FORMATION KUBERNETES

Mise en œuvre

Agenda

1. Introduction
2. Projet Kubernetes, communauté et gouvernance
3. Architecture
4. Concepts et Objets
5. Networking
6. Service Mesh
7. Stockage
8. Gestion de la configuration des applications
9. Gestion des ressources
10. Scheduling
11. Service Mesh
12. Utilisation et déploiement des ressources
13. Helm
14. Comment déployer ?
15. Sécurité et contrôle d'accès

OBJECTIFS PÉDAGOGIQUES

À l’issue de la formation vous serez en mesure de :

- Comprendre le positionnement de Kubernetes et la notion d'orchestration
- Installer Kubernetes et ses différents composants
- Utiliser les fichiers descriptifs YAML
- Définir les bonnes pratiques pour travailler avec Kubernetes

Introduction

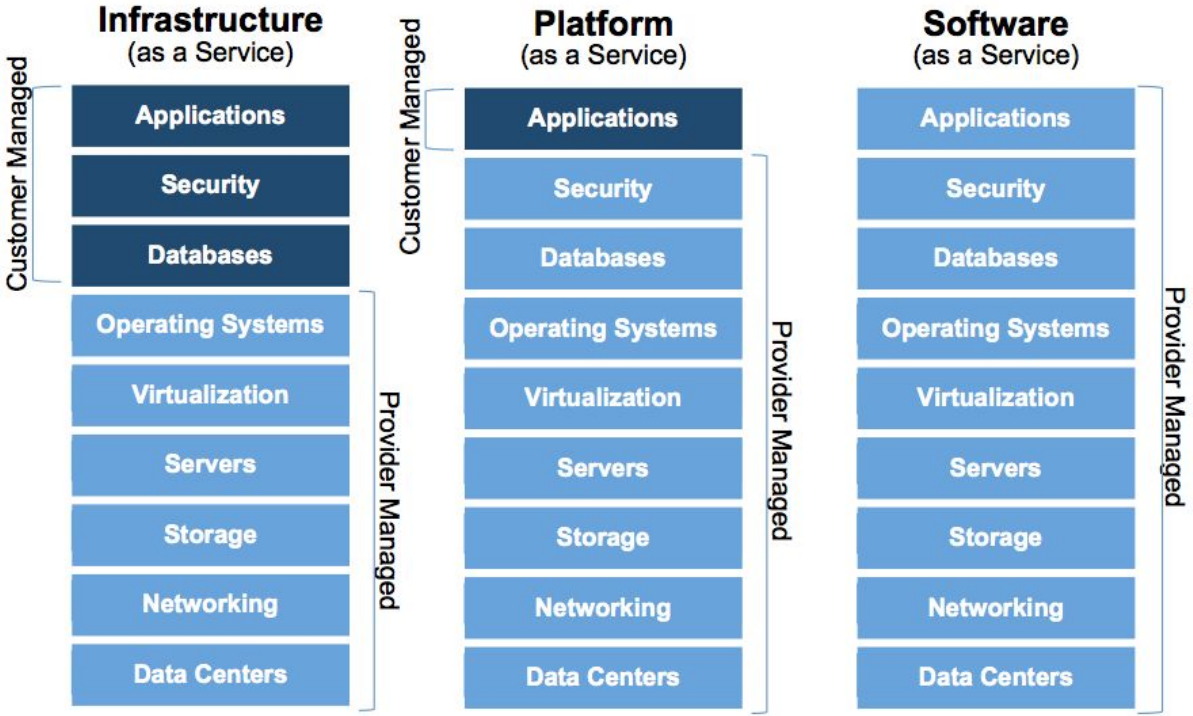
Vue d'ensembe du cloud

- Stockage/calcul distant (on oublie, cf. externalisation)
- Virtualisation++
- Abstraction du matériel (voire plus)
- Accès normalisé par des APIs
- Service et facturation à la demande
- Flexibilité, élasticité

WAAS : WHATEVER AS A SERVICE

- IaaS : Infrastructure as a Service
- PaaS : Platform as a Service
- SaaS : Software as a Service

WAAS : WHATEVER AS A SERVICE



POURQUOI DU CLOUD ?

- Abstraction des couches basses
- On peut tout programmer à son gré (API)
- Permet la mise en place d’architectures scalables

VIRTUALISATION DANS LE CLOUD

- Le cloud IaaS repose souvent sur la virtualisation
- Ressources compute : virtualisation
- Virtualisation complète : KVM, Xen
- Virtualisation conteneurs : OpenVZ, LXC, Docker, RKT

NOTIONS ET VOCABULAIRE IAAS

- L’instance est par définition éphémère
- Elle doit être utilisée comme ressource de calcul
- Séparer les données des instances

ORCHESTRATION DES RESSOURCES ?

