





Déploiement de micro-services Spring avec Kubernetes

David THIBAU – 2021

david.thibau@gmail.com



Agenda

- **Rappels Spring Cloud / Kubernetes**
 - Kubernetes
 - Support natif SpringBoot
 - Spring Cloud Kubernetes
- **Appels REST**
 - *DiscoveryClient*
 - Service natif Kubernetes
 - Load Balancing
 - Circuit Breaker
- **Outillage**
 - Plugins Maven
 - Skaffold
- **Istio**
 - Présentation Istio
 - Istio et Spring Cloud
- **Configuration centralisée**
 - *ConfigMap*
 - Rechargements dynamiques
 - Secrets



Introduction

Rappel Kubernetes

Support natif SpringBoot
Spring Cloud Kubernetes



Auto-correctif

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

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

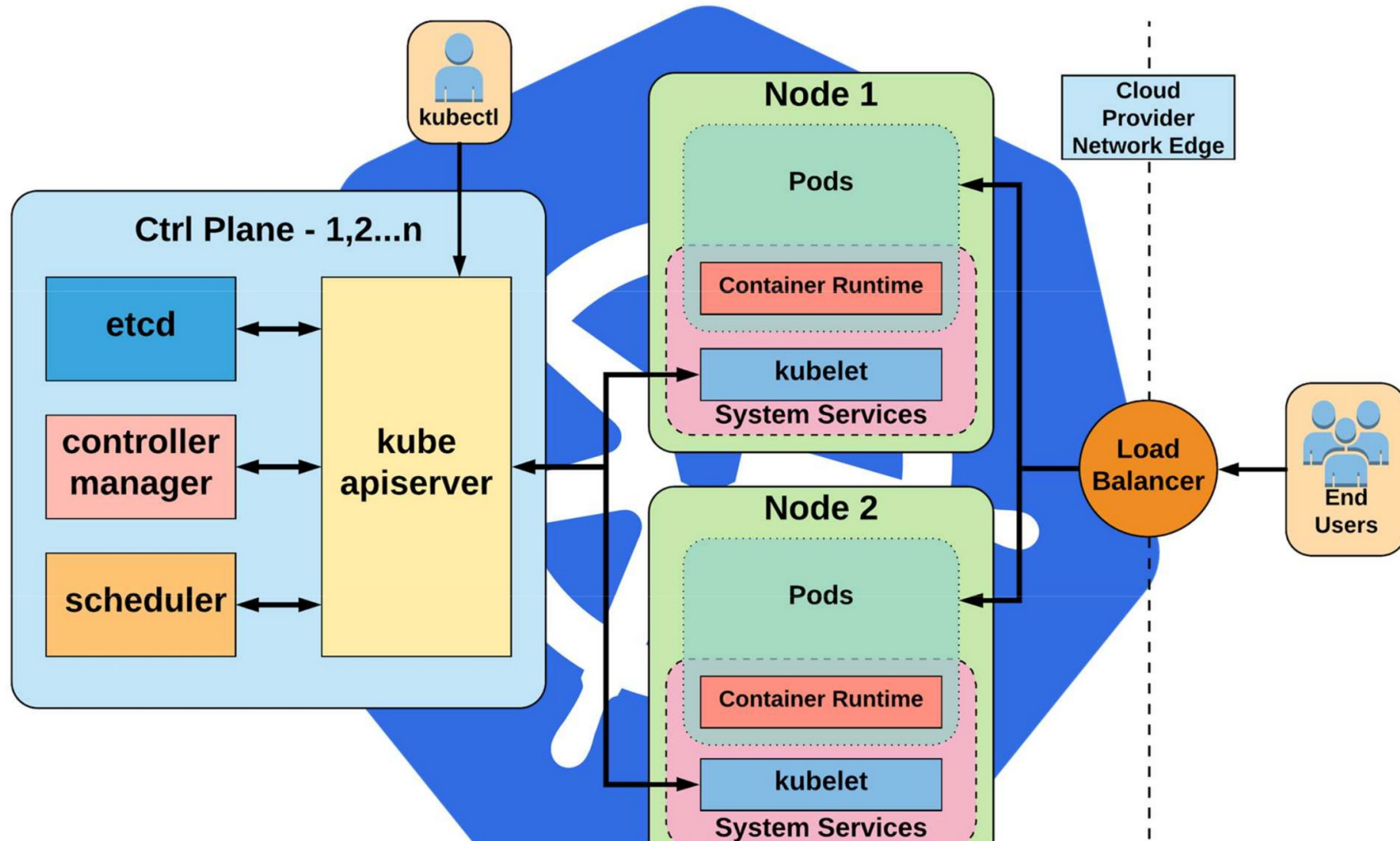


Fonctionnalités applicatives

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

pod = 1 ou plusieurs conteneurs co-localisés

Architecture cluster





API

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

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

Format:

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

Examples:

`/apis/apps/v1/deployments`

`/apis/batch/v1beta1/cronjobs`

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

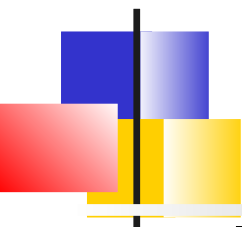


Principes

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

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

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



Ressources applicatives

Les principales ressources d'une application sont :

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

pod

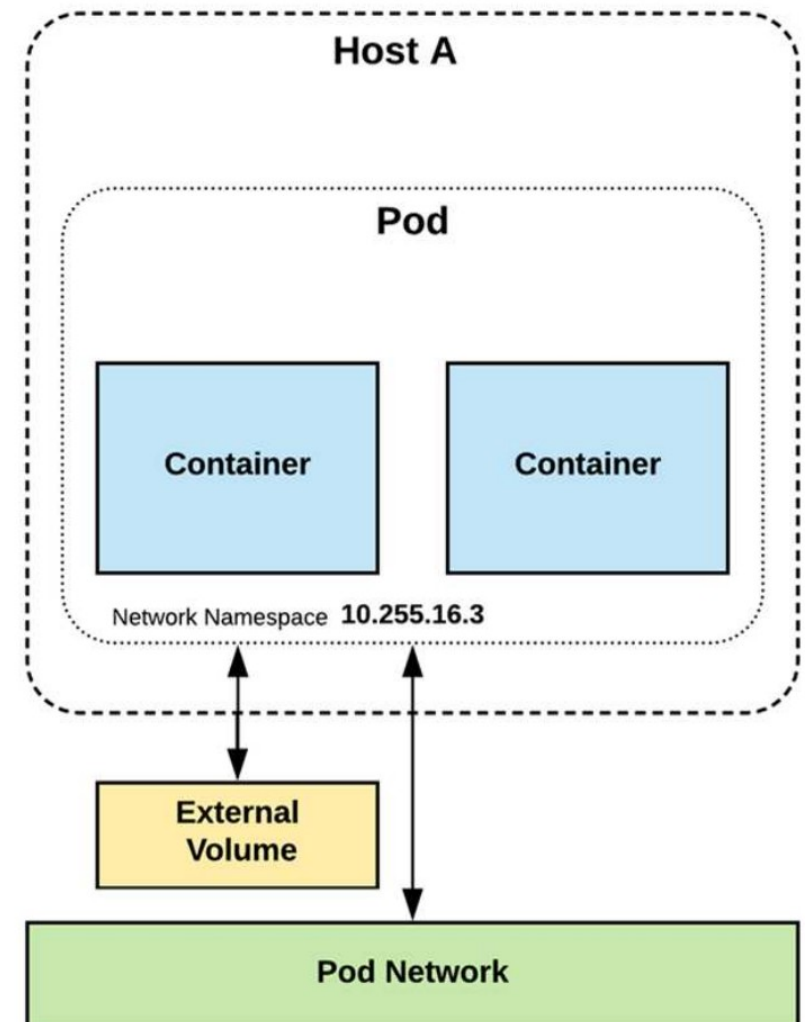
Un **pod** est la plus petite unité de travail

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

- Une adresse réseau
- Les mêmes volumes

Les pods sont éphémères. Ils disparaissent lorsqu'ils :

