

Déploiement de micro-services Spring avec Kubernetes

David THIBAU – 2020

david.thibau@gmail.com



Agenda

- **Rappels Spring Cloud / Kubernetes**
 - Architecture Micro-services
 - Services techniques : Framework ou infrastructure
 - L'offre Spring-Cloud
 - Kubernetes
 - Spring Cloud Kubernetes
- **Découverte de service**
 - DiscoveryClient
 - Service natif Kubernetes
 - Discovery Ribbon
- **Services Spring Cloud**
 - Actuator
 - Résilience avec Hystrix
- **Istio**
 - Présentation Istio
 - Istio et Spring Cloud
- **Outillage**
 - Profils SpringBoot
 - Plugins Maven
 - Skaffold
- **Configuration centralisée**
 - ConfigMap
 - Secrets
 - Rechargements dynamiques



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* 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...)

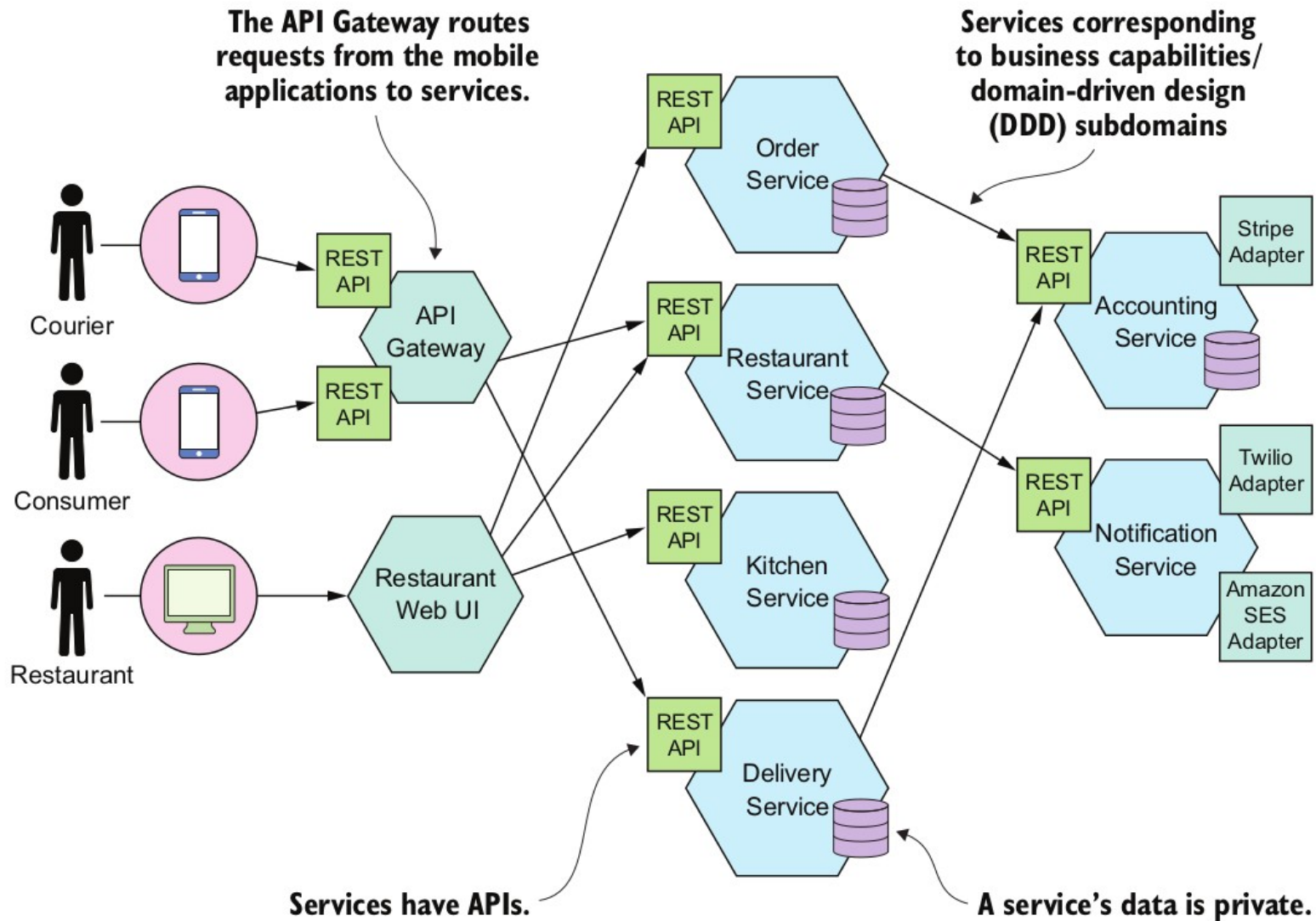
Découverte automatique : Les services sont typiquement distribués dans l'environnement d'exécution, le scaling peut être automatisé. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement

