom du client Ribbon
```

```
        Store store =
```

```
        restTemplate.getForObject("http://ribbonApp/store", Store.class) ;
```

```
    }
```

```
}
```



# Utilisation avec Eureka

---

Quand *Eureka* est utilisé en conjonction avec *Ribbon*

- *ribbonServerList* est surchargé avec une extension de ***DiscoveryEnabledNIWSServerList*** qui renseigne la liste de serveurs à partir d'Eureka
- Il remplace également l'interface *IPing* avec ***NIWSDiscoveryPing*** qui délègue à Eureka de déterminer si un serveur est UP



## *Pattern disjoncteur avec Hystrix*



# Introduction

---

Netflix a créé la librairie Hystrix qui implémente le pattern « ***circuit breaker*** »

- Une erreur dans un service bas niveau peut provoquer en cascade des erreurs jusqu'à l'utilisateur final.
- Lorsque des appels vers un service particulier atteint un seuil d'erreur (Par défaut, 20 erreurs en 5 secondes), le circuit s'ouvre et les appels ne sont plus effectués
- Un ***fallback*** peut être fourni par le développeur



# Exemple de *Hystrix*

---

Starter : **hystrix**

```
--

@SpringBootApplication
@EnableCircuitBreaker
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}

@Component
public class StoreIntegration {

    @HystrixCommand(fallbackMethod = "defaultStores")
    public Object getStores(Map<String, Object> parameters) {
        //do stuff that might fail
    }

    public Object defaultStores(Map<String, Object> parameters) {
        return /* something useful */;
    }