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





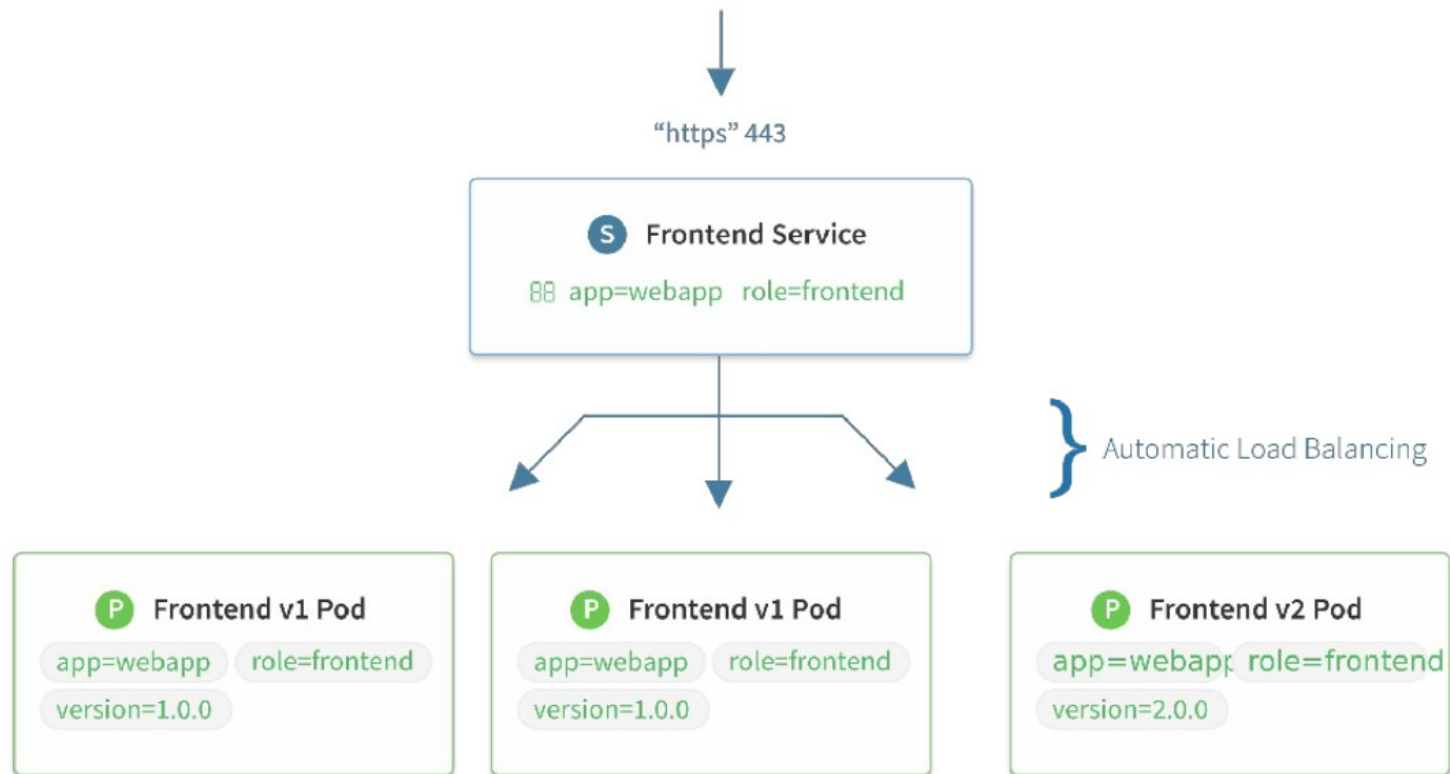
Services

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

Ressource durable. Les services ne sont pas éphémères :

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

Service





Ressource *deployment*

Exemple description d'un déploiement:

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

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

```
kubectl create -f ./my-manifest.yaml
```



Exemple service

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

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



Commandes *kubectl*

create : Crée une ressource à partir d'un fichier ou de stdin.

expose : Exposer un nouveau service

execute : Exécuter une image particulière sur le cluster

set : Mettre à jour des attributs sur une ressource

get : Afficher 1 ou plusieurs ressources

edit : Éditer une ressource

delete : Supprimer des ressources

describe : Afficher les détails sur une ou plusieurs ressources

logs : Afficher les logs d'un container

attach : S'attacher à un container qui s'exécute

exec : Exécuter une commande dans un container

port-forward : Forward un ou plusieurs ports d'un pod

cp : Copier des fichiers entre conteneurs

auth : Inspecter les autorisations

...



Exemples

Affiche les paramètres fusionnés de *kubeconfig*

```
kubectl config view
```

Liste tous les services d'un namespace

```
kubectl get services
```

Liste tous les pods de tous les namespaces

```
kubectl get pods --all-namespaces
```

Description complète d'un pod

```
kubectl describe pods my-pod
```

Supprime les pods et services ayant le noms "baz"

```
kubectl delete pod,service baz
```

Affiche les logs du pod (stdout)

```
kubectl logs my-pod
```

S'attacher à un conteneur en cours d'exécution

```
kubectl attach my-pod -i
```

Exécute une commande dans un pod existant (un seul conteneur)

```
kubectl exec my-pod -- ls /
```

Écoute le port 5000 de la machine locale et forward vers le port 6000 de my-pod

```
kubectl port-forward my-pod 5000:6000
```



La commande *apply*

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

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

```
kubectl apply -f ./my-manifest.yaml
```



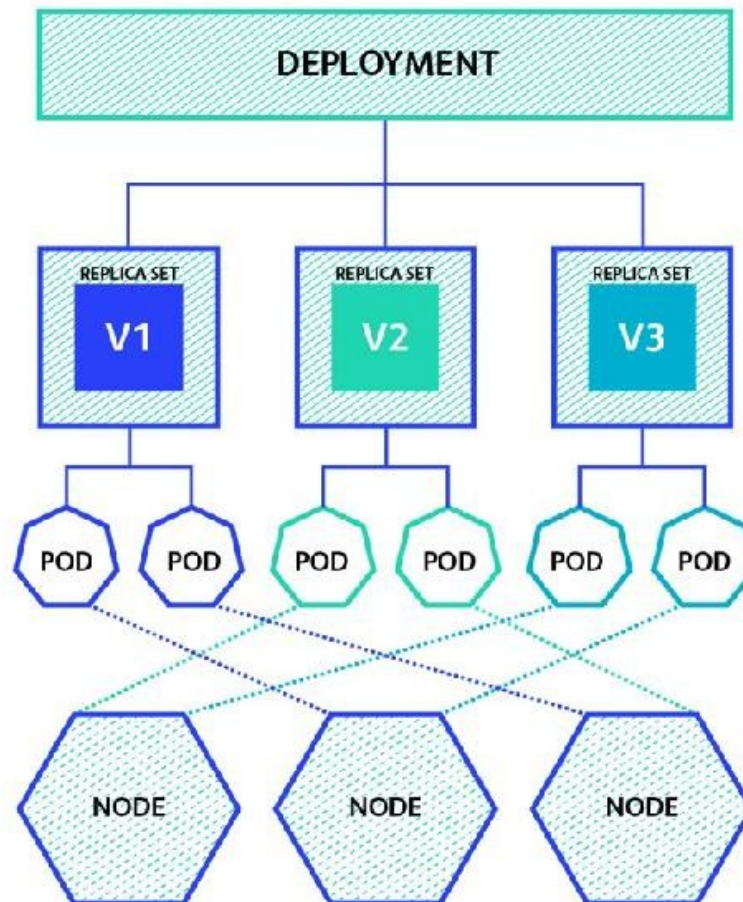
Déploiement

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

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

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

Versions de ReplicaSet





Commandes de déploiement *kubectl*

Mettre à jour une image dans un déploiement existant

Enregistrer la mise à jour

```
kubectl set image deployment/nginx-deployment  
nginx=nginx:1.9.1 -record
```

Regarder le statut d'un rollout

```
kubectl rollout status deployment/nginx-deployment
```

Obtenir l'historique des révisions

```
kubectl rollout history deployment/nginx-deployment
```

Roll-back sur la version précédente

```
kubectl rollout undo deployment/nginx-deployment
```

Scaling

```
kubectl scale deployment/nginx-deployment --replicas=10
```



Scheduler et Workload

Les actions de l'API sont souvent asynchrones

Pour *Kubernetes*, ces ordres sont considérés comme des **workloads** à exécuter via le scheduler.

Les *workload* sont visibles via l'API, elles comportent 2 blocs de données :

- **spec** : La spécification de la ressource
- **status** : Est géré par *Kubernetes* et décrit l'état actuel de l'objet et son historique.



Autres ressources du cluster

ClusterRole : Rôle avec permissions sur l'API

VolumePersistent : Système de stockage

PersistentVolumeClaims : Demande d'usage d'un volume persistant

ConfigMaps : Stockage clé-valeur pour la configuration

Secrets : Stockage de crédeniels



Namespace

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

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

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

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



Labels et sélecteurs

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

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

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

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

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



Annotations

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

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



Écosystème *Kubernetes*

De nombreux outils peuvent compléter une installation cœur de Kubernetes :

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



Distribution Kubernetes

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

Kubernetes est donc proposé par les acteurs du cloud

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

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

Des versions en ligne comme : <https://labs.play-with-k8s.com/>

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

Terraform permet de provisionner des cluster (et services) as Code



Introduction

Rappel Kubernetes
Support natif SpringBoot
Spring Cloud Kubernetes



Sondes Kubernetes et Actuator

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

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

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



Sondes Kubernetes

Le kubelet utilise des sondes et prend des actions en conséquences :

- **Liveness** est utilisé pour redémarrer un conteneur. Par exemple, lorsqu'une application est en cours d'exécution, mais incapable de progresser (deadlock).
- **Readiness** est utilisé pour savoir si un conteneur est prêt à accepter du trafic. Lorsqu'un pod n'est pas prêt, il est supprimé des équilibresurs de charge d'un service.
- **Startup** détermine quand un conteneur a démarré. Les vérifications *liveness* et *readiness* sont désactivées tant que cette sonde n'est pas correcte.