- Groupement fonctionnel de ressources : micro services
- Infrastructure as Code : Définir toute une infrastructure dans un seul fichier texte de manière déclarative
- Scalabilité : passer à l'échelle son infrastructure en fonction de différentes métriques.

POSITIONNEMENT DES CONTENEURS DANS L'ÉCOSYSTÈME CLOUD ?

- Facilitent la mise en place de PaaS
- Fonctionnent sur du IaaS ou sur du bare-metal
- Simplifient la décomposition d'applications en micro services

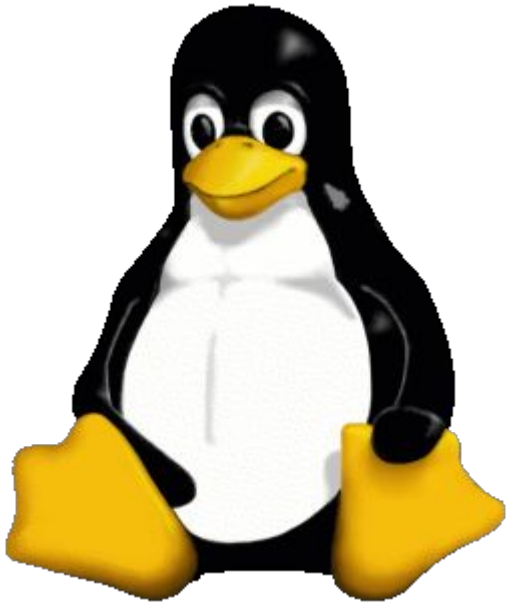
LES CONTENEURS

DÉFINITION:

Les conteneurs fournissent un environnement isolé sur un système hôte, semblable à un chroot sous Linux ou une jail sous BSD, mais en proposant plus de fonctionnalités en matière d'isolation et de configuration. Ces fonctionnalités sont dépendantes du système hôte et notamment du kernel.

LE KERNEL LINUX

- Namespaces
- Cgroups (control groups)



LES NAMESPACES

- MOUNT NAMESPACES (LINUX 2.4.19)

Permet de créer un arbre des points de montage indépendants de celui du système hôte.

- UTS NAMESPACES (LINUX 2.6.19)

Unix Time Sharing : Permet à un conteneur de disposer de son propre nom de domaine et d'identité NIS sur laquelle certains protocoles tel que LDAP peuvent se baser.

- IPC NAMESPACES (LINUX 2.6.19)

Inter Process Communication : Permet d'isoler les bus de communication entre les processus d'un conteneur.

LES NAMESPACES

- PID NAMESPACES (LINUX 2.6.24)

Isole l'arbre d'exécution des processus et permet donc à chaque conteneur de disposer de son propre processus maître (PID 0) qui pourra ensuite exécuter et manager d'autres processus avec des droits illimités tout en étant un processus restreint au sein du système hôte.

- USER NAMESPACES (LINUX 2.6.23-3.8)

Permet l'isolation des utilisateurs et des groupes au sein d'un conteneur. Cela permet notamment de gérer des utilisateurs tels que l'UID 0 et GID 0, le root qui aurait des permissions absolues au sein d'un namespace mais pas au sein du système hôte.

- NETWORK NAMESPACES (LINUX 2.6.29)

Permet l'isolation des ressources associées au réseau, chaque namespace dispose de ses propres cartes réseaux, plan IP, table de routage, etc.

CGROUPS : CONTROL CROUPS

```
CGroup: /
           |--docker
           |  |--7a977a50f48f2970b6ede780d687e72c0416d9ab6e0b02030698c1633fdde956
           |  |--6807 nginx: master process ngin
           |  |  |--6847 nginx: worker proces
```

- **Limitation des ressources :** des groupes peuvent être mis en place afin de ne pas dépasser une limite de mémoire.

- **Priorisation :** certains groupes peuvent obtenir une plus grande part de ressources processeur ou de bande passante d'entrée-sortie.

CGROUPS : CONTROL CROUPS

- **Comptabilité :** permet de mesurer la quantité de ressources consommées par certains systèmes, en vue de leur facturation par exemple.

- **Isolation :** séparation par espace de nommage pour les groupes, afin qu'ils ne puissent pas voir les processus des autres, leurs connexions réseaux ou leurs fichiers.

- **Contrôle :** figer les groupes ou créer un point de sauvegarde et redémarrer.