}
```





# Résilience avec *Hystrix*

---

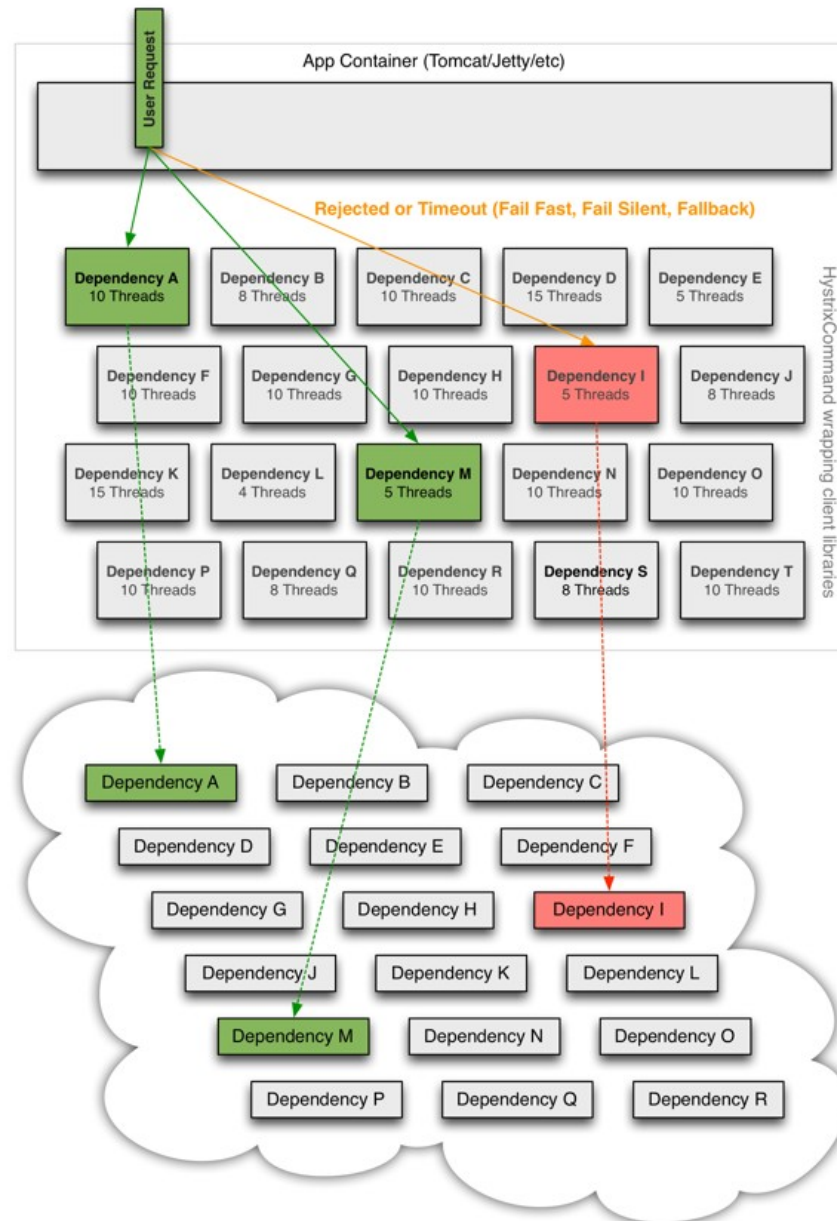
*Hystrix* permet la résilience

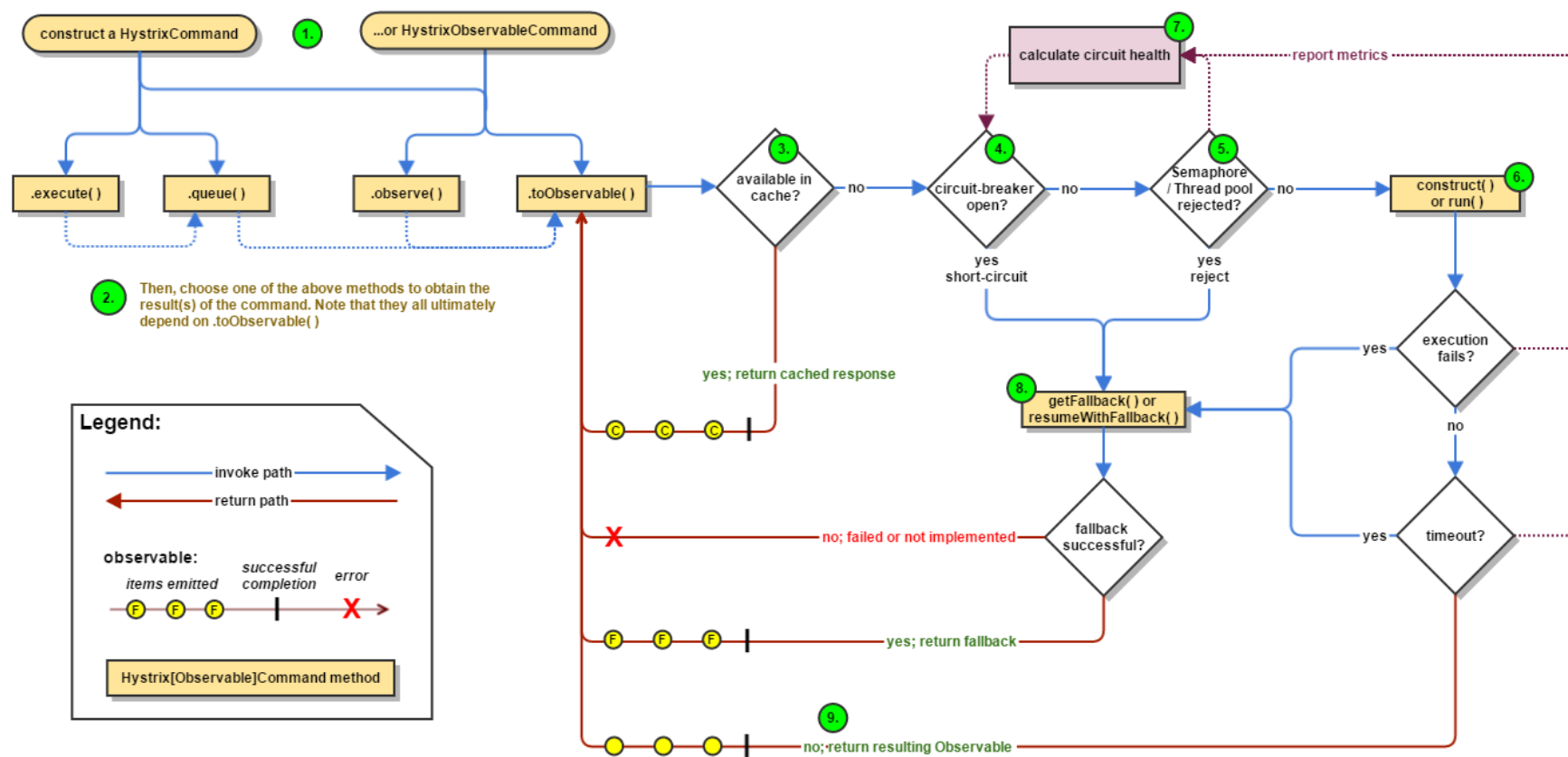
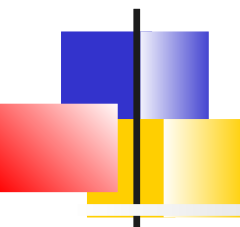
- Les appels sont effectués via un autre pool de threads
- Cela ne bloque pas la thread de service
- Peut implémenter des timeouts

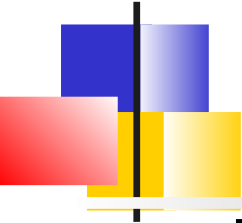
=> Mécanisme de disjonction si un service ne répond pas ou répond mal

Spring Boot permet une approche par annotation

# Thread pools Hystrix







# Ouverture/fermeture de circuit

---

La décision d'ouvrir le circuit (condition anormale) s'appuie

- Sur un nombre d'erreurs dans une fenêtre de temps

La décision de fermer le circuit (retour aux conditions normales de fonctionnement) s'effectue ainsi :

- Après un certain temps où le circuit est resté ouvert (*circuitBreakerSleepWindowInMilliseconds*)
- Une (seule) requête est tentée :
  - Si elle échoue le circuit reste ouvert
  - Si elle réussit le circuit redevient fermé.



# Intégration SpringBoot

---

L'annotation ***@HystrixCommand*** est fourni par la librairie "*javanica*" de Netflix.

Spring Cloud encapsule automatiquement les beans Spring annotés avec *@HystrixCommand* dans un proxy connecté au disjoncteur Hystrix.

Le disjoncteur calcule quand ouvrir ou fermer le circuit et quoi faire en cas d'erreur.



# Indicateurs de santé

---

D'autre part, le statut des disjoncteurs sont exposés via l'endpoint ***/health***