Support SpringBoot

Spring Boot supporte directement les états de disponibilité

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

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

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



Indicateurs de santé des pods

L'indicateur de santé Kubernetes expose les informations suivantes:

- Nom du pod, adresse IP, espace de noms, compte de service, nom de nœud et son adresse IP
- Un indicateur qui indique si l'application Spring Boot est interne ou externe à Kubernetes



Descripteur Kubernetes

livenessProbe:

httpGet:

path: /actuator/health/liveness

port: <actuator-port>

failureThreshold: ...

periodSeconds: ...

readinessProbe:

httpGet:

path: /actuator/health/readiness

port: <actuator-port>

failureThreshold: ...

periodSeconds: ..



Détection automatique d'un environnement Kubernetes

Spring Boot détecte automatiquement les environnements de déploiement Kubernetes en vérifiant les variables d'environnement "***_SERVICE_HOST**" et "***_SERVICE_PORT**".

La détection automatique peut être surchargée via la propriété ***spring.main.cloud-platform***



Arrêt d'une instance

Lors de la suppression d'une instance, plusieurs actions sont exécutées en // : hooks d'arrêt, désinscription du service, suppression de l'instance de l'équilibreur de charge...

=> il y a un risque que du trafic soit routé vers un pod qui a commencé son processus d'arrêt.

Pour éviter ce risque, il est conseillé d'introduire une temporisation dans le hook ***preStop***



Exemple

spec:

containers:

- name: example-container

image: example-image

lifecycle:

preStop:

exec:

command: ["sh", "-c", "sleep 10"]



Introduction

Rappel Kubernetes
Support natif SpringBoot
Spring Cloud Kubernetes



Introduction

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

- *@EnableDiscoveryClient*
- Les objets *PropertySource* configurés via des *ConfigMaps*
- Équilibrage de charge côté client via Spring Cloud LoadBalancer
- ...



Clients Kubernetes

Avec Spring Cloud Kubernetes, 2 client java kubernetes sont possibles :

- Fabric8 Kubernetes Java Client

<https://github.com/fabric8io/kubernetes-client>

Starters commençant par

spring-cloud-starter-kubernetes-fabric8

- Kubernetes Java Client

<https://github.com/kubernetes-client/java>

Starters commençant par

spring-cloud-starter-kubernetes-client



Fonctionnalités vs starters

Implémentation de DiscoveryClient :

spring-cloud-starter-kubernetes-fabric8

spring-cloud-starter-kubernetes-client

Chargement de la config via les ConfigMap et Secret

spring-cloud-starter-kubernetes-fabric8-config

spring-cloud-starter-kubernetes-client-config

Load balancing

spring-cloud-starter-kubernetes-fabric8-loadbalancer

spring-cloud-starter-kubernetes-client-loadbalancer

Toutes les fonctionnalités :

spring-cloud-starter-kubernetes-fabric8-all

spring-cloud-starter-kubernetes-client-all



Exemple DiscoveryClient

```
@SpringBootApplication
@EnableDiscoveryClient
@RestController
public class App {

    @Autowired
    private DiscoveryClient discoveryClient;

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

    @RequestMapping("/")
    public String hello() {
        return "Hello World";
    }

    @RequestMapping("/services")
    public List<String> services() {
        return this.discoveryClient.getServices();
    }
}
```



Client Kubernetes

L'un des apports de SpringCloud et de Spring Cloud Commons est que le code peut fonctionner aussi bien dans un environnement *Kubernetes* qu'en développement

- => Lors du développement d'un micro-service, il n'est pas toujours nécessaire de le déployer sur un environnement *Kubernetes*

Les dépendances sont soit mockées soit déployées sur un cluster de développement



Profils auto-activés

Lors que l'application s'exécute en tant que pod dans un cluster, le profil **kubernetes** est automatiquement activé

- Cela permet d'avoir des différences de configuration avec le profil de développement par exemple

De la même façon, lorsqu'Istio est dans le classpath, le profil **istio** est activé si Spring Cloud détecte que l'on s'exécute sur un cluster Kubernetes avec Istio installé



Sécurité Kubernetes

Pour les distributions de *Kubernetes* qui prennent en charge un accès basé sur les rôles, on doit assurer que les pods aient accès à l'API Kubernetes.

Par exemple, lecture des *ConfigMap*

=> Les comptes de service attribués à un déploiement SCK doivent avoir le rôle approprié



Permissions nécessaires

En fonction des dépendances, il faut les permissions *get*, *list* ou *watch* sur différentes ressources

spring-cloud-starter-kubernetes	Pods, services, endpoints
spring-cloud-starter-kubernetes-config	configmaps, secrets
spring-cloud-starter-kubernetes-ribbon	Pods, services, endpoints



fabric8

Spring Cloud Kubernetes utilise le client Java du projet *fabric8* qui peut offrir des fonctionnalités intéressantes :

- Accès à l'API Kubernetes via un *dsl*
- La possibilité de mocker un cluster Kubernetes



Exemple : Client Java

```
KubernetesClient client = new DefaultKubernetesClient();
// Lister les ressources
NamespaceList myNs = client.namespaces().list();
ServiceList myServices = client.services().list();
ServiceList myNsServices =
    client.services().inNamespace("default").list();
// Get a ressource
Namespace myns = client.namespaces().withName("myns").get();
Service myservice =
    client.services().inNamespace("default").withName("myservice").get();
// Création de ressource
Service myservice = client.services().inNamespace("default").createNew()
    .withNewMetadata()
    .withName("myservice")
    .addToLabels("another", "label")
    .endMetadata()
    .done();
```




Mock Server

Le projet Java Client fournit un serveur mock de Kubernetes pour des tests

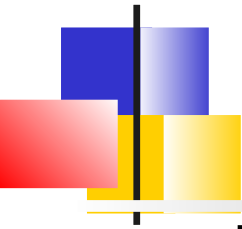
Il propose 2 modes opératoires :

- Expectations : Dans ce mode on spécifie les requêtes HTTP attendus et leurs réponses
- CRUD : Plus simple, permet de faire des requêtes CRUD.
Les ressources Kubernetes sont alors stockées en mémoire



Outillage

Plugins Maven
Skaffold



jkube

Fabric8 proposait également un plugin Maven pour construire et déployer des images. Il a été remplacé par Eclipse ***jkube***

A partir d'un simple *pom.xml*, il est capable :

- De construire des images docker
- De déployer sur OpenShift ou Kubernetes



Installation

Pour utiliser le plugin, il faut modifier le fichier *settings.xml* de Maven

```
<pluginGroups>  
  <pluginGroup>org.eclipse.jkube</pluginGroup>  
</pluginGroups>
```

Puis référencer et éventuellement configurer le plugin dans le *pom.xml*

```
<plugin>  
  <groupId>org.eclipse.jkube</groupId>  
  <artifactId>kubernetes-maven-plugin</artifactId>  
  <version>1.1.0</version>  
</plugin>
```



Construction d'images

L'objectif **k8s:build** crée des images Docker.

- Il utilise le descripteur d'assemblage de *maven-assembly-plugin* pour spécifier le contenu ajouté à l'image

Les images peuvent être poussées sur les dépôts via **k8s:push**

L'objectif **k8s:watch** permet de réagir à des changements de code pour reconstruire automatiquement les images



Ressources Kubernetes

Les descripteurs de ressources Kubernetes peuvent être créés ou générés à partir de **k8s:resource**

Ces fichiers sont regroupés dans les artefacts Maven et peuvent être déployés avec **k8s:apply**.



Configuration

3 niveaux de configuration sont possibles :

- Zéro-config : prend des décisions en fonction de ce qui est présent dans le *pom.xml* comme l'image de base à utiliser ou les ports à exposer. Idéal pour démarrer
- Configuration XML : Similaire à *docker-maven-plugin* .
- Fragments de ressources : Permettant de fournir des fragments YAML enrichis par le plugin



Exemple : Configuration XML

```
<configuration>
  <images>
    <image>
      <alias>service</alias>
      <name>fabric8/docker-demo:${project.version}</name>

      <build>
        <from>java:8</from>
        <assembly> <descriptor>docker-assembly.xml</descriptor>          </assembly>
        <cmd>  <shell>java -jar /maven/service.jar</shell>          </cmd>
      </build>

      <run>
        <ports>  <port>tomcat.port:8080</port> </ports>
        <wait>
          <http>
            <url>http://localhost:${tomcat.port}/access</url>
          </http>
          <time>10000</time>
        </wait>
        <links>
          <link>database:db</link>
        </links>
      </run>
    </image>
```




Exemple : Configuration XML

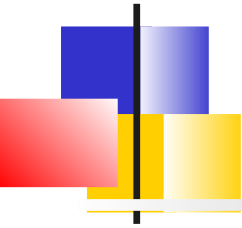
...

```
<image>
  <alias>database</alias>
  <name>postgres:9</name>
  <run>
    <wait>
      <log>database system is ready to accept
connections</log>
      <time>20000</time>
    </wait>
  </run>
</image>
</images>
</configuration>
```



Exemple : Déploiement

```
mvn clean install  
k8s:deploy -Pkubernetes
```



Outillage

Plugins Maven
Skaffold



Scaffold

Scaffold est un outil simple en ligne de commande capable de gérer le workflow pour créer, pousser et déployer votre application sur Kubernetes.