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

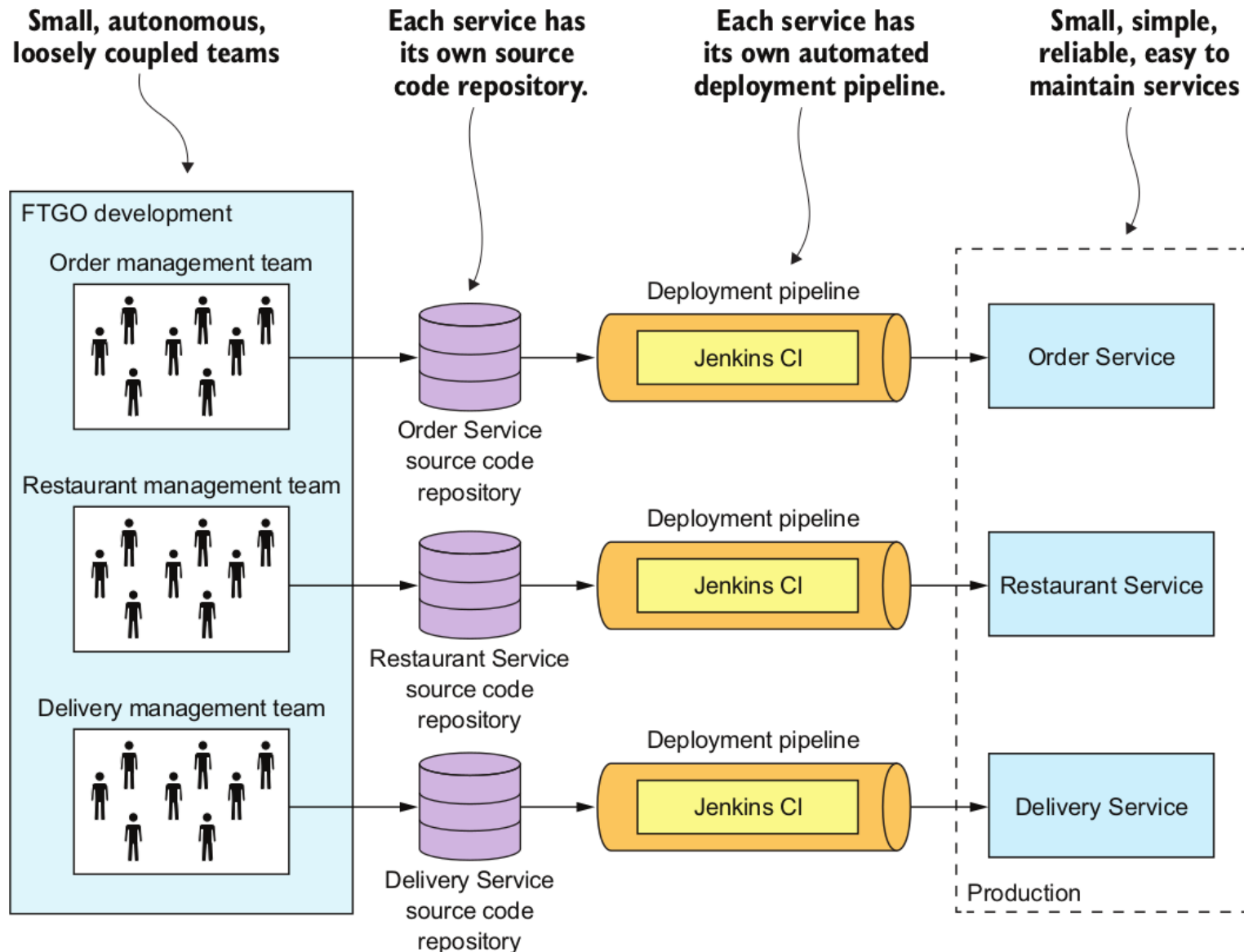
Résilience : Plus de services peuvent être en erreur. L'application doit pouvoir résister aux erreurs.

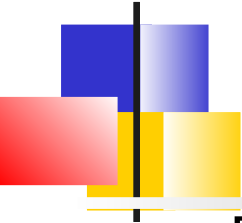
DevOps : L'intégration et le déploiement continu sont indispensables pour le succès.

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 ?