```
{
  "hystrix": {
    "openCircuitBreakers": [
      "StoreIntegration::getStoresByLocationLink"
    ],
    "status": "CIRCUIT_OPEN"
  },
  "status": "UP"
}
```

Il faut avoir activé *spring-actuator* dans les dépendances



*Zuul*



# Zuul routing

---

***Zuul Routing*** (également Netflix) permet d'automatiquement mapper des routes (URLs) vers des services enregistrés

=> Pas de configuration

=> Peut ajouter des filtres, etc.

Il inclut également Ribbon et son répartiteur de charge

Les règles de routage sont exprimées dans des langages JVM (essentiellement Java et Groovy)





# Cas d'usage

---

*Netflix* utilisait *Zuul* pour :

- L'authentification et la sécurité
- Des tests de stress
- Canary Testing : Déploiement de nouvelles fonctionnalités sur un ensemble restreint d'utilisateurs
- Du délestage de charge
- Du routage dynamique
- De la migration de service
- Le traitement des réponses statiques
- Gestion active du trafic



# Mise en place *SpringBoot*

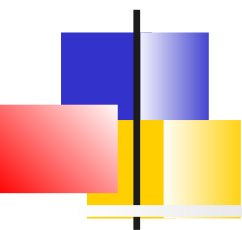
---

**Starter : zuul**

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableZuulProxy
public class ProxyApplication {

    public static void main(String[] args) {
        SpringApplication.run(ProxyApplication.class, args);
    }

}
```



# Fonctionnement par défaut

---

Le proxy utilise *Ribbon* pour localiser une instance

Toutes les requêtes sont exécutées dans une commande *Hystrix*

- => Ainsi, les erreurs sont remontées dans les métriques Hystrix
- => Une fois que le circuit est ouvert, le proxy n'essaie plus de contacter le service.

Par défaut, un service nommé ***users*** reçoit les requêtes de ***/users***



# Configuration

---

**# Tous les services sont ignorés à l'exception de users,  
# on utilise par défaut Ribbon et Hystrix**

```
zuul:
  ignoredServices: '*'
  routes:
    users: /myusers/**
```

**# Alternative avec serviceId, Ribbon et Hystrix doivent être configurés manuellement**

```
zuul:
  routes:
    users:
      path: /myusers/**
      serviceId: users_service
```

**# Ou, sans service de Discovery, Ribbon et Hystrix doivent être configurés manuellement**

```
zuul:
  routes:
    users:
      path: /myusers/**
      url: http://example.com/users_service
```



# Cookies et entête HTTP

---

Il est possible de spécifier une liste des entêtes à ignorer.

```
zuul.ignoredHeaders
```

```
zuul.ignoreSecurityHeaders # entêtes  
SpringSecurity
```

Les cookies pour le navigateur semblent venir tous du proxy.

=> Cela devient difficile de s'appuyer sur ces techniques à moins de partager les cookies dans tous les services de backend



# Endpoint */routes*

---

Si Spring Actuator est activé, un nouvelle endpoint est disponible : ***/routes***

- Un *GET* affiche les routes configurées
- Un *POST* force un rafraîchissement des routes



# Monitoring

---

Hystrix dashboard et Turbine



# Flux Hystrix

---

Une application utilisant *Hystrix* peut générer un flux *json* permettant de surveiller les circuits

Le flux est généré si l'application est annotée avec **@EnableCircuitBreaker**

Avec la version 2.x il faut également la propriété :

*management.endpoints.web.exposure.include: hystrix.stream*

Le flux est alors accessible sur le endpoint ***/hystrix.stream***





# Tableau de bord Hystrix

---

Un flux *Hystrix* fourni par un micro-service peut être visualisé graphiquement par le tableau de bord Hystrix