Il s'appuie sur :

- un fichier de configuration *scaffold.yaml*
- un fichier de ressource Kubernetes
k8s/deployment.yaml



Utilisation

Après avoir démarré une instance *minikube*,
il suffit d'exécuter à la racine du projet
skaffold dev

Cette commande

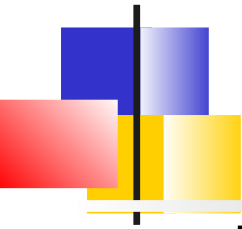
- crée une image Docker applicative
- la déploie sur Minikube.
- Surveille toute modification du code source et déclenche une nouvelle version après chaque modification du système de fichiers



Options

La commande prend différents paramètres :

- **--port-forward** : Exécute *kubectrl port-forward* pour tous les ports exposés du container
- **--trigger=manual** : Permet de désactiver la reconstruction automatique
- **--no-prune** : Pour ne pas supprimer les images à l'arrêt de *skaffold*



Debug

Pour permettre le debug :

scaffold debug

La commande démarre un agent de debug exposé sur le port 8005

Il suffit alors de configurer son IDE à ce port pour pouvoir positionner des breakpoints



Jib

Jib dédié uniquement aux applications Java permet de créer des images Docker optimisées sans démon Docker.

Il est disponible en tant que plugin Maven ou Gradle, ou simplement en tant que bibliothèque Java

Il s'intègre facilement avec *Skaffold*



Configuration

ConfigMap

Secrets

Rechargement dynamique



Introduction

Les ressources ***ConfigMap*** de Kubernetes peuvent être utilisées :

- Pour définir des paires clés valeurs de configuration
- Pour embarquer des fichiers complets de configuration (*application.properties* ou *application.yml*)

Les valeurs de configuration sont lues au *bootstrap* et peuvent être rechargées en cours d'exécution lors de la détection de changement



Mécanisme

Starter : `spring-cloud-starter-kubernetes-config`

L'objet *ConfigMapPropertySource* est chargé à partir d'une ressource Kubernetes ConfigMap dont le ***metadata.name*** est identique

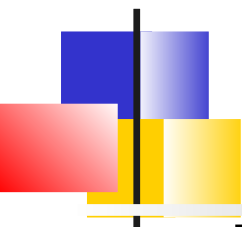
- à la propriété de bootstrap
spring.application.name
- Ou à la propriété spécifique :
spring.cloud.kubernetes.config.name

Des configurations plus avancées sont possibles comme l'utilisation de plusieurs *ConfigMap*



Configuration avancée

```
spring:
  application:
    name: cloud-k8s-app
  cloud:
    kubernetes:
      config:
        name: default-name
        # namespace par défaut, si non définie => le namespace de l'appli
        namespace: default-namespace
      sources:
        # Recherche d'une ConfigMap c1 dans le namespace default-namespace
        - name: c1
        # Recherche d'une ConfigMap nommée default-name dans le namespace n2
        - namespace: n2
        # Recherche ConfigMap c3 dans le namespace n3
        - namespace: n3
          name: c3
```



Application des *ConfigMap*

Pour chaque *ConfigMap* retrouvée,

- Les propriétés individuelles de configuration sont appliquées
- Si la propriété s'appelle *application.yml*, son contenu est traité comme un fichier *.yml*
- Si la propriété s'appelle *application.properties*, son contenu est traité comme un fichier *.properties*

La seule exception à ce comportement est si la *ConfigMap* ne contient qu'une seule clé, dans ce cas la valeur est traité comme un format *.yaml* ou *.properties*.

=> Facilitation de :

```
kubectl create configmap game-config --from-file=/path/to/app-config.yaml
```



Exemples

Valeurs individuelles

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  pool.size.core: 1
  pool.size.max: 16
```

Simple clé

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: demo
data:
  custom-name.yaml: |-
    pool:
      size:
        core: 1
        max: 16
```



Profils

La gestion des différents profils peut être effectuée :

- A l'intérieur du contenu *.yml* présent dans la *ConfigMap*
- En définissant plusieurs *ConfigMap* par profils.

Exemple :

```
monService-dev  
monService-prod
```



Activation d'un profil

Pour activer un profil sur un pod, le plus simple est de positionner la variable d'environnement **SPRING_PROFILES_ACTIVE**

```
apiVersion: apps/v1
kind: Deployment
```

```
...
```

```
  spec:
    containers:
      - name: container-name
        image: your-image
        env:
          - name: SPRING_PROFILES_ACTIVE
            value: "development"
```




Droits de lecture sur la *ConfigMap*

Pour que le mécanisme soit effectif, il faut que les pods aient les permissions de lecture des *ConfigMap*

Si RBAC est activée dans le cluster,

- Créer un *ClusterRole*
- L'associer via un *ClusterRoleBinding* au compte du service



Exemple

Création du ClusterRole

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-discovery-client
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["services", "pods", "configmaps", "endpoints"]
  verbs: ["get", "watch", "list"]
```

Association à default:default

```
kubectl create rolebinding default:service-discovery-client
--clusterrole service-discovery-client
--serviceaccount default:default
```



Montage de volume

Une autre option est de monter les *ConfigMap* comme volume et de configurer les pods afin qu'ils lisent leur config via le système de fichier.

Dans ce cas, il faut utiliser la propriété :
`spring.cloud.kubernetes.config.path`



Configuration

ConfigMap
Rechargement dynamique
Secrets



Introduction

La fonctionnalité de rechargement de *Spring Cloud Kubernetes* est capable de déclencher un rechargement des propriétés lorsqu'une *ConfigMap* ou un *Secret* change.

Il est nécessaire d'activer la fonctionnalité via :

`spring.cloud.kubernetes.reload.enabled`



Options de rechargement

Différents niveaux de rechargement sont possibles en fonction de la propriété **`spring.cloud.kubernetes.reload.strategy`**

- ***refresh*** (défaut) : Seules les beans annotés par *@ConfigurationProperties* ou *@RefreshScope* sont rechargés
- ***shutdown*** : Le contexte Spring est redémarré



Mode opératoire

2 modes opératoires sont supportés :

- **Event** (défaut) : Surveille les changements via l'API Kubernetes et des socket Web. Tout événement provoque une détection de changement. Le rôle *view* est requis pour ConfigMap, *edit* pour Secret
- **Polling** : Recrée périodiquement la configuration afin de détecter les changements. Ne nécessite pas de privilèges supplémentaires



Configuration

ConfigMap
Rechargement dynamique
Secrets



Introduction

Kubernetes propose les ressources ***Secrets*** pour stocker les informations sensibles (mot de passe, etc.)

Les secrets peuvent être accédés par les pods *SpringCloud*

Cette fonctionnalité est activée avec **`spring.cloud.kubernetes.secrets.enabled`**



Mécanisme

Si la fonctionnalité est activée, l'interface ***SecretsPropertySource*** recherche des secret *Kubernetes* à partir de :

- Lecture récursive des volumes secrets montés
- Secret nommé comme l'application (*spring.application.name*)
- Des secrets correspondant à des labels particuliers



Recommandation

En termes de sécurité, il est recommandé d'utiliser les volumes.

Les autres méthodes utilisent l'API, il faut donc ouvrir l'API à la lecture d'informations sensibles
=> Utilisation de RBAC



Création d'un secret

création

```
kubectl create secret generic db-secret  
  --from-literal=username=user --from-literal=password=p455w0rd
```

GET API

```
apiVersion: v1  
data:  
  password: cDQ1NXcwcmQ=  
  username: dXNlcg==  
kind: Secret  
metadata:  
  creationTimestamp: 2017-07-04T09:15:57Z  
  name: db-secret  
  namespace: default  
  resourceVersion: "357496"  
  selfLink: /api/v1/namespaces/default/secrets/db-secret  
  uid: 63c89263-6099-11e7-b3da-76d6186905a8  
type: Opaque
```



Utilisation

Utilisation des secrets comme variable d'environnement

```
apiVersion: v1
kind: Deployment
metadata:
  name: ${project.artifactId}
spec:
  template:
    spec:
      containers:
        - env:
            - name: DB_USERNAME
              valueFrom:
                secretKeyRef:
                  name: db-secret
                  key: username
            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-secret
                  key: password
```



Montage de volume

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
```



Spécification du secret

Le secret à utiliser peut être spécifié de différentes façons :

Lister les répertoires ou les secrets sont mappés

-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets/db-secret,etc/secrets/postgresql

Ou indiquer un répertoire racine

-Dspring.cloud.kubernetes.secrets.paths=/etc/secrets

En donnant le nom du secret

-Dspring.cloud.kubernetes.secrets.name=db-secret

En définissant une liste de labels

-Dspring.cloud.kubernetes.secrets.labels.broker=activemq

-Dspring.cloud.kubernetes.secrets.labels.db=postgresql



Appels REST

DiscoveryClient

Service natif Kubernetes

LoadBalancing

CircuitBreaker



Introduction

L'implémentation de *DiscoveryClient* pour Kubernetes permet de découvrir les points d'accès Kubernetes par leur nom

Seule dépendance nécessaire :
spring-cloud-starter-kubernetes

Et pour autoriser le chargement du Bean
@EnableDiscoveryClient



Utilisation

Une fois la configuration mise en place, on peut s'injecter le bean et utiliser son API

```
@Autowired
```

```
private DiscoveryClient discoveryClient;
```

```
// Récupérer les instances associés
```

```
// à un service et leurs informations
```

```
discoveryClient.getInstances("serviceId")
```

Naturellement, il faut aligner le nom de l'application avec le nom du service *Kubernetes*



Enregistrement de service

Avec Kubernetes, l'enregistrement est contrôlé par la plateforme,

- l'application ne s'enregistre pas elle-même comme sur d'autres plateformes (Eureka par exemple).