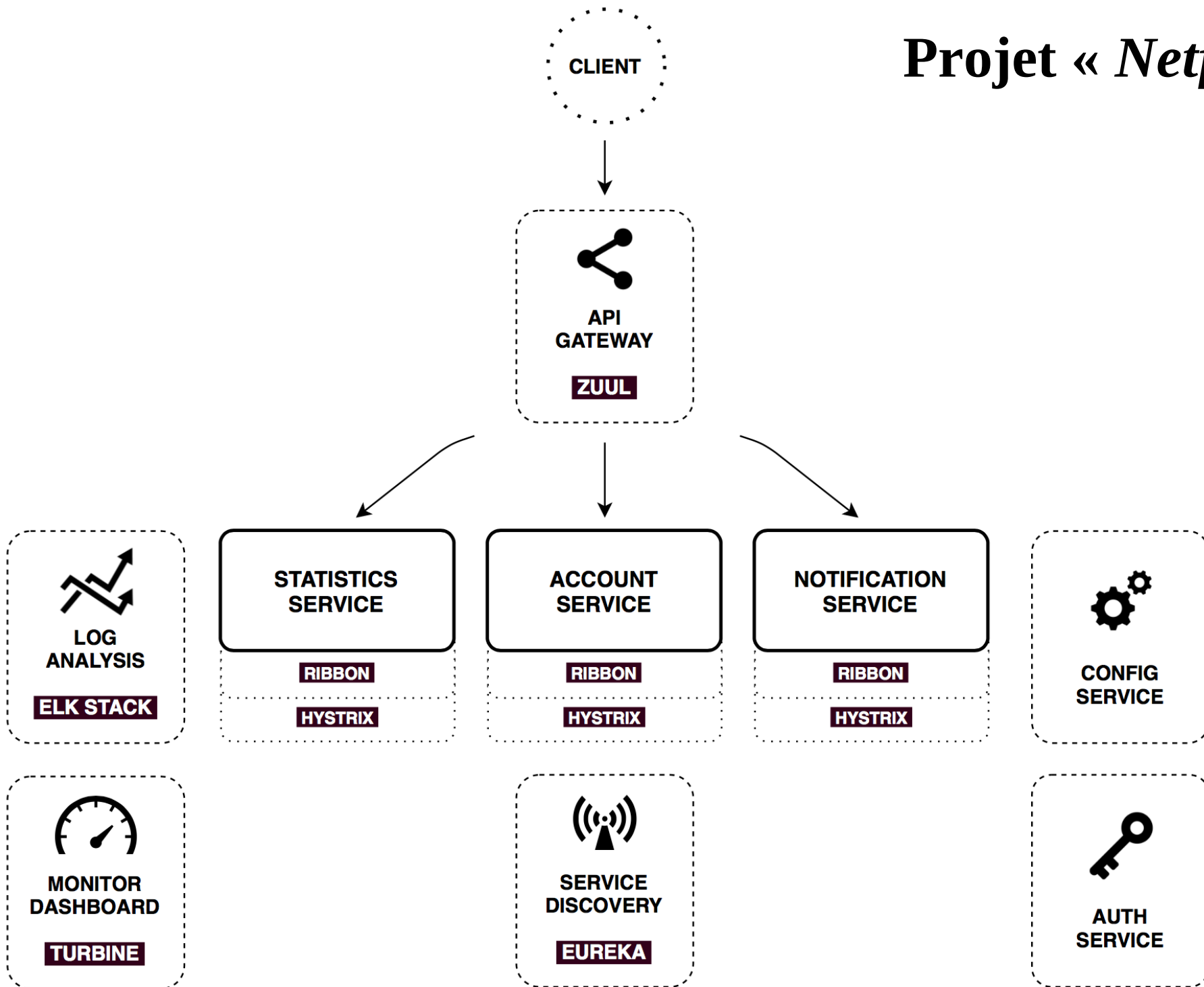


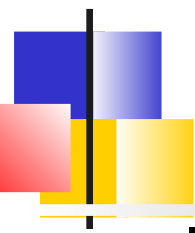
Services techniques

Les architectures micro-services nécessitent des services technique :

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

Projet « Netflix »





Infrastructure de déploiement

Infrastructure de déploiement pour ce type d'architecture :

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



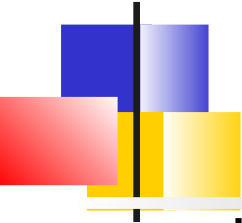
Services techniques

Qui fournit les services techniques ?

- Dans les premières architectures, c'est le software => framework Netflix
- Actuellement, de nombreux services techniques migrent vers l'infrastructure :
 - Services, Config, Répartition de charge
Kubernetes
 - Réilience, 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



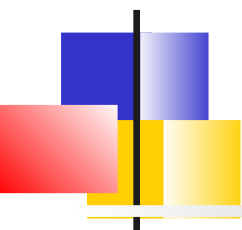
Kubernetes



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. »