DEUX PHILOSOPHIES DE CONTENEURS

- Systeme : simule une séquence de boot complète avec un init process ainsi que plusieurs processus (LXC, OpenVZ).

- Process : un conteneur exécute un ou plusieurs processus directement, en fonction de l'application conteneurisée (Docker, Rkt).

ENCORE PLUS “CLOUD” QU’UNE INSTANCE

- Partage du kernel
- Un seul processus par conteneur
- Le conteneur est encore plus éphémère que l’instance
- Le turnover des conteneurs est élevé : orchestration

CONTAINER RUNTIME

- Permettent d'exécuter des conteneurs sur un système
- **docker**: historique
- **containerd**: implémentation de référence
- **cri-o**: implémentation Open Source développée par RedHat
- kata containers: Conteneurs dans des VMs

CONSTRUCTION D'UNE IMAGE

- Possibilité de construire son image à la main (long et source d'erreurs)

- Suivi de version et construction d'images de manière automatisée

- Utilisation de *Dockerfile* afin de garantir l'idempotence des images

DOCKERFILE

- Suite d'instruction qui définit une image

- Permet de vérifier le contenu d'une image

```
FROM alpine:3.4
MAINTAINER Ahmed Hosni <ahmedhosni.contact@gmail.com>
RUN apk -U add nginx
EXPOSE 80 443
CMD ["nginx"]
```

DOCKERFILE : PRINCIPALES INSTRUCTIONS

- **FROM :** baseimage utilisée
- **RUN :** Commandes effectuées lors du build de l'image
- **EXPOSE :** Ports exposées lors du run (si -P est précisé)
- **ENV :** Variables d'environnement du conteneur à l'instanciation
- **CMD :** Commande unique lancée par le conteneur
- **ENTRYPOINT :** "Préfixe" de la commande unique lancée par le conteneur

DOCKERFILE : BEST PRACTICES

- Bien choisir sa baseimage

- Chaque commande Dockerfile génère un nouveau layer

- Comptez vos layers !

DOCKERFILE : BAD LAYERING

```
RUN apk --update add \
    git \
    tzdata \
    python \
    unrar \
    zip \
    libxslt \
    py-pip \
RUN rm -rf /var/cache/apk/*
VOLUME /config /downloads
EXPOSE 8081
CMD ["--datadir=/config", "--nolaunch"]
ENTRYPOINT ["/usr/bin/env","python2","/sickrage/SickBeard.py"]
```

DOCKERFILE : GOOD LAYERING

```
RUN apk --update add \
    git \
    tzdata \
    python \
    unrar \
    zip \
    libxslt \
    py-pip \
    && rm -rf /var/cache/apk/*
VOLUME /config /downloads
EXPOSE 8081
CMD ["--datadir=/config", "--nolaunch"]
ENTRYPOINT ["/usr/bin/env","python2","/sickrage/SickBeard.py"]
```

DOCKERFILE : DOCKERHUB

- Build automatisée d'images Docker

- Intégration GitHub / DockerHub

- Plateforme de stockage et de distribution d'images Docker

SHIP : LES CONTENEURS SONT MANIPULABLES

- Sauvegarder un conteneur :

```
docker commit mon-conteneur backup/mon-conteneur
```

```
docker run -it backup/mon-conteneur
```

- Exporter une image :

```
docker save -o mon-image.tar backup/mon-conteneur
```

- Importer un conteneur :

```
docker import mon-image.tar backup/mon-conteneur
```

SHIP : DOCKER REGISTRY

- DockerHub n’est qu’au Docker registry ce que GitHub est à git

- Pull and Push

- Image officielle : registry

RUN : LANCER UN CONTENEUR

docker run

-d (detach)

-i (interactive)

-t (pseudo tty)

RUN : BEAUCOUP D'OPTIONS...

-v /directory/host:/directory/container

-p portHost:portContainer

-P

-e “VARIABLE=valeur”

--restart=always

--name=mon-conteneur

RUN : ...DONT CERTAINES UN PEU DANGEREUSES

--privileged (Accès à tous les devices)

--pid=host (Accès aux PID de l’host)

--net=host (Accès à la stack IP de l’host)

RUN : SE “CONNECTER” À UN CONTENEUR

docker exec

docker attach

RUN : DÉTRUIRE UN CONTENEUR

docker kill (SIGKILL)

docker stop (SIGTERM puis SIGKILL)

docker rm (détruit complètement)

KUBERNETES

Projet, communauté
et gouvernance

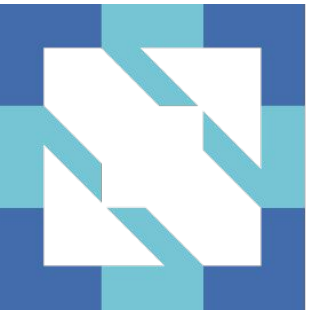
KUBERNETES

- COE développé par Google, devenu open source en 2014
- Adapté à tout type d'environnement
- Devenu populaire en très peu de temps
- Premier projet de la CNCF

kubernetes

CNCF

The Foundation’s mission is to create and drive the adoption of a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes.