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>

    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifa
ctId>
  </dependency>
</dependencies>
```

+

**@EnableHystrixDashboard**

=> Endpoint **/hystrix**

# /hystrix



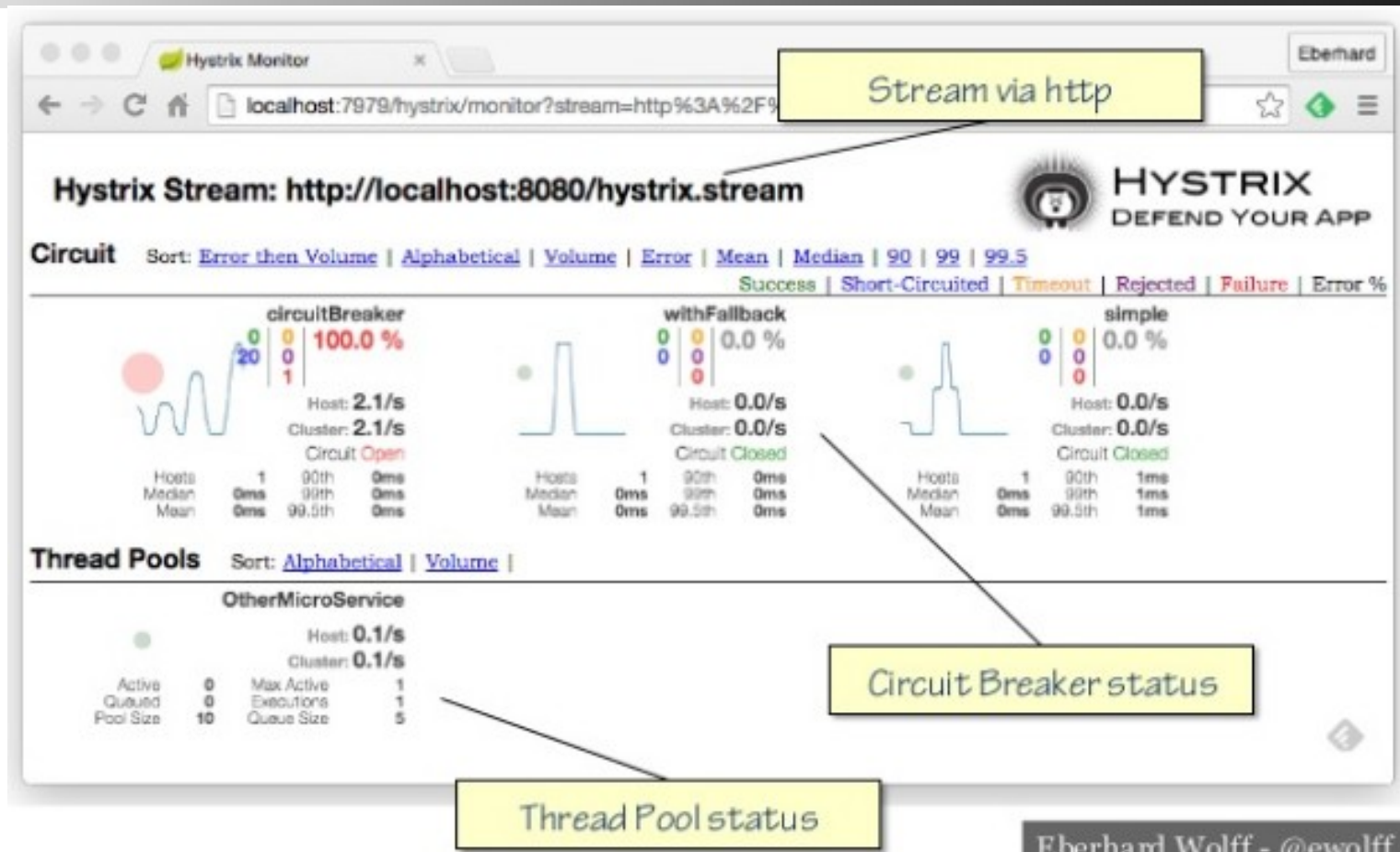
## Hystrix Dashboard

*Cluster via Turbine (default cluster):* <http://turbine-hostname:port/turbine.stream>  
*Cluster via Turbine (custom cluster):* [http://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](http://turbine-hostname:port/turbine.stream?cluster=[clusterName])  
*Single Hystrix App:* <http://hystrix-app:port/hystrix.stream>

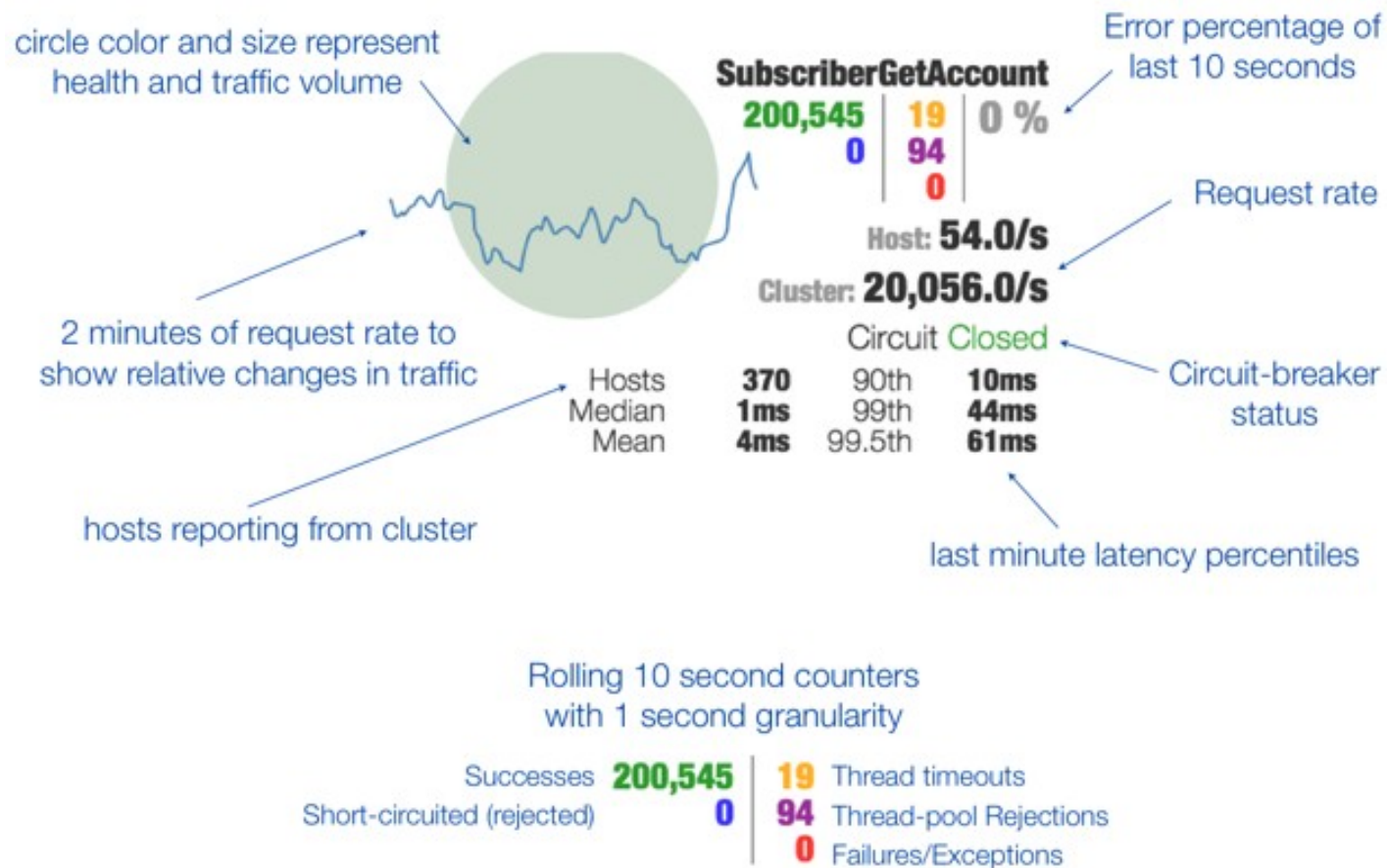
Delay:  ms

Title:

# Tableau de bord



# Détails dashboard





# Introduction Turbine

---

*Hystrix Dashboard* fournit des informations sur une seule application

**Turbine** permet d'agrégier les informations de toutes les applications d'un cluster



# Intégration de Turbine

---

Ajouter les dépendances :

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-turbine</artifactId>
</dependency>
```