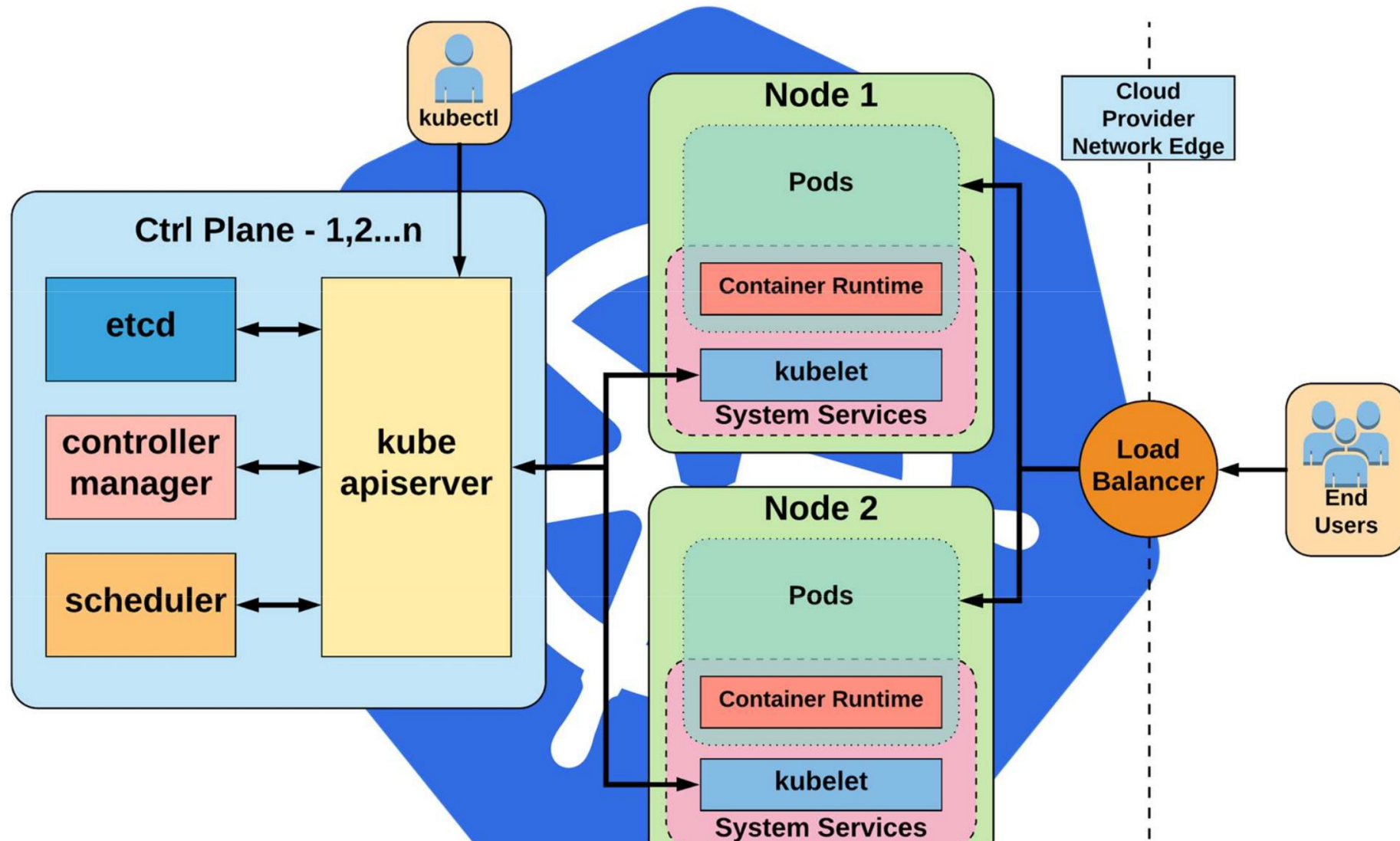


Fonctionnalités applicatives

- 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

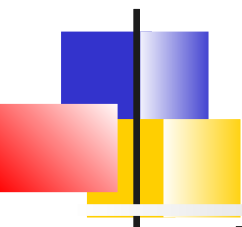


Principes

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

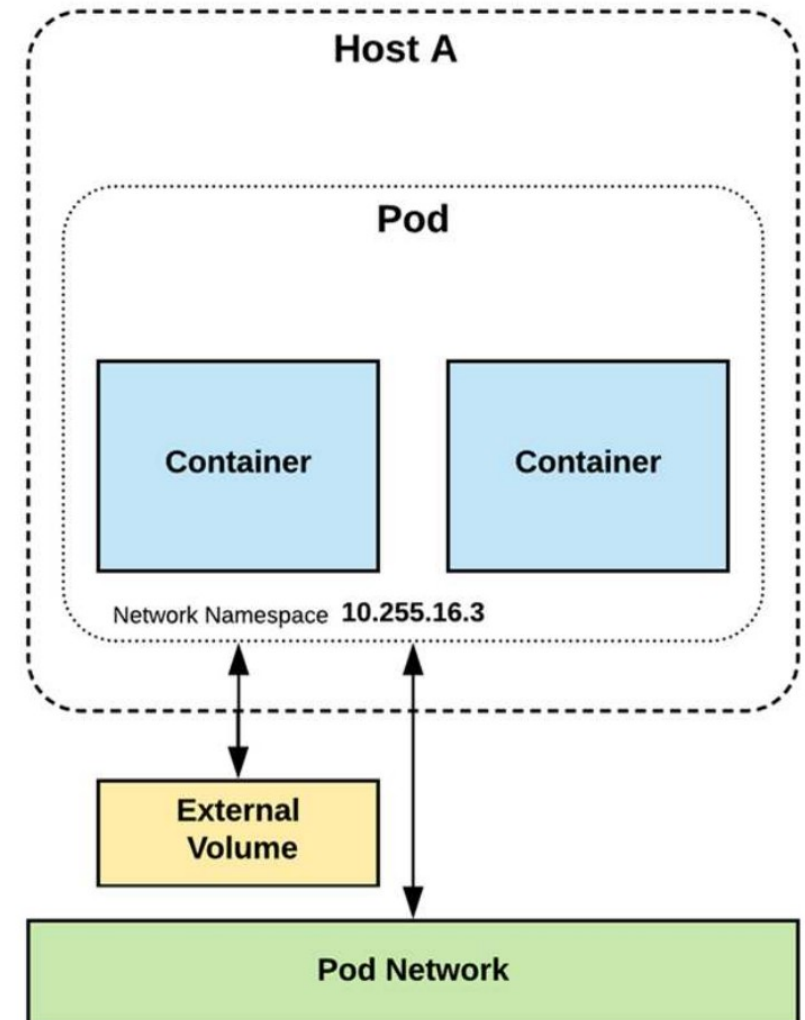
Un **pod** est la plus petite unité de travail