CLOUD NATIVE
COMPUTING FOUNDATION

CNCF

- PRÉREQUIS:

- Distribuer sous forme de conteneurs
- Gestion dynamique de la configuration
- Orienté micro services

- LES RÔLES:

- Intendance des projets
- Faire grossir et évoluer l'écosystème
- Rendre la technologie accessible
- Promouvoir la technologie

OCI

- Créé sous la Linux Fondation
- But : Créer un standard Open Source concernant la manière de "runner" et le format des conteneurs et images
- Non lié à des produits
- Non lié à des COE
- runC a été donné par Docker à l'OCI comme implémention de base

KUBERNETES : PROJET

Docs : https://kubernetes.io/docs/

Slack : http://slack.k8s.io/

Discuss : https://discuss.kubernetes.io

Stack Overflow : https://stackoverflow.com/questions/tagged/kubernetes

KUBERNETES : PROJET

Hébergé sur Github : https://github.com/kubernetes/kubernetes

Issues : https://github.com/kubernetes/kubernetes/issues

Pull Requests https://github.com/kubernetes/kubernetes/pulls

Releases : https://github.com/kubernetes/kubernetes/releases

Projets en incubation : https://github.com/kubernetes-sigs/

KUBERNETES : CYCLE DE DÉVELOPPEMENT

- Chaque release a son propre planning, pour exemple : https://github.com/kubernetes/sig-release/tree/master/releases/release-1.23#timeline
- Chaque cycle de développement dure 12 semaines et peut être étendu si nécessaire
- Features freeze
- Code Freeze
- Alpha Release
- Beta Releases
- Release Candidates

KUBERNETES : COMMUNAUTÉ

Contributor and community guide :
https://github.com/kubernetes/community/blob/master/README.md#kubernetes-community

Décomposée en [Special Interest Groups] :
https://github.com/kubernetes/community/blob/master/sig-list.md

Les SIG sont des projets, centres d'intérêts ou Working Group différents :

- Network
- Docs
- AWS
- etc

Chaque SIG peut avoir des guidelines différentes.

KUBERNETES : KUBECON

La CNCF organise trois KubeCon par an :

- Amérique du Nord (San Diego, Seattle, Boston, etc)
- Europe (Berlin, Barcelone, Amsterdam, etc)
- Chine