Autoriser Turbine dans une application SpringBoot :

```
@SpringBootApplication
@EnableHystrixDashboard
@EnableTurbine
public class MonitorApplication {
    ...
}
```



# Configuration

---

turbine:

```
clusterNameExpression: new String("default")  
appConfig: MEMBERS-SERVICE,PROXY-SERVICE
```

Un flux turbine pour le cluster SAMPLE-HYSTRIX-AGGREGATE est disponible à

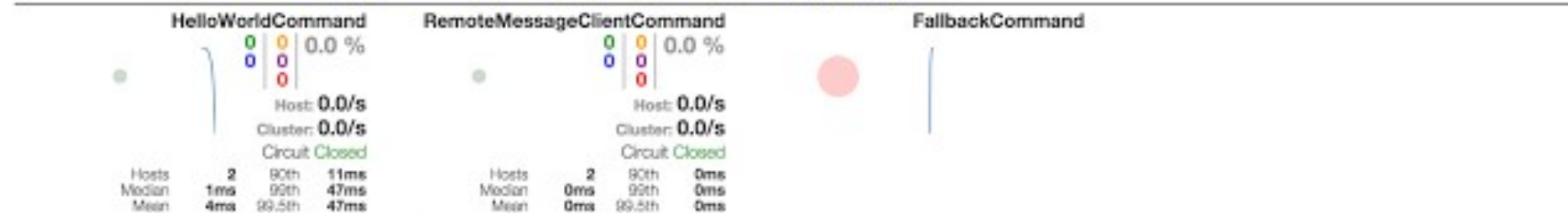
`/turbine.stream?cluster=SAMPLE-HYSTRIX-AGGREGATE`

La constitution du cluster est déduite d'Eureka, turbine récupère les flux Hystrix de toutes les instances et les agrège