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





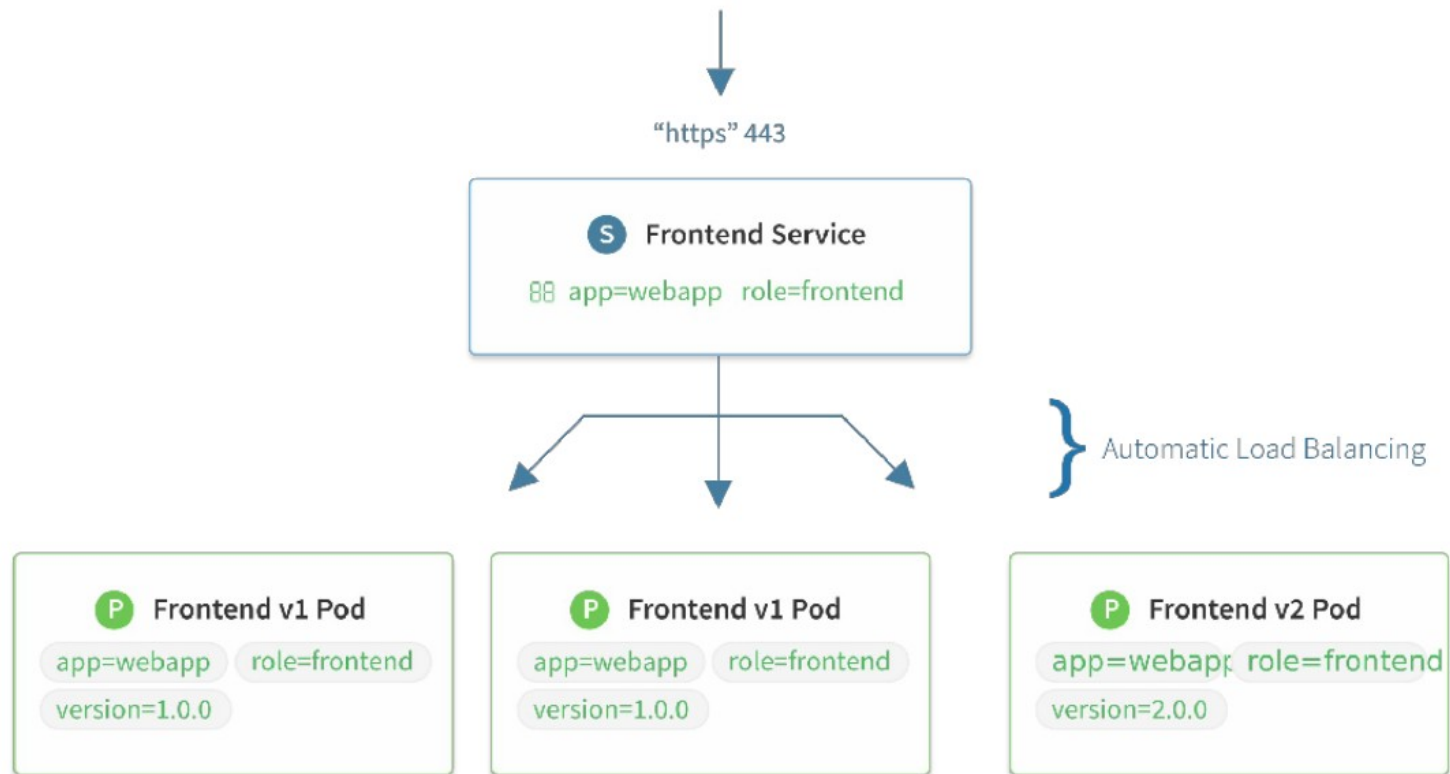
Services

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



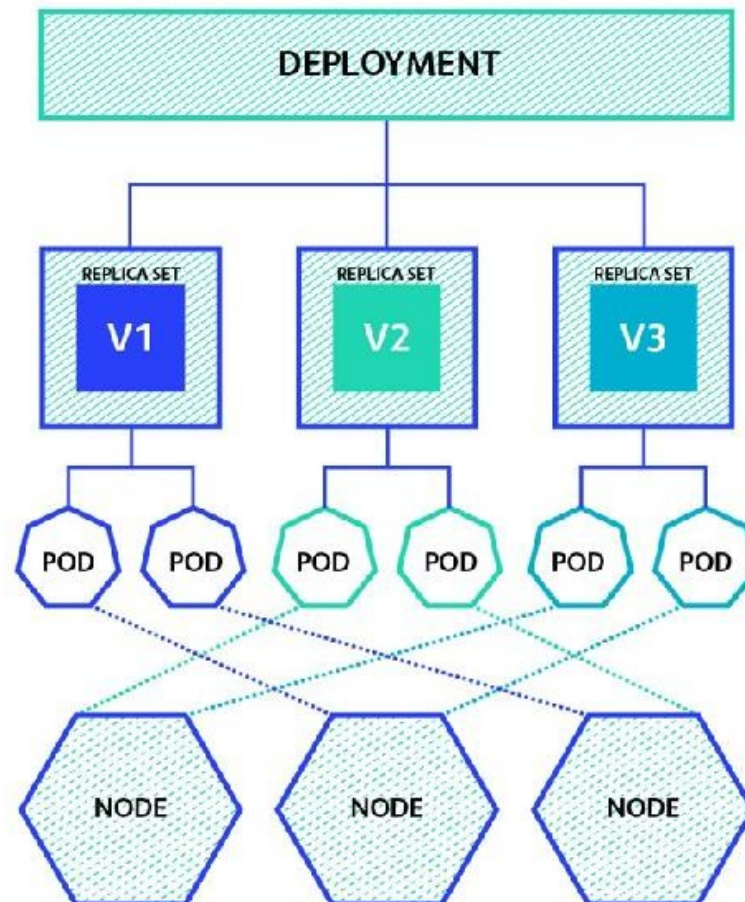
Déploiement

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 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.

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

Les sélecteurs permettent de rechercher des objets ayant des labels spécifiques

2 types de sélecteurs: égalité ou ensemble.

Ils sont utilisés par les opérations LIST et WATCH de l'API

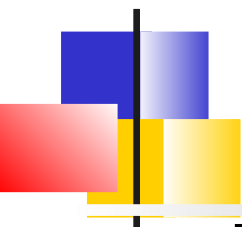
Les services et les ReplicaSet utilisent les labels pour sélectionner les pods



Annotations

Les annotations permettent d'attacher des métadonnées arbitraires non identifiables à des objets.

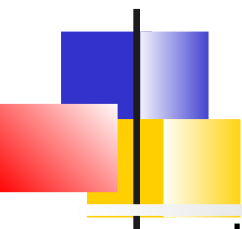
- Les clients tels que les outils et les bibliothèques peuvent récupérer ces métadonnées.



Écosystème *Kubernetes*

De nombreux outils peuvent être ajoutés à une installation coeur 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), ajoute un proxy sur chaque pod qui sécurise, monitore, 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

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



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 ***ConfigMaps***
- Équilibrage de charge côté client via Netflix Ribbon



Mise en place

spring-cloud-starter-kubernetes-all

Par défaut, Spring Cloud Kubernetes active le ***profil kubernetes*** lorsqu'il détectera qu'il est exécuté dans un cluster Kubernetes.