KUBERNETES

Architecture

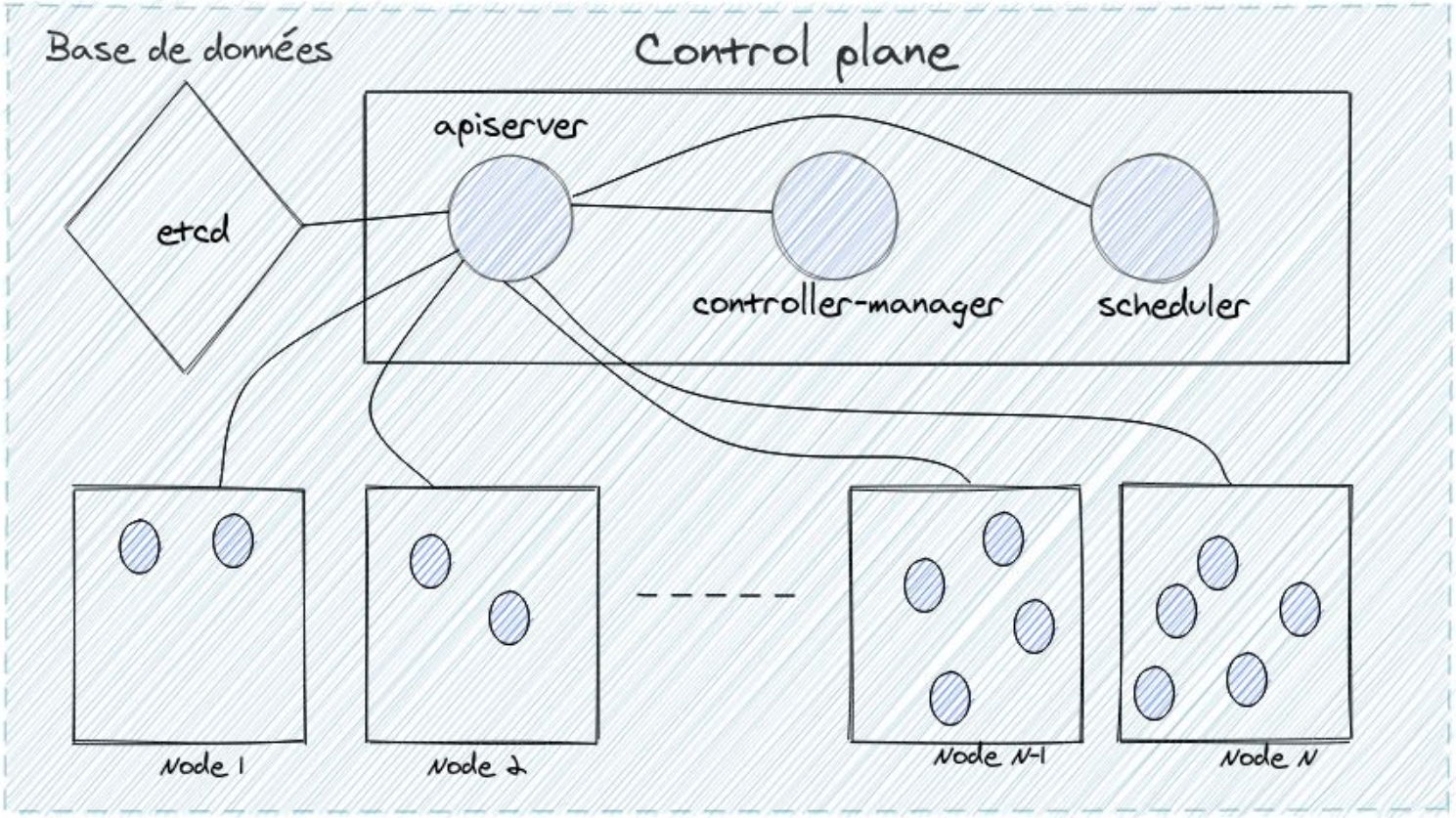
KUBERNETES : COMPOSANTS

- Kubernetes est écrit en Go, compilé statiquement.
- Un ensemble de binaires sans dépendance
- Faciles à conteneuriser et à packager
- Peut se déployer uniquement avec des conteneurs sans dépendance d'OS

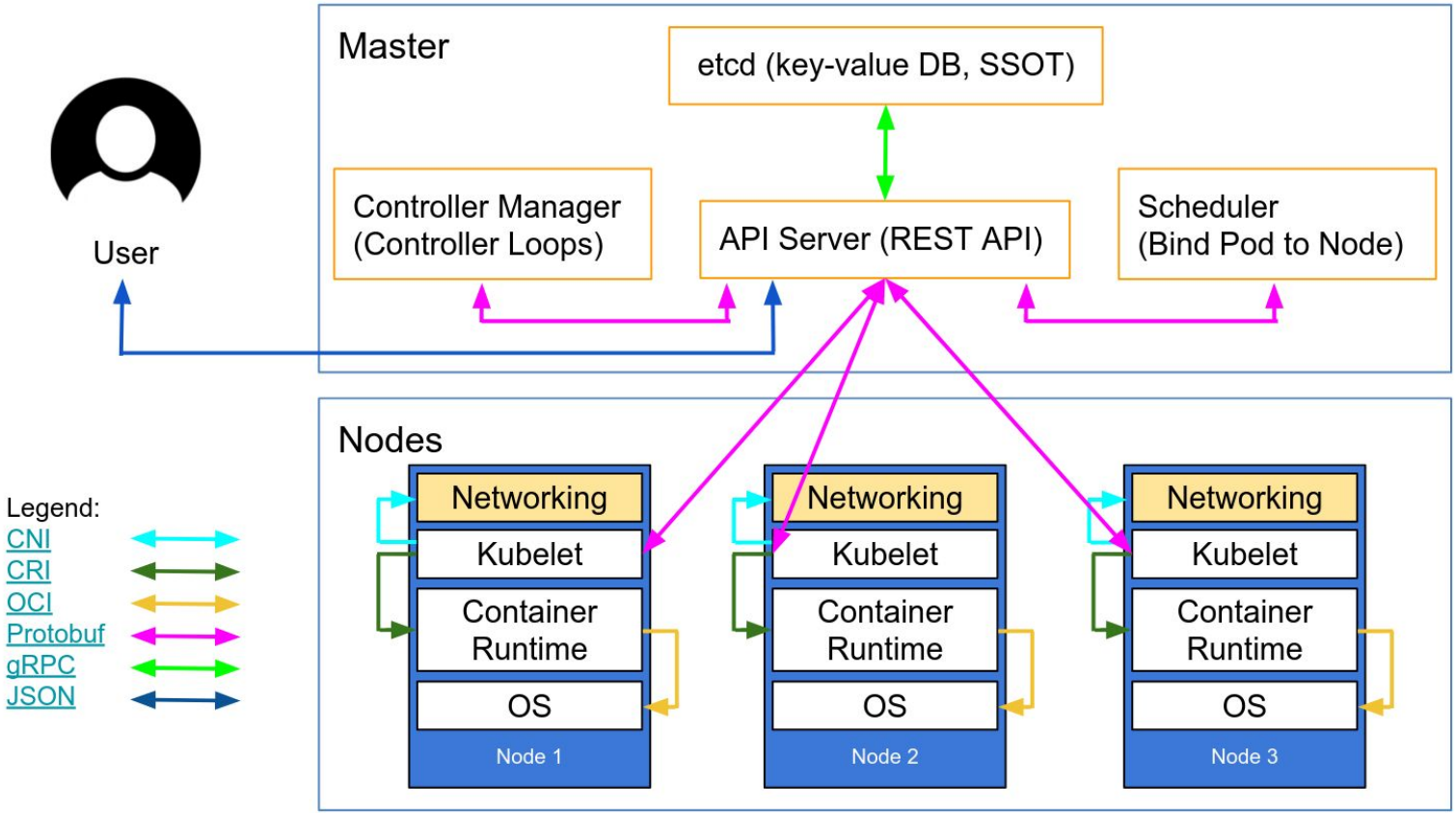
KUBERNETES : COMPOSANTS DU CONTROL PLANE