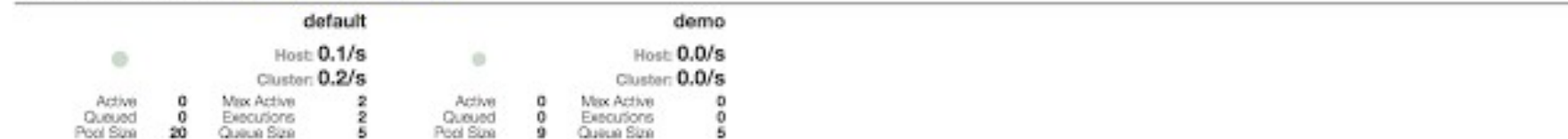
# Tableau de bord Turbine

Hystrix Stream: <http://samplemonitor:8080/turbine.stream?cluster=SAMPLE-HYSTRIX-AGGREGATE>

Circuit Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)



Thread Pools Sort: [Alphabetical](#) | [Volume](#)







# *Spring Cloud Bus*



# Introduction

---

***Spring Cloud Bus*** utilise un message broker pour faire communiquer les micro-services.

L'usage principal est de diffuser les changements de configuration.  
Il joue alors le rôle d'un Actuator distribué.

*Spring Cloud Bus* supporte *Apache Kafka*,  
*Redis*, *RabbitMQ*



# Objectifs

---

Sans Cloud Bus, la mise à jour de configuration d'un micro-service nécessite :

- Un redémarrage
- Ou si l'application contient Actuator, un appel individuel à l'URL */refresh*

*Spring Cloud Bus* permet de propager les changements de configuration à tous les microservices via une seule opération



# Config Server et WebHook

---

Au niveau du serveur de configuration, il est possible de rajouter une dépendance sur *spring-cloud-config-monitor* ainsi un endpoint */monitor* est disponible

Ce endpoint est capable de traiter les webhooks fourni par Github, Gitlab, ...

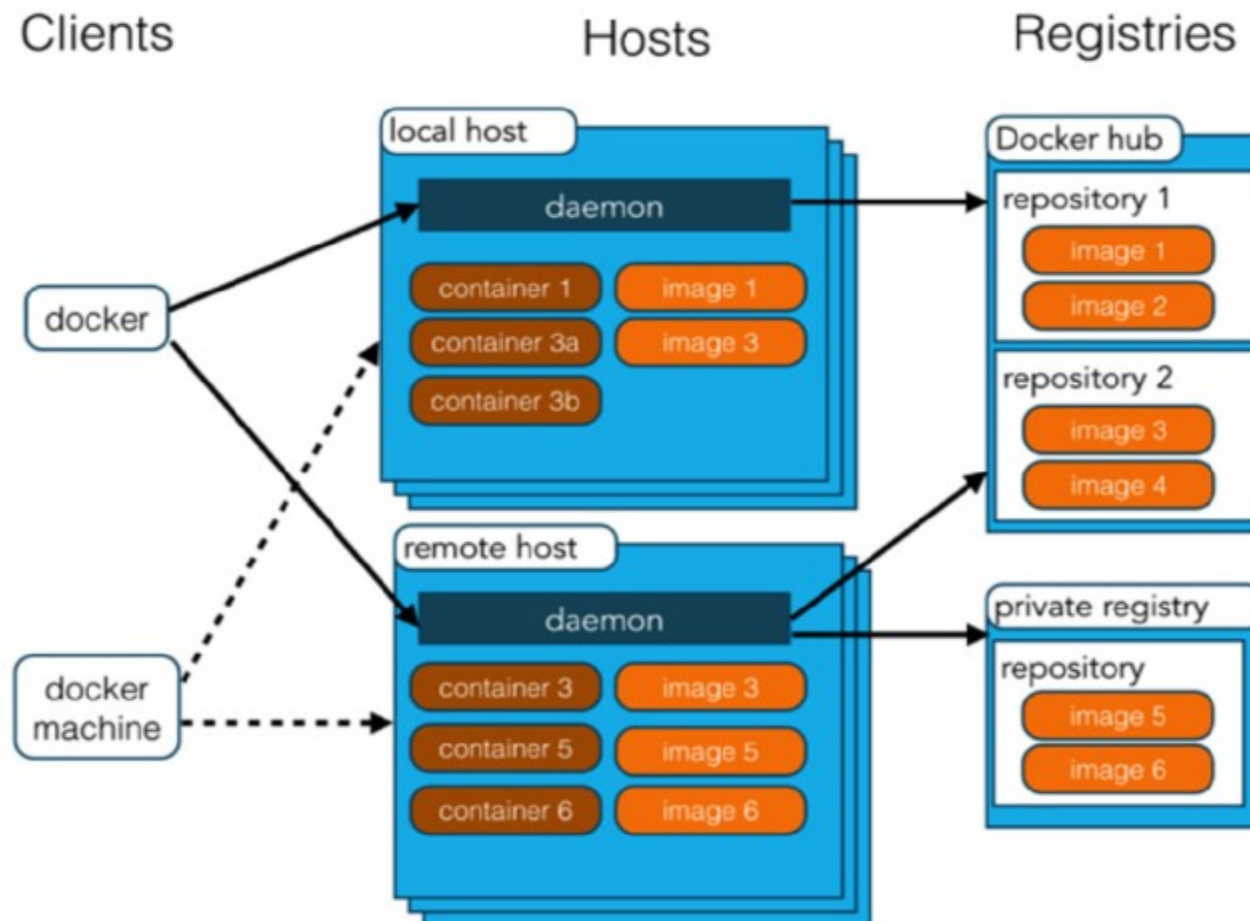
Lors de la réception d'un webhook, le serveur de config envoie un évènement sur le bus aux micro-services qui ont changé.