=>

```
spring.cloud.service-registry.auto-registration.enabled
```

ou

```
@EnableDiscoveryClient (autoRegister = false)
```

n'ont aucun effet dans Spring Cloud Kubernetes.



Découverte de service

DiscoveryClient
Service natif Kubernetes
LoadBalancing
CircuitBreaker



Introduction

Kubernetes lui-même est capable de découvrir des services (côté serveur)

L'utilisation de service natif garantit la compatibilité avec des outils de l'écosystème *Kubernetes* (*Istio* par exemple)



FQDN

Le service appelant n'a besoin que de se référer aux noms pouvant être résolus dans un cluster *Kubernetes* particulier.

Une implémentation simple peut utiliser un *RestTemplate* faisant référence à un nom de domaine complet (FQDN) :

```
{service-name}. {Namespace} .svc.{Cluster} .local: {service-port}.
```



Découverte de service

DiscoveryClient
Service natif Kubernetes
Load Balancing
CircuitBreaker



Introduction

Spring Cloud Kubernetes Load Balancer a été ajoutée dans la version de Spring cloud Hoxton.SR8.

Il remplace Ribbon comme équilibreur de charge côté client.

L'implémentation est basée sur le projet Spring Cloud LoadBalancer qui fournit 2 modes de communication :

- Soit il détecte les adresses IP de tous les pods exécutés dans un service donné.
- Soit il utilise le nom du service Kubernetes pour rechercher toutes les instances cibles.

Il peut être utilisé avec OpenFeign ou RestTemplate (@LoadBalanced)



Mode POD

Par défaut, Spring Cloud Kubernetes Load Balancer utilise le mode POD.

Dans ce mode, il obtient la liste des points de terminaison Kubernetes pour détecter l'adresse IP de tous les pods d'application

Il est cependant nécessaire de désactiver Ribbon qui est l'implémentation par défaut :

```
spring:
  cloud:
    loadbalancer:
      ribbon:
        enabled: false
```



Mode SERVICE

Pour activer le mode SERVICE

```
spring:  
  cloud:  
    kubernetes:  
      loadbalancer:  
        mode: SERVICE
```

Le load balancer essaie alors d'atteindre
le service via son FQDN



Découverte de service

DiscoveryClient
Service natif Kubernetes
Load Balancing
CircuitBreaker



Introduction

Il est possible d'appliquer le pattern
CircuitBreaker avec Spring Cloud Load
Balancer.

L'implémentation par défaut est basée sur
Resilience4j.

Pour l'activer :

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-circuitbreaker-  
    resilience4j</artifactId>  
</dependency>
```



Configuration

La configuration (timeout, sliding window, etc) peut s'effectuer en fournissant un bean

Customizer<Resilience4JCircuitBreakerFactory>

@Bean

```
Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {  
    return factory -> factory.configureDefault(id ->  
        new Resilience4JConfigBuilder(id)  
            .timeLimiterConfig(TimeLimiterConfig.custom()  
                .timeoutDuration(Duration.ofMillis(1000))  
                .build())  
            .circuitBreakerConfig(CircuitBreakerConfig.custom()  
                .slidingWindowSize(10)  
                .failureRateThreshold(66.6F)  
                .slowCallRateThreshold(66.6F)  
                .build())  
            .build()  
    );  
}
```



Usage

```
@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");
    }

}
```



Istio

Concepts Istio

Istio et Spring Cloud



Introduction

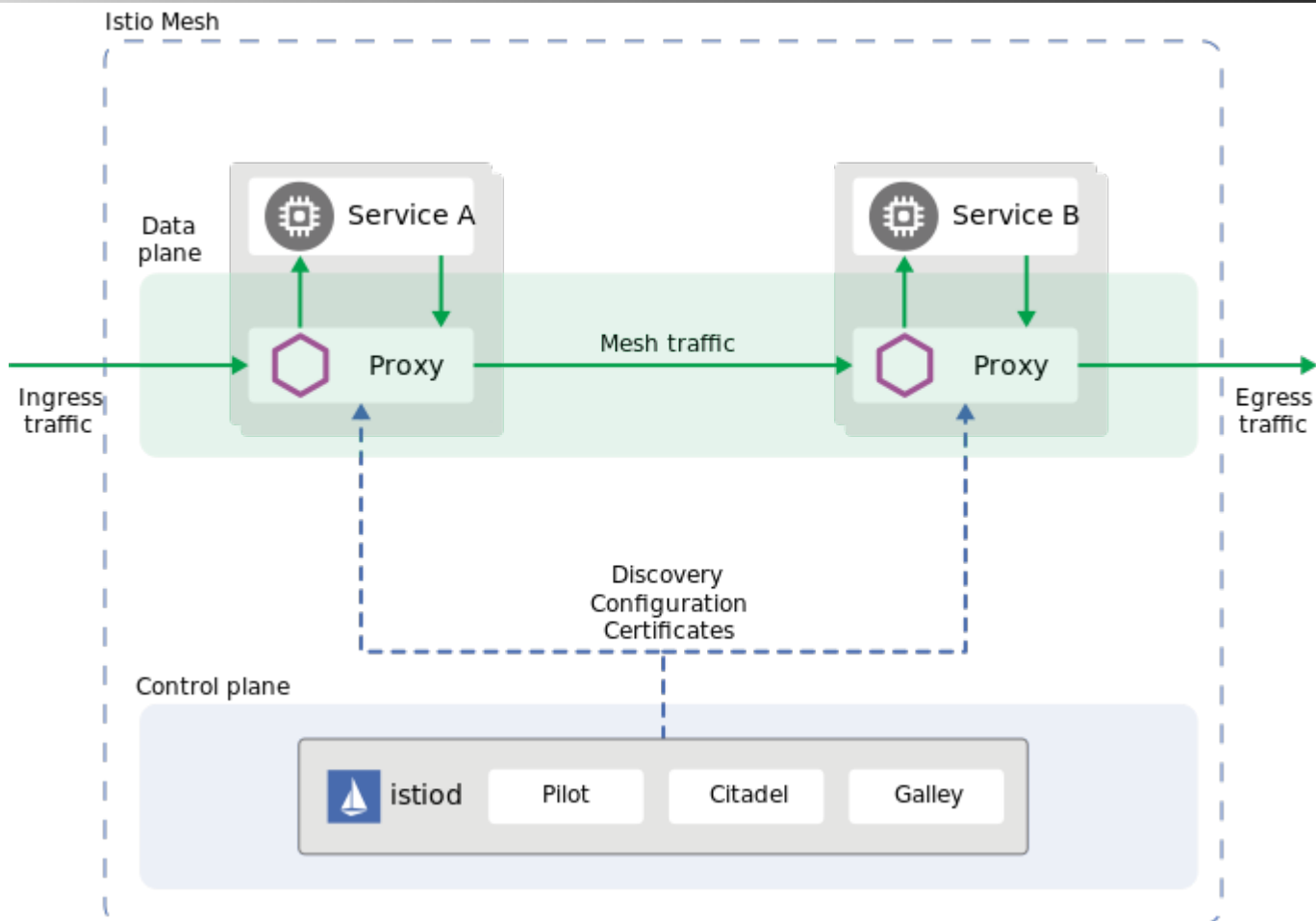
Istio est un ***service mesh*** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

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

Les fonctionnalités apportées sont nombreuses :

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

Architecture





Istio.io

Gestion de trafic: Contrôler le flux d'appels entre service, exécuter des tests, déploiement *red/black*

Sécurité : Authentification, Autorisations et chiffrement des communications entre services

Control : Appliquer des stratégies garantissant la distribution de la charge

Observabilité : Tracing, Surveillance et logs



Service Registry

Istio renseigne son propre registre de services à partir des endpoints de Kubernetes (service)

Grâce à son registre de services, les proxy Envoy peuvent router le trafic vers les endpoints Kubernetes

Une stratégie Round-robin est appliqué par défaut mais on peut contrôler très finement les règles de routage :

- Rediriger un pourcentage des requêtes vers de nouvelles versions
- Avoir différentes stratégie de répartition en fonction de sélecteurs Kubernetes
- Gérer les trafic externes au service mesh
- ...

L'API utilise les ressources personnalisés Kubernetes



Ressources liées à la gestion de trafic

Virtual services : Permet de configurer comment les requêtes sont routées.

Destination rules : Stratégie de répartition de charge, Sécurisation, Circuit breaker

Gateways : Trafic extérieur au service mesh.
(proxy envoy autonome)

Service entries : Une entrée dans le registre de service de istio

Sidecars : Configuration du proxy envoy sidecar



Exemple Virtual Service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
    # Services, IP de destination
  http:
    - match:
        - headers:
            end-user:
              exact: jason
        # Règles de routage
        # 1 règle sur les entêtes http
      route:
        - destination:
            host: reviews
            subset: v2
        # Route par défaut
      - destination:
          host: reviews
          subset: v3
```



Règles de destination

Les **règles de destination** sont appliquées après les règles de routage définies par les virtual services

Elles configurent des sous ensembles de service (par exemple, grouper les instances par version) qui peuvent être utilisées dans les règles de routage.