- etcd: Base de données
- kube-apiserver : API server qui permet la configuration d'objets Kubernetes (Pod, Service, Deployment, etc.)
- kube-proxy : Permet le forwarding TCP/UDP et le load balancing entre les services et les backends (Pods)
- kube-scheduler : Implémente les fonctionnalités de scheduling
- kube-controller-manager : Responsable de l'état du cluster, boucle infinie qui régule l'état du cluster afin d'atteindre un état désiré

KUBERNETES : COMPOSANTS DU CONTROL PLANE



KUBERNETES : COMPOSANTS DU CONTROL PLANE



KUBERNETES : ETCD

- Base de données de type Clé/Valeur (Key Value Store)
- Stocke l'état d'un cluster Kubernetes
- Point sensible (stateful) d'un cluster Kubernetes
- Projet intégré à la CNCF

KUBERNETES : KUBE-APISERVER

- Les configurations d'objets (Pods, Service, RC, etc.) se font via l'API server
- Un point d'accès à l'état du cluster aux autres composants via une API REST
- Tous les composants sont reliés à l'API server

KUBERNETES : KUBE-SCHEDULER

- Planifie les ressources sur le cluster
- En fonction de règles implicites (CPU, RAM, stockage disponible, etc.)
- En fonction de règles explicites (règles d'affinité et anti-affinité, labels, etc.)

KUBERNETES : KUBE-PROXY

- Responsable de la publication des Services
- Utilise iptables
- Route les paquets à destination des conteneurs et réalise le load balancing TCP/UDP

KUBERNETES : KUBE-CONTROLLER-MANAGER

- Boucle infinie qui contrôle l'état d'un cluster
- Effectue des opérations pour atteindre un état donné
- De base dans Kubernetes : replication controller, endpoints controller, namespace controller et serviceaccounts controller

KUBERNETES : AUTRES COMPOSANTS

- kubelet : Service "agent" fonctionnant sur tous les nœuds et assure le fonctionnement des autres services