=> ***application-kubernetes.properties***
pour toutes les propriétés de configuration
spécifique kubernetes



Détails des starters

spring-cloud-starter-kubernetes :

Discovery client vers les services natifs de Kubernetes

spring-cloud-starter-kubernetes-config :

Chargement de la configuration via les configMaps, les secrets. Rechargement

spring-cloud-starter-kubernetes-ribbon :

Répartition côté client à partir de listes de serveur fournis par Kubernetes



Exemple Client Ribbon

```
@SpringBootApplication
@EnableDiscoveryClient // Service de registry de Kubernetes
@EnableCircuitBreaker
@EnableRibbonClient(name = "name-service", configuration = RibbonConfiguration.class)
public class GreetingServiceApplication {

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

    @LoadBalanced
    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```



Appel Rest

```
@Service
public class NameService {

    private final RestTemplate restTemplate;

    public NameService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

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

    private String getFallbackName(int delay) {
        return "Fallback";
    }

}
```



Outillage

Profil Spring Boot
Plugins Maven
Scaffold



Fabric8 Client

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 module, il n'est pas toujours nécessaire de le déployer car le code dépend de **Fabric8 Kubernetes Java client** qui lui effectue des requêtes http vers un cluster Kubernetes



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

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



Sécurité Kubernetes

Pour les distributions de Kubernetes qui prennent en charge un accès basé sur les rôles, on doit assurer qu'un pod qui s'exécute avec spring-cloud-kubernetes ait accès à l'API Kubernetes.

- Les comptes de service attribués à un déploiement doivent avoir le rôle approprié



Permissions nécessaires

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

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



Outillage

Création d'images
Plugins Maven
Skaffold



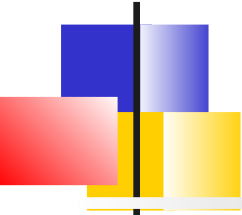
fabric8

Spring Cloud Kubernetes utilise et particulier le client Java du projet fabric8¹ qui fournit un accès aux API via un dsl

Le projet fournit également :

- La possibilité de mocker un cluster Kubernetes

1. Le projet fabric8 (RedHat) avec comme objectif un IDE pour les micro-services est déprécié.



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 : Permet de faire des requêtes CRUD. Les ressources sont alors stockées en mémoire



Plugin Maven

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>1.0.0-alpha-3</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:resources**

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

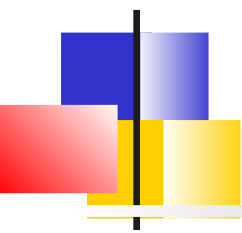


Exemple : Configuration XML

```
<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

Profils Spring Boot
fabric8-client et Plugins Maven
Skaffold



Skaffold

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

Il s'appuie sur :

- un fichier de configuration *skaffold.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

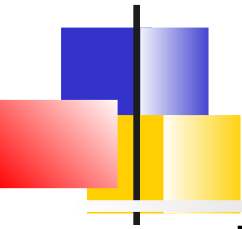
- crée une image Docker applicative
- la déploie sur Minikube.
- Surveille toute modification du code source et déclenche une nouvelle version après chaque modification du système de fichiers



Options

La commande prend différents paramètres :

- `--port-forward` : Exécute `kubectl 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



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 or Gradle, ou simplement en tant que bibliothèque Java

Il s'intègre 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

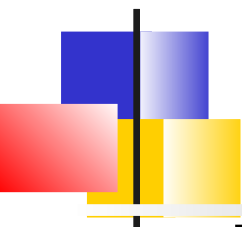
- à la propriété de bootstrap
spring.application.name
- Ou à la propriété spécifique :
spring.cloud.kubernetes.config.name

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

Secrets

Rechargement dynamique



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'opération sensible
=> 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



Configuration

ConfigMap
Secrets

Rechargement dynamique



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



Découverte de service

DiscoveryClient

Service natif Kubernetes

Ribbon Discovery



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
Ribbon Discovery



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
Ribbon Discovery



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'



Services Spring Cloud avec Kubernetes

Actuator et pod health
Circuit breaker et Hystrix



Indicateurs de santé des pods

Spring Boot utilise *HealthIndicator* pour exposer des informations sur la santé d'une application.

L'indicateur de santé Kubernetes (qui fait partie du module principal) 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