Elles permettent également de configurer

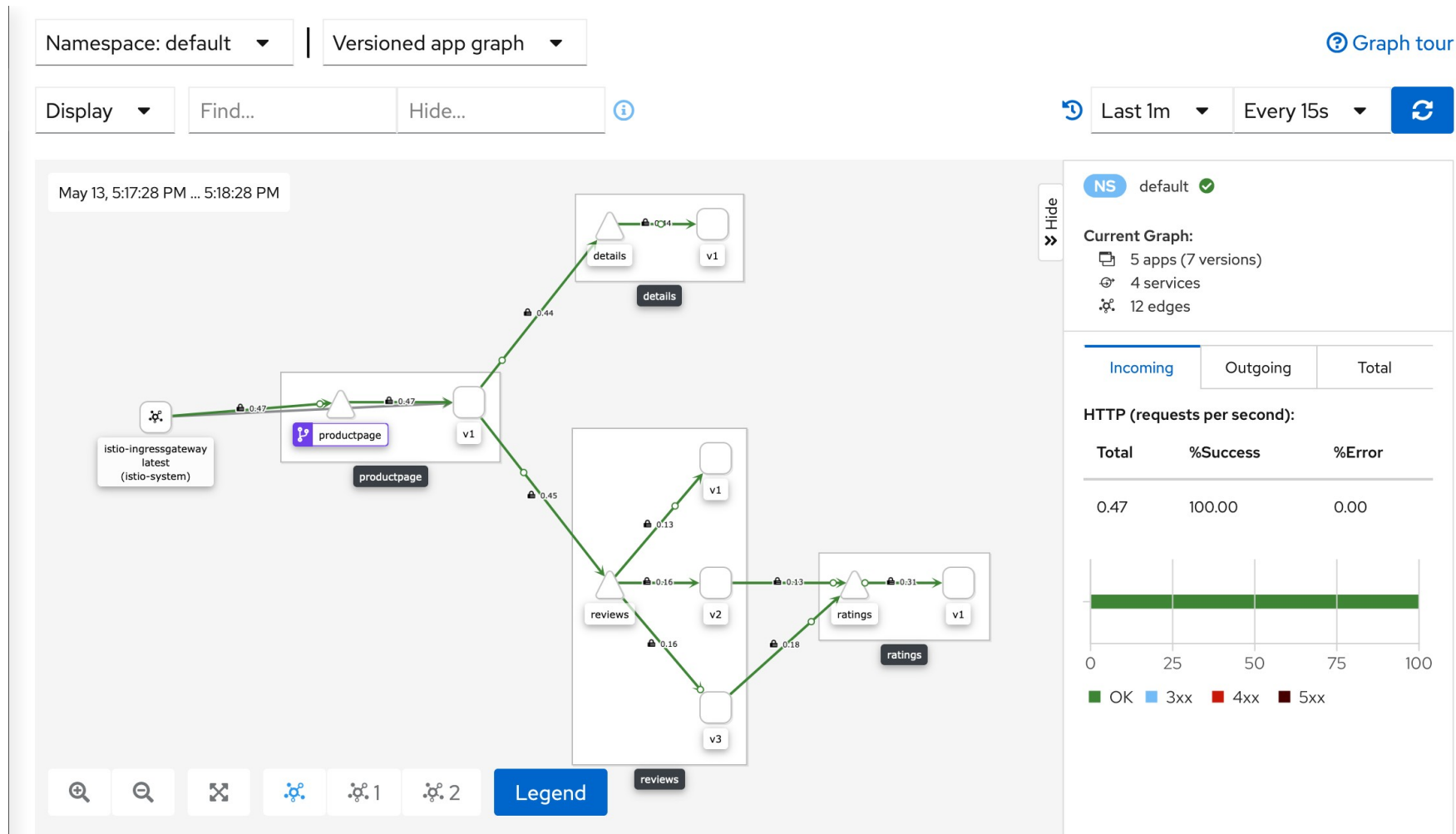
- la stratégie de répartition sur la destination finale :
Round-robin, Random, Pondérée, Moindre requête
- La résilience : Circuit-Breaker, Retry, Timeout, Fault-Injection