L'évènement provoque le rafraîchissement de la configuration



## *Rappels Docker*

# Rappels docker architecture





# Commandes Docker

---

#Récupération d'une image

```
docker pull ubuntu
```

#Récupération et instantiation

```
docker run hello-world
```

#Mode interactif

```
docker run -i -t
```

#Visualiser les sortie standard d'un conteneur

```
docker logs <container_id>
```

#Conteneurs en cours

```
docker ps
```

#Toutes les exécutions de conteneurs (même arrêt)

```
docker ps -a
```

#Lister les images

```
docker images
```

#Identifier les différences entre l'image et le conteneur

```
docker diff <container_id>
```

#Committer les différences

```
docker commit <container_id> <image_name>
```

#Tagger une image d'un repository

```
docker tag <image_name>[:tag] <name>[:tag]
```

#Pousser vers un dépôt distant

```
docker push <image_name>[:tag]
```



# Exemple DockerFile

---

**FROM** ubuntu

**MAINTAINER** Kimbro Staken

**RUN** apt-get install -y software-properties-common python

**RUN** add-apt-repository ppa:chris-lea/node.js

**RUN** echo "deb http://us.archive.ubuntu.com/ubuntu/ precise universe" >>  
/etc/apt/sources.list

**RUN** apt-get update

**RUN** apt-get install -y nodejs

**RUN** mkdir /var/www

**ADD** app.js /var/www/app.js

**EXPOSE** 8080

**CMD** ["/usr/bin/node", "/var/www/app.js"]

—

**docker build -t dthibau/myappli:latest**





# Communication entre machines

---

Chaque conteneur s'exécute à sa propre interface réseau (gérée par Docker)

Par défaut, Il est isolé

- De la machine hôtes
- Des autres containers



# Communication avec la machine hôte

---

Au démarrage d'un conteneur on peut :

- Associer un port exposé par le conteneur à un port local  
Option **-p**
- Monté un répertoire du conteneur sur le système de fichier local.  
Option **-v**

```
docker run -p 80:8080  
            -v /home/jenkins:/var/lib/jenkins  
            myImage
```



# *docker-compose*

***docker-compose*** est un outil pour définir et exécuter des applications Docker utilisant plusieurs conteneurs

- Avec un simple fichier, on spécifie les différents conteneurs, les ports exposés, les liens entre conteneurs.
- Ensuite avec une commande unique, on peut démarrer, arrêter, redémarrer l'ensemble des services.

Docker s'installe séparément sur Linux et est inclus dans Docker pour les distributions Mac ou Windows

Orienté au départ pour faciliter le développement et l'intégration ; il intègre de plus en plus des fonctionnalités pour la production



# Exemple configuration

---

**# Le fichier de configuration définit des services, des networks et des volumes.**

version: '2'

services:

annuaire:

build: ./annuaire/ **# context de build, présence d'un Dockerfile**

networks:

- back
- front

ports:

- "1111:1111" **# Exposition de port**

documentservice:

build: ./documentService/

networks:

- back

proxy:

build: ./proxy/

networks:

- front

ports:

- 8080:8080

**# Analogue à 'docker network create'**

networks:

back:

front:



# Commandes

---

**build** : Construire ou reconstruire les images

**config** : Valide le fichier de configuration

**down** : Stoppe et supprime les conteneurs

**exec** : Exécute une commande dans un container up

**logs** : Visualise la sortie standard

**port** : Affiche le port public d'une association de port

**pull** / **push** : Pull/push les images des services

**restart** : Redémarrage des services

**scale** : Fixe le nombre de container pour un service

**start** / **stop** : Démarrage/arrêt des services

**up** : Création et démarrage de conteneurs



---

*@EnableAuthorizationServer*



# Introduction

---

Avec ***spring-security-oauth2*** dans le classpath, l'auto-configuration permet de facilement mettre en place le serveur d'autorisation et/ou ressources

L'architecture micro-services étend le modèle de *oAuth2* appliqué aux services comme Google/Facebook/Github

Attention : Spring Security 5 reprend certains des fonctionnalités de *spring-security-oAuth2*



# Implémentation OAuth2

---

*Avec Spring Security OAuth2 :*

- Les demandes pour les jetons sont traitées par des contrôleurs de SpringMVC
- L'accès aux ressources protégées est vérifié par des filtres *http*





# Mécanismes

---

Du côté du serveur d'autorisation, 2 endpoints sont fournies :

- ***AuthorizationEndpoint*** (par défaut */oauth/authorize*) traite les requêtes d'autorisation (Autoriser le client à demander des jetons)
- ***TokenEndpoint*** (*/oauth/token*) pour demander des jetons

Pour le serveur de ressource, le filtre ***OAuth2AuthenticationProcessingFilter*** permet d'extraire le jeton d'une requête



# Configuration de l'autorisation

---

L'annotation **@EnableAuthorizationServer** fournit une configuration par défaut d'un serveur d'autorisation (**AuthorizationServerConfigurer**) .