- kubectl : Ligne de commande permettant de piloter un cluster Kubernetes

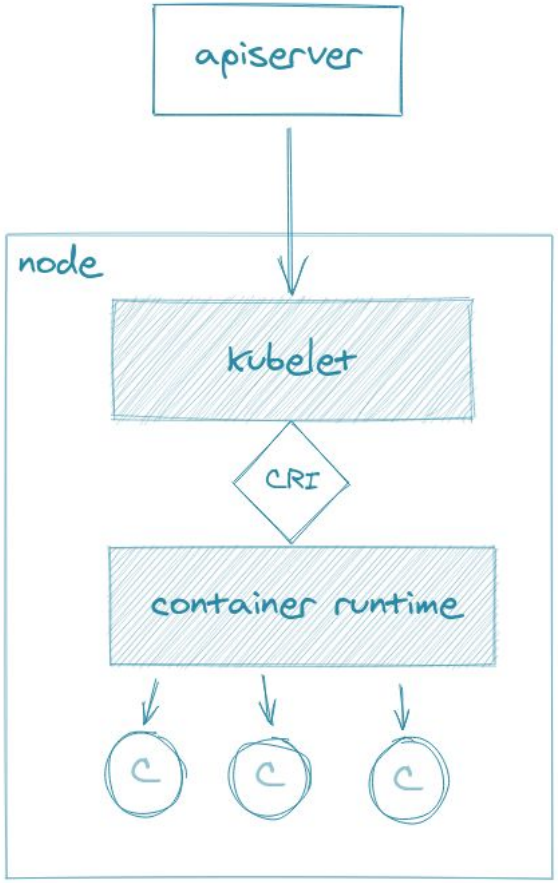
KUBERNETES : KUBELET

- Service principal de Kubernetes
- Permet à Kubernetes de s'auto configurer :
  - Surveille un dossier contenant les manifests (fichiers YAML des différents composant de Kubernetes).
  - Applique les modifications si besoin (upgrade, rollback).
- Surveille l'état des services du cluster via l'API server (kube-apiserver).

KUBERNETES : KUBELET

- Assure la communication entre les nodes et l'apiserver
- En charge de créer les conteneurs au travers de l'interface Container Runtime Interface (CRI)
- Peut fonctionner avec différentes container runtimes

KUBERNETES : KUBELET



KUBERNETES: NETWORK

Kubernetes n'implémente pas de solution réseau par défaut, mais s'appuie sur des solutions tierces qui implémentent les fonctionnalités suivantes :

- Chaque pods reçoit sa propre adresse IP
- Les pods peuvent communiquer directement sans NAT

KUBERNETES

Concepts et Objets

KUBERNETES : API RESOURCES

- Namespaces
- Pods
- Deployments
- DaemonSets
- StatefulSets
- Jobs
- Cronjobs

KUBERNETES : NAMESPACES

- Fournissent une séparation logique des ressources :
    - Par utilisateurs
    - Par projet / applications
    - Autres...
- Les objets existent uniquement au sein d'un namespace donné
- Évitent la collision de nom d'objets

KUBERNETES : LABELS

- Système de clé/valeur
- Organisent les différents objets de Kubernetes (Pods, RC, Services, etc.) d'une manière cohérente qui reflète la structure de l'application
- Corrèlent des éléments de Kubernetes : par exemple un service vers des Pods

KUBERNETES : LABELS

Exemple de label :

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

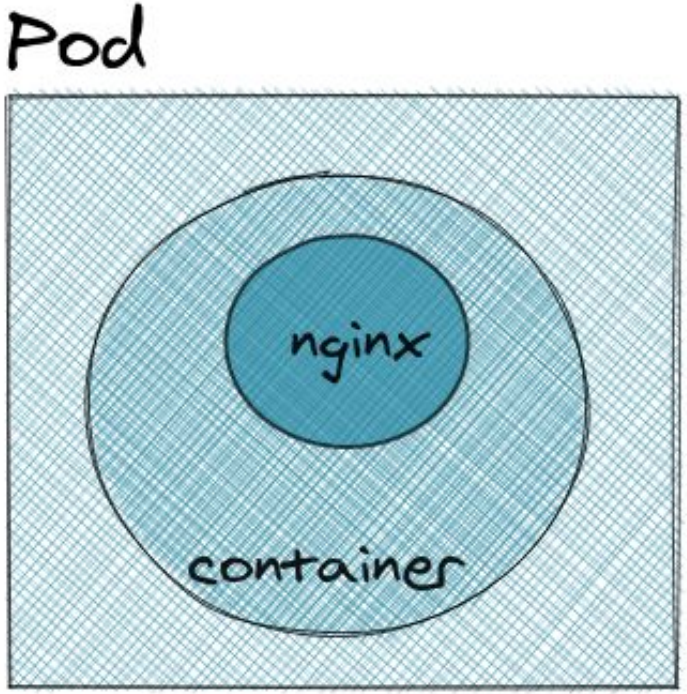
KUBERNETES : POD

- Ensemble logique composé de un ou plusieurs conteneurs
- Les conteneurs d'un pod fonctionnent ensemble (instanciation et destruction) et sont orchestrés sur un même hôte
- Les conteneurs partagent certaines spécifications du Pod :
  - La stack IP (network namespace)
  - Inter-process communication (PID namespace)
  - Volumes
- C'est la plus petite et la plus simple unité dans Kubernetes