Règles de destination

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: my-svc
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN
    - name: v3
      labels:
        version: v3
```

Tableau de bord Kiali





Istio et Spring Cloud



Introduction

Lorsque la dépendance

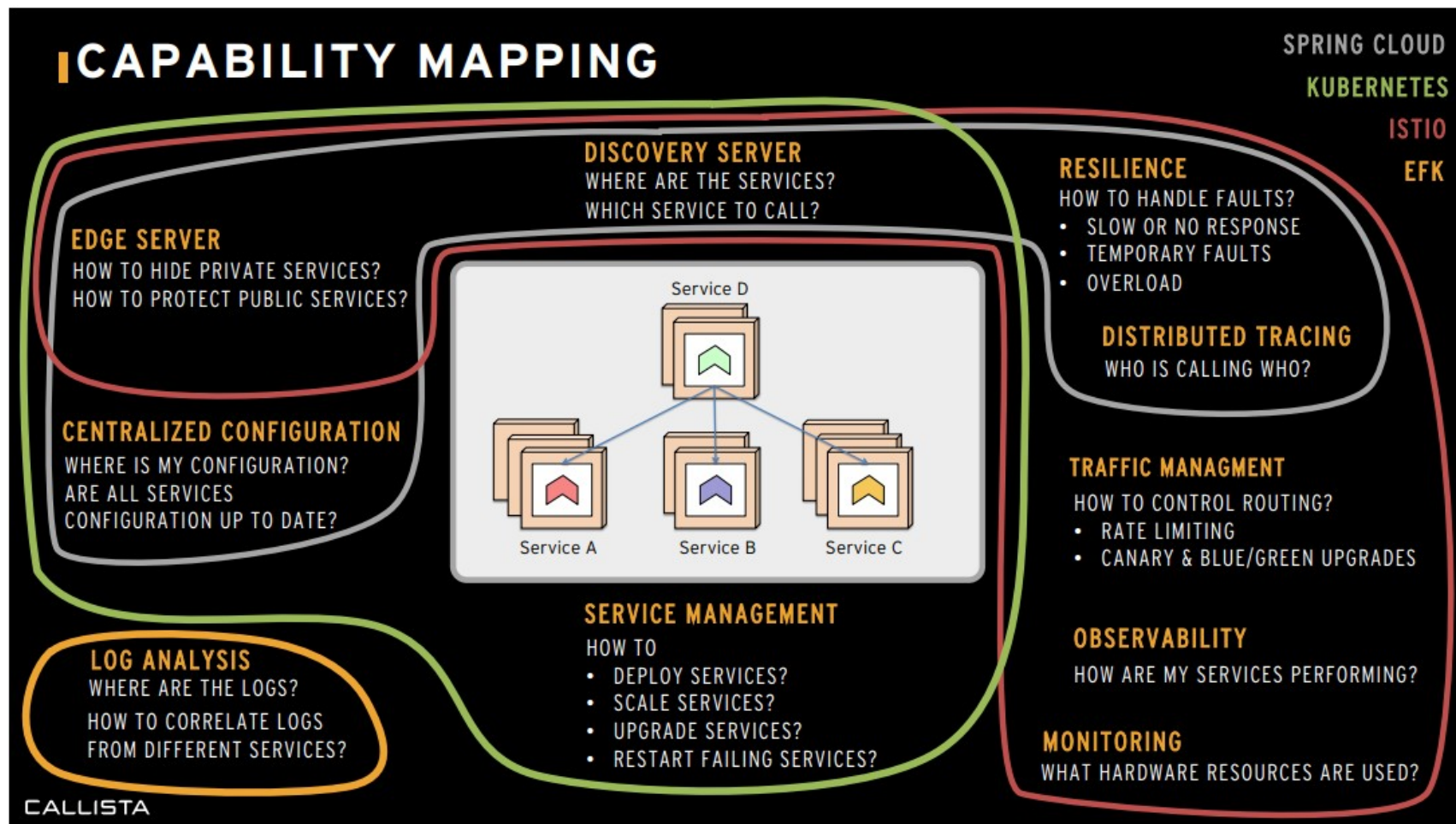
spring-cloud-kubernetes-fabric8-istio

est présente, la présence d'Istio est détectée et un nouveau profil *istio* activé

Le module utilise ***me.snowdrop:istio-client*** pour interagir avec Istio.

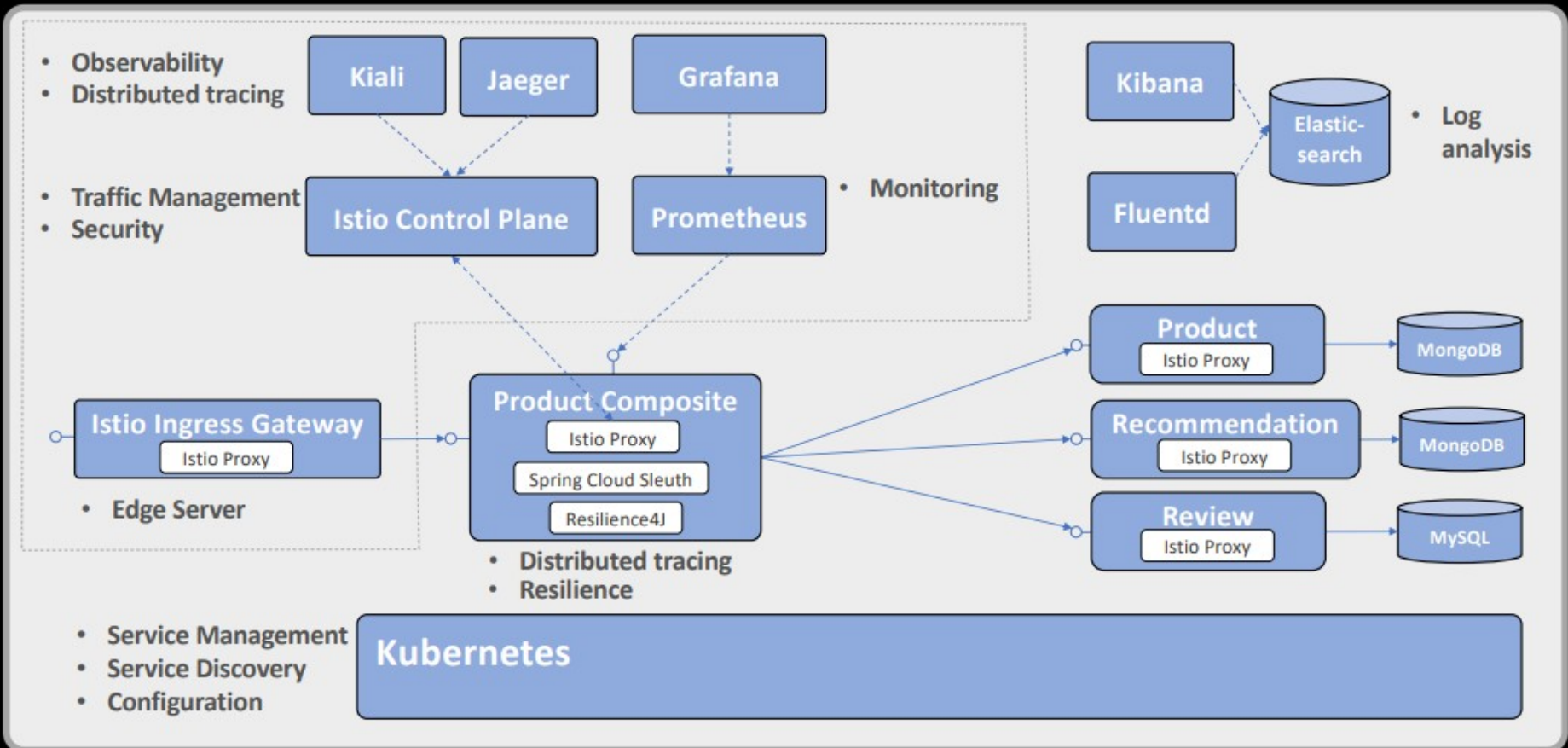
Il est possible alors de découvrir les règles configurées dans Istio et agir en conséquence

Service requis



Un exemple de choix

SPRING CLOUD + KUBERNETES + ISTIO





Annexes



Introduction

Architectures micro-services

Services techniques frameworks ou
infrastructure

L'offre Spring Cloud

Kubernetes

Spring Cloud Kubernetes



Introduction

Le terme « ***micro-services*** » décrit un nouveau pattern de développement 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*



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...)

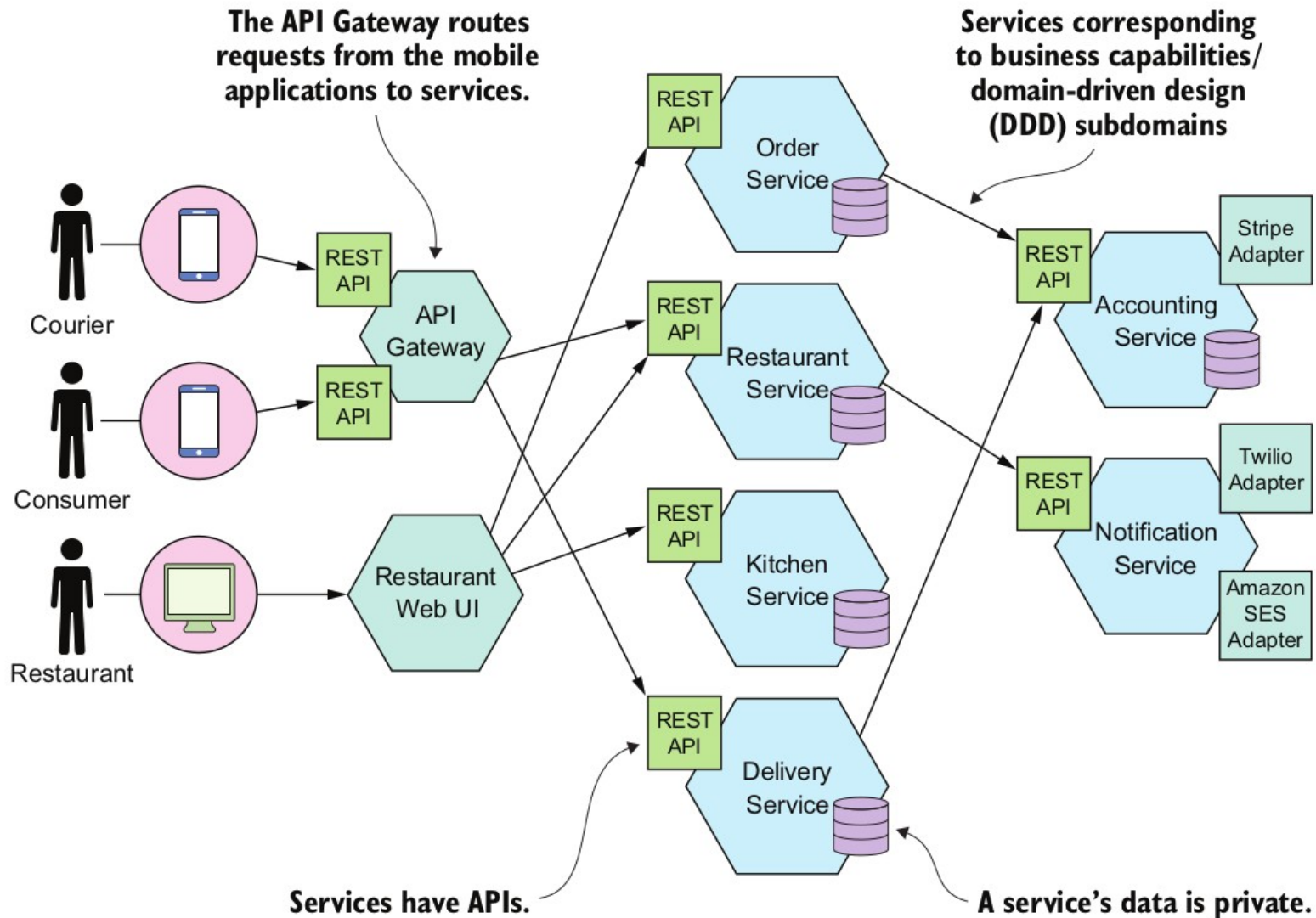
Découverte automatique : Les services sont typiquement distribués dans l'environnement d'exécution, le scaling peut être automatisé. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

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

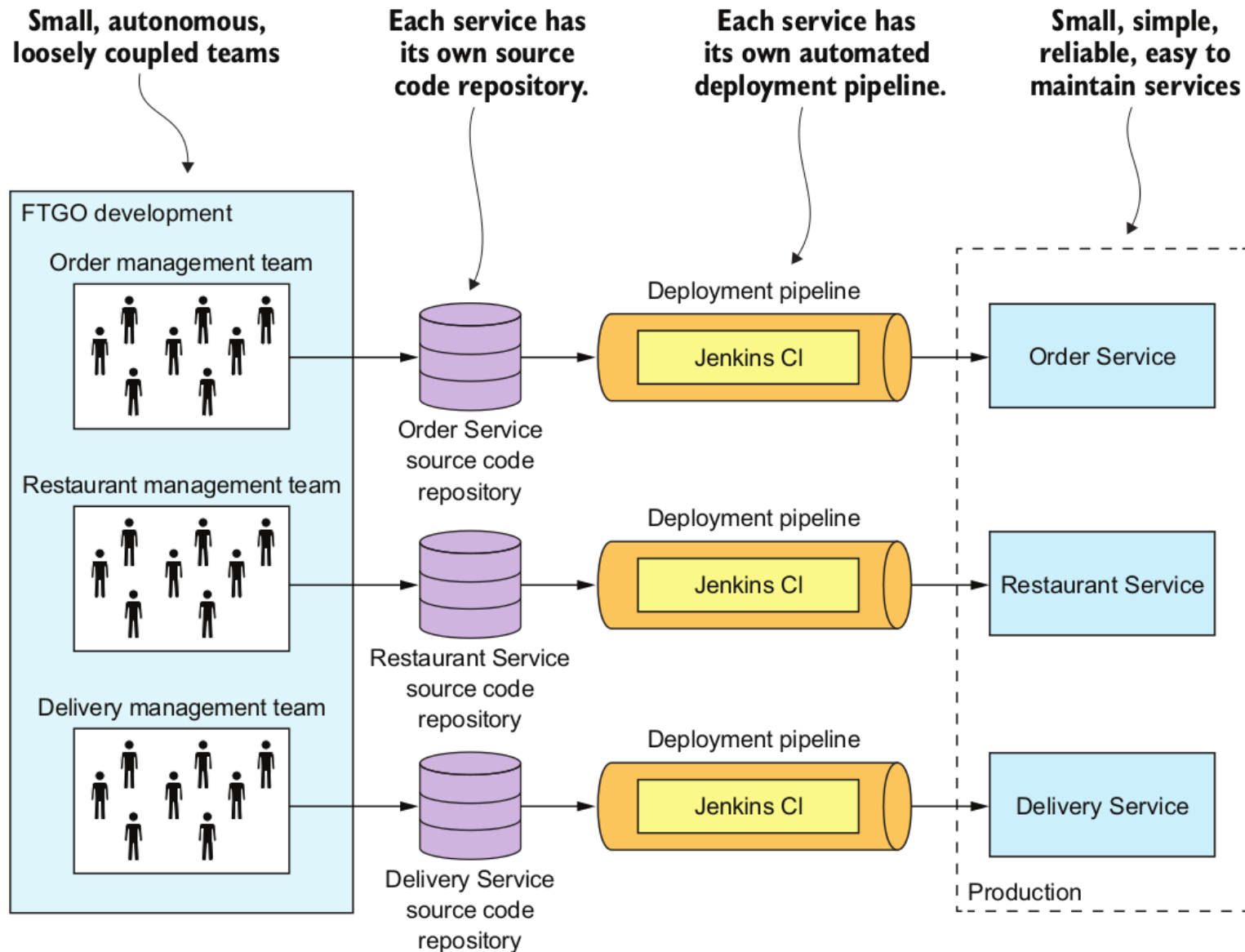
Résilience : Plus de services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

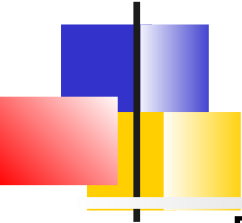
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.

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*, ...



Services techniques :
Framework ou infrastructure ?

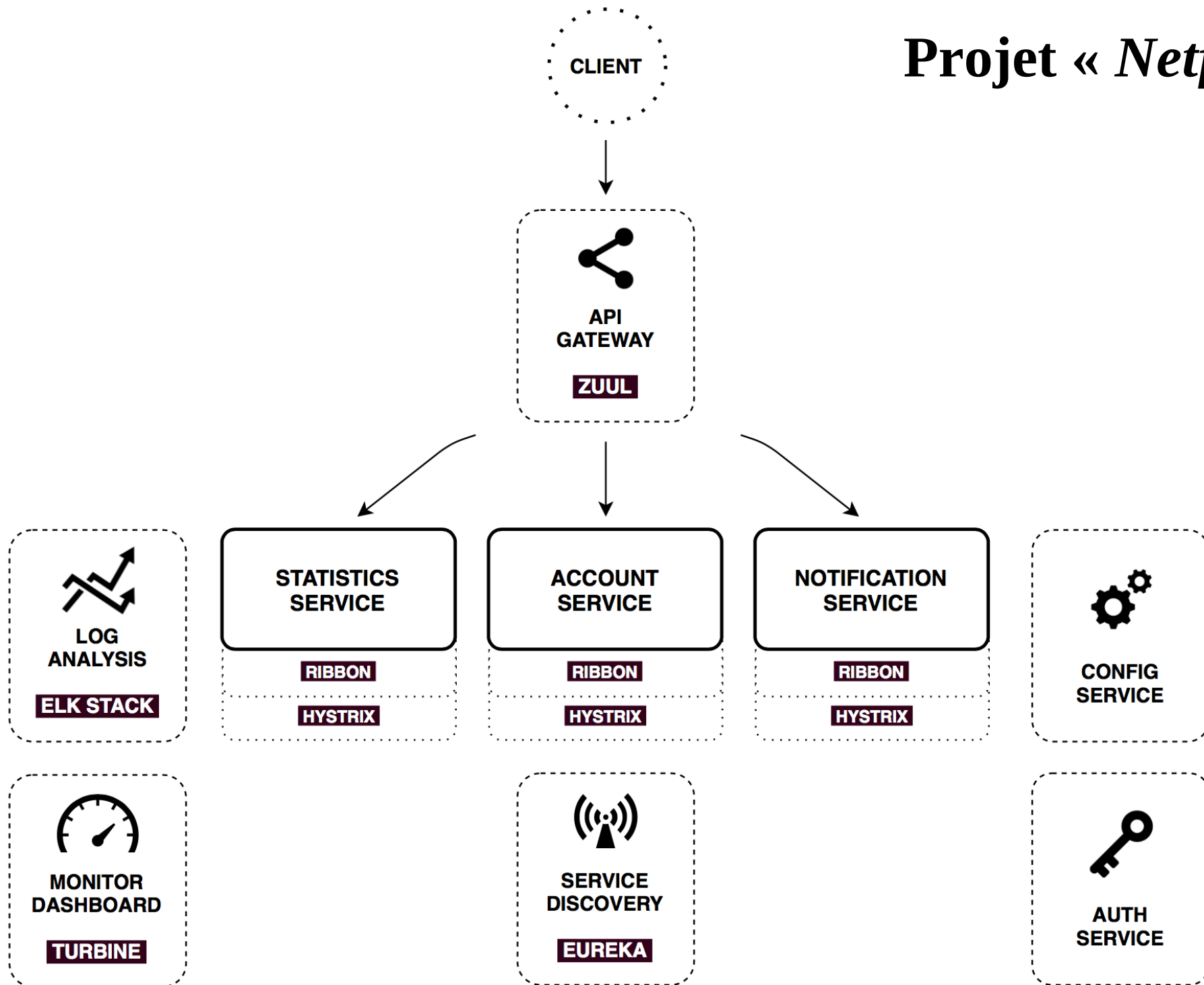
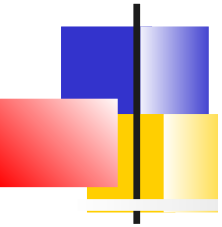


Services techniques

Les architectures micro-services nécessitent des services techniques :

- Service **de discovery**
- Service de centralisation de **configuration**
- Services **d'authentification**
- Service de **monitoring** agrégeant les métriques de surveillance en un point central
- Support pour la répartition de charge, le fail-over, la résilience
- Service de session applicative, horloge synchronisée,
...

Projet « Netflix »





Infrastructure de déploiement

Infrastructure de déploiement pour ce type d'architecture :

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



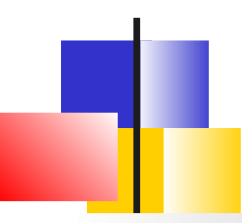
Services techniques

Qui fournit les services techniques ?

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



L'offre Spring Cloud



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.



Exemple : *bootstrap.yml*

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




Spring Cloud Commons

Abstractions et implémentations

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

LoadBalancer : Ribbon, SpringRestTemplate, Reactive Web Client

Circuit Breaker : Hystrix, Resilience4J, Sentinel, Spring Retry



Découverte de service

Ribbon et Hystrix



Projet

Les clients de micro-service peuvent être intéressés par une répartition de charge côté client

spring-cloud-kubernetes-ribbon
renseigne l'objet *Ribbon ServerList* à partir des points d'accès trouvés via l'API de Kubernetes



Mise en place

Le client *Kubernetes* recherche les points d'accès enregistrés s'exécutant dans l'espace de noms ou le projet en cours.

Il fait correspondre le nom du service recherché avec l'annotation du client Ribbon.