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);
}
```



Istio

Concepts Istio
Istio et Spring Cloud



Introduction

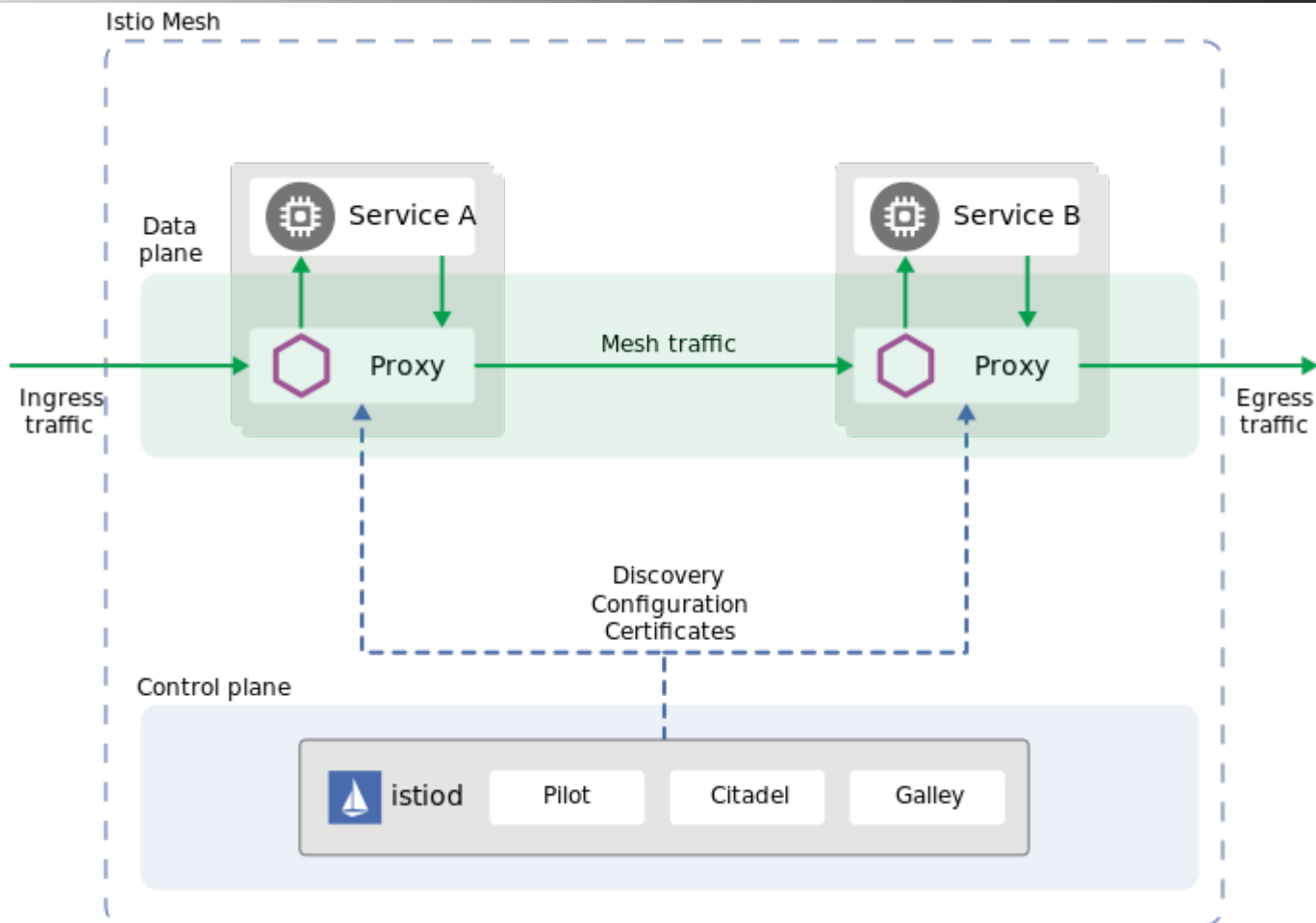
Istio est un ***service mesh*** qui permet de gérer la communication entre les micro-services déployés sous Kubernetes

- 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 :

- 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

Connect : Contrôler le flux d'appels entre service, exécuter des tests, déploiement red/black

Secure : Authentification, Autorisations et chiffrement des communications entre services

Control : Appliquer des stratégies garantissant la distribution de la charge

Observe : Tracing, Surveillance et logs



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)

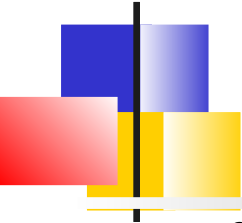
Service entries : Une entrée dans le registre de service de istio

Sidecars : Configuration du proxy envoy sidecar



Exemple Virtual Service

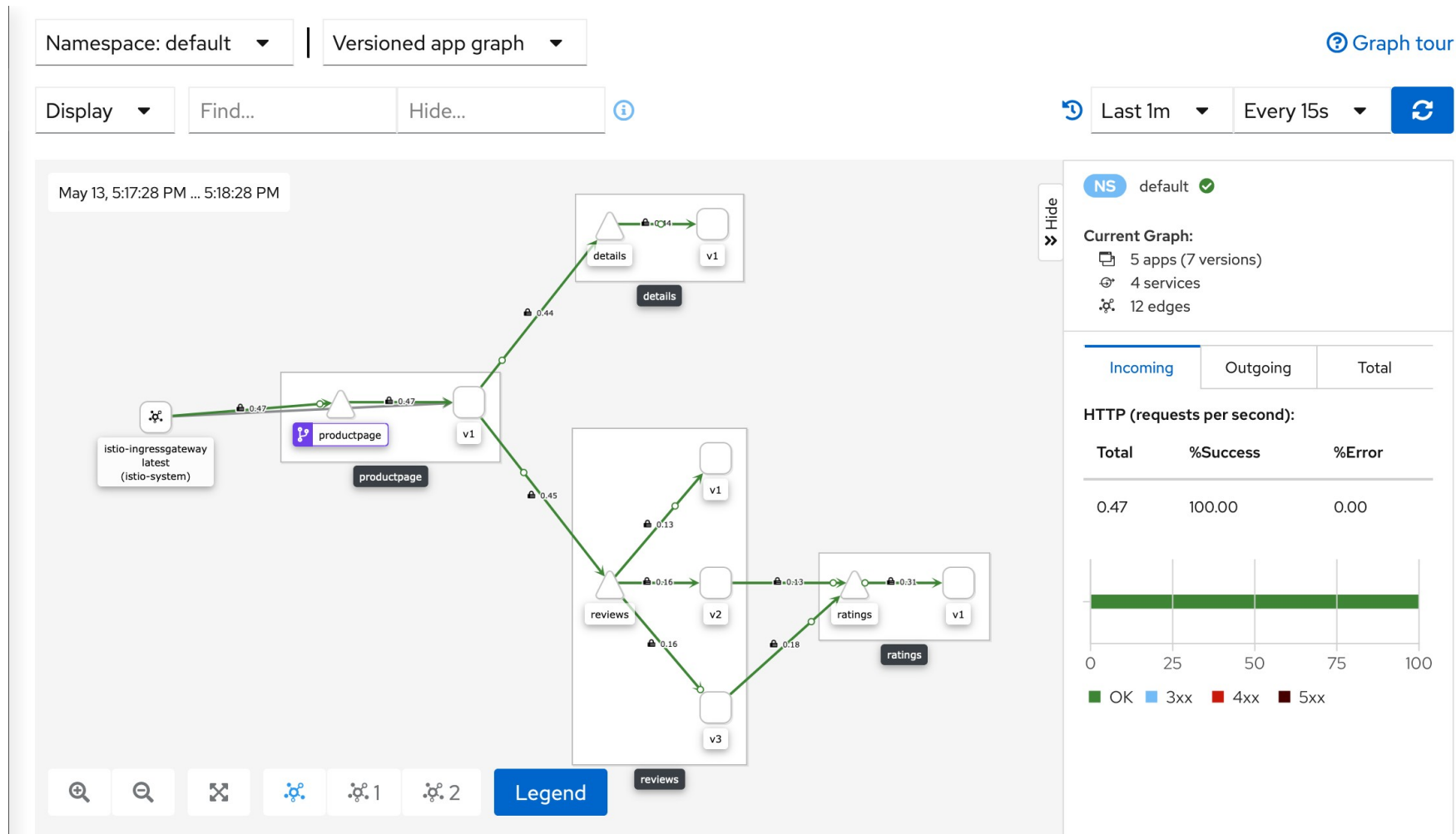
```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:                                     # Services, IP de destination
  - reviews
  http:                                     # Règles de routage
  - match:                                 # 1 règle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