Généralement, 3 aspects reste à être configurés :

- **ClientDetailsServiceConfigurer** : Comment récupérer les informations sur le client
- **AuthorizationServerSecurityConfigurer** : Les contraintes de sécurité sur les endpoints « jeton » (/oauth/\*)
- **AuthorizationServerEndpointsConfigurer** : Le token store, le gestionnaire d'authentification permettant de déterminer les rôles de l'utilisateur final et de vérifier son mot de passe



# Attributs d'un client

---

Un client a différents attributs :

- **clientId**: id.
- **secret**: (pour les clients ne nécessitant pas de login/mot de passe)
- **scope**: Le *scope* permettant de limiter les accès du client (~rôle)
- **authorizedGrantTypes** : Les moyens supportés pour le consentement du user (mot de passe, secret, implicit, ..)
- **authorities**: Permissions accordées aux clients (Permissions standard de Spring Security).



# Exemple configuration *clientDetails*

---

```
@Override
public void configure(ClientDetailsServiceConfigurer clients) throws Exception {

    // @formatter:off
    clients.inMemory()
        .withClient("browser")
        .authorizedGrantTypes("refresh_token", "password")
        .scopes("ui")
        .and()
        .withClient("account-service")
        .secret(env.getProperty("ACCOUNT_SERVICE_PASSWORD"))
        .authorizedGrantTypes("client_credentials", "refresh_token")
        .scopes("server")
        .and()
        .withClient("statistics-service")
        .secret(env.getProperty("STATISTICS_SERVICE_PASSWORD"))
        .authorizedGrantTypes("client_credentials", "refresh_token")
        .scopes("server");
}
```



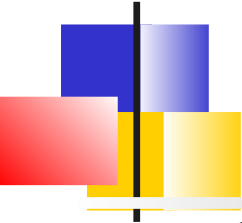
# Choix du *TokenStore*

---

Les jetons fournis au client sont générés et stockés dans des ***TokenStore***

3 implémentations :

- ***InMemoryStore*** : Une fois le jeton obtenu, les serveurs de ressources doivent refaire des requêtes vers le serveur d'autorisation pour vérifier que le jeton est valide
- ***JdbStore*** : Les jetons sont stockés dans une base, le serveur de ressources doit pouvoir y accéder
- ***JwtTokenStore*** : Les jetons sont cryptés et le(s) serveurs de ressource détiennent le moyen de décrypter le jeton et d'en déduire les permissions accordées



# Exemple configuration *TokenStore et gestionnaire d'authentification utilisateur*

---

```
@Bean
    public TokenStore tokenStore() {
        return new JwtTokenStore(accessTokenConverter());
    }

// Sécurisation des endpoints
@Override
public void configure(AuthorizationServerEndpointsConfigurer
    endpoints) throws Exception {
    endpoints
        .tokenStore(tokenStore)
        .authenticationManager(authenticationManager)
        .userDetailsService(userDetailsService);
}
```



# Configuration des ACLs sur les endpoints jeton

---

```
@Override
public void configure(AuthorizationServerSecurityConfigurer
    oauthServer) throws Exception {
    oauthServer
        .tokenKeyAccess("permitAll()")
        .checkTokenAccess("isAuthenticated()");
}
```



# Obtention de jeton

---

Avec la configuration par défaut, le endpoint ***/oauth/token*** permet d'obtenir le jeton typiquement en :

- Utilisant le crédentiel client pour s'authentifier en http sur le endpoint */oauth/token*
- Utilisant le grant type : *password*
- En fournissant les crédentiels de l'utilisateur final

Exemple :

```
curl client:secret@localhost:8080/oauth/token  
-d grant_type=password -d username=user -d password=pwd
```





# Serveur de ressources

---

La création d'un serveur de ressources consiste à annoter

**@EnableResourceServer** et

- De configurer la sécurité (ACLs)
- Si le serveur de ressources est différent du serveur d'autorisation.  
Configurer le moyen pour décoder les jetons.



# ACLs

---

Le jeton contient des informations sur :

- Le client et ses scopes
- L'utilisateur et ses rôles

Des droits d'accès sur les scopes  
peuvent alors être définis de cette  
façon :

```
http.authorizeRequests().anyRequest()  
    .access("#oauth2.hasScope('write')")
```