```
@RibbonClient(name = "name-service")
```

Le comportement du *LoadBalancer* peut être précisé avec les propriétés

```
<name-service>.ribbon.<Ribbon configuration key>
```



Autres configurations

`spring.cloud.kubernetes.ribbon.enabled`

- Activation/désactivation

`spring.cloud.kubernetes.ribbon.mode`

- POD (défaut) : Accès direct aux pods, LoadBalancing de Ribbon et non pas de Kubernetes, (Istio est également court-circuité)
- SERVICE : Accès via service Kubernetes, LoadBalancing via le service Kubernetes, Istio n'est pas court-circuité

`spring.cloud.kubernetes.ribbon.cluster-domain`

- Définit le suffixe de domaine de cluster Kubernetes. par défaut : 'cluster.local'



Hystrix et Circuit Breaker

2 options pour implémenter Hystrix et Circuit Breaker

- Directement via l'annotation *@HystrixCommand*
- Via OpenFeign (Nécessite Ribbon)



Circuit breaker

```
@HystrixCommand(fallbackMethod = "getFallbackName",
commandProperties = { @HystrixProperty(
name = "execution.isolation.thread.timeoutInMilliseconds",
value = "1000") })
public String getName(int delay) {
    return this.restTemplate.getForObject(
        String.format("http://name-service/name?delay=%d", delay),
        String.class);
}
```