Règles ~ zuul

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
    - bookinfo.com
  http:
    - match:
        - uri:
            prefix: /reviews
      route:
        - destination:
            host: reviews
    - match:
        - uri:
            prefix: /ratings
      route:
        - destination:
            host: ratings
```

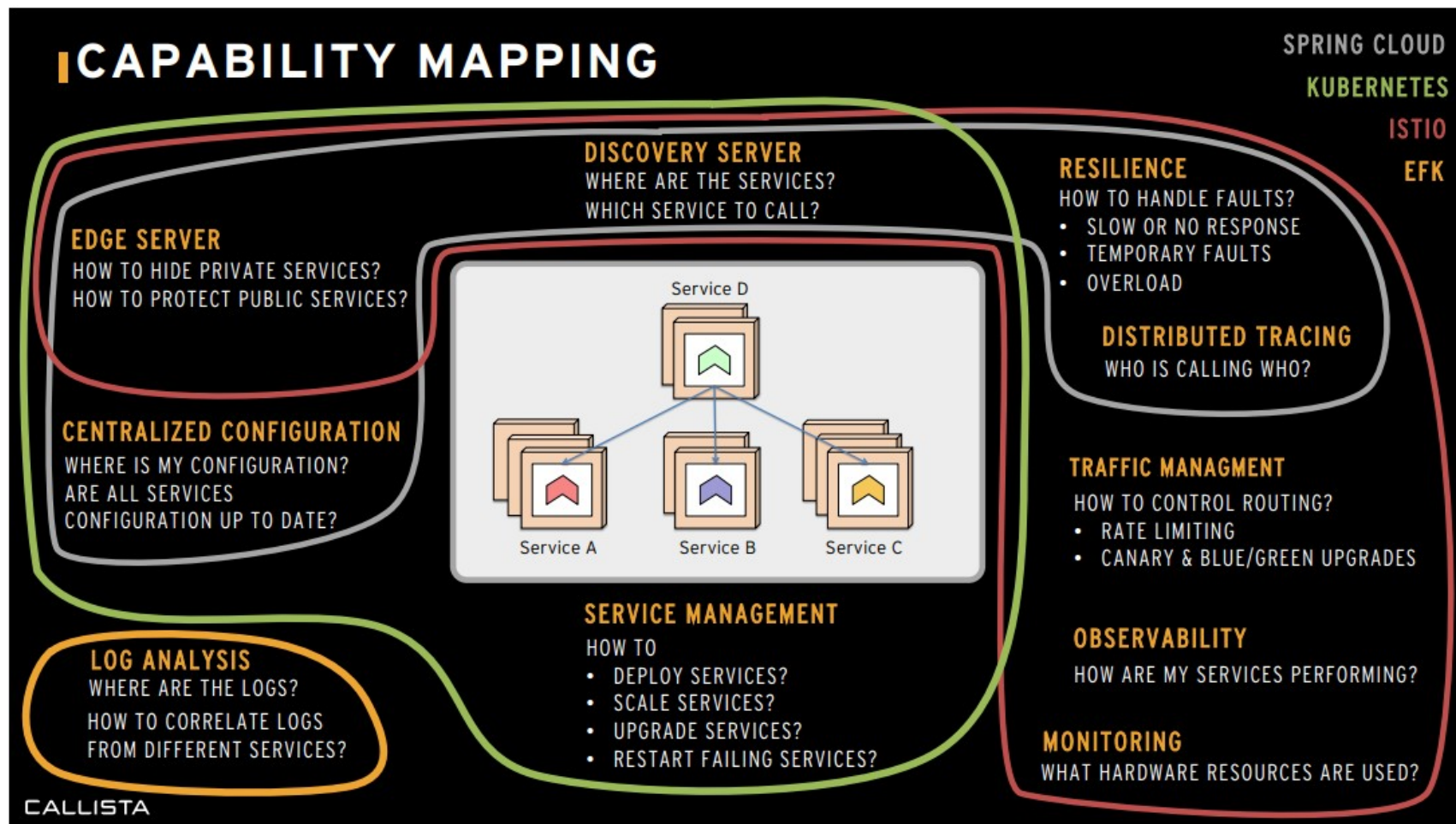
Tableau de bord Kiali





Istio et Spring Cloud

Service requis



Un exemple de choix

SPRING CLOUD + KUBERNETES + ISTIO