KUBERNETES : POD

Les Pods sont définis en YAML comme les fichiers docker-compose :

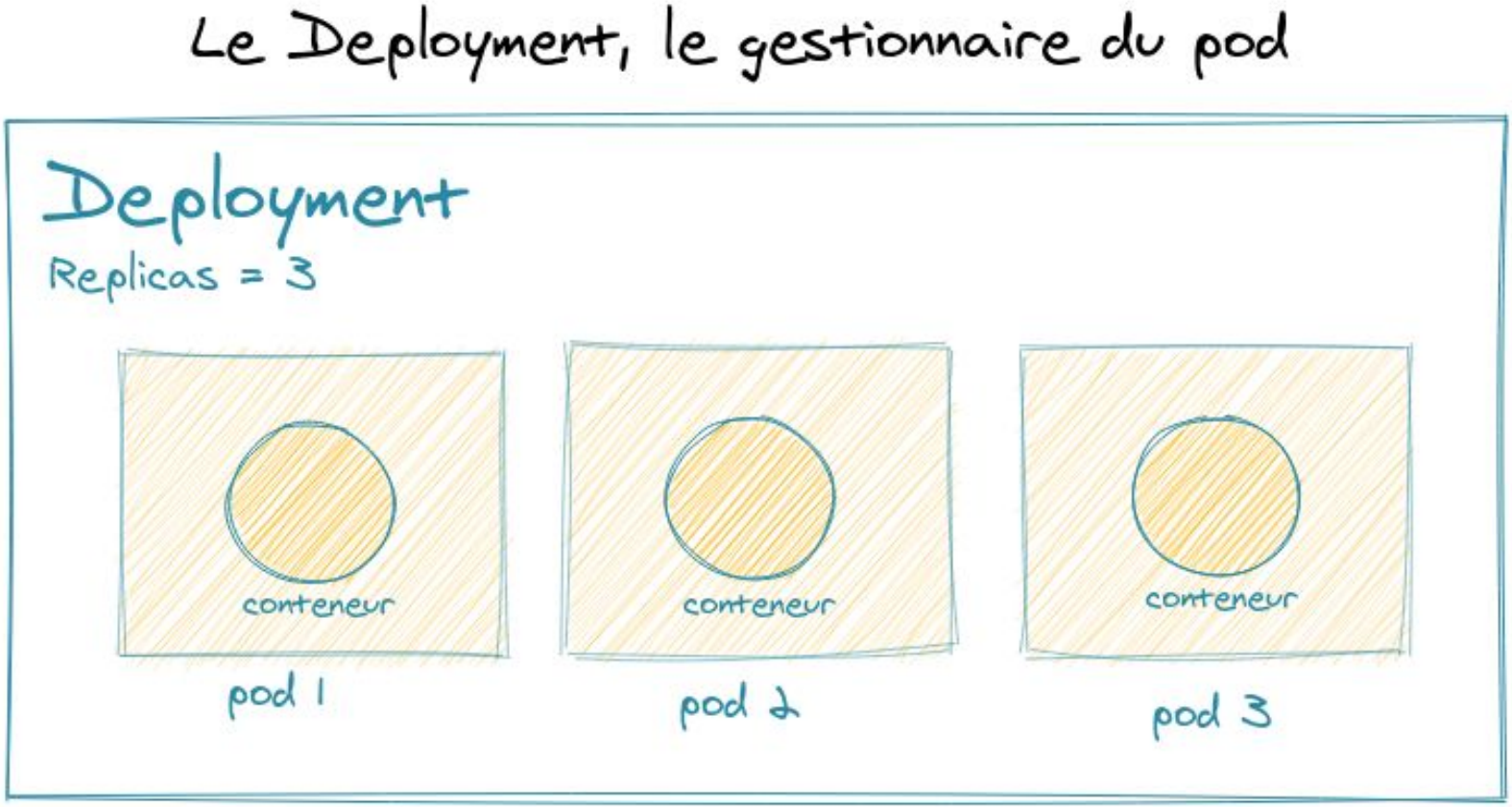
```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: mon_pod
spec:
  containers:
    - name: conteneur
      image: nginx:latest
```



KUBERNETES : DEPLOYMENT

- Permet d'assurer le fonctionnement d'un ensemble de Pods
- Version, Update et Rollback
- Anciennement appelés Replication Controllers

KUBERNETES : DEPLOYMENT



KUBERNETES : DEPLOYMENT

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

KUBERNETES : DAEMONSET

- Assure que tous les noeuds exécutent une copie du pod
- Ne connaît pas la notion de replicas.
- Utilisé pour des besoins particuliers comme :
  - L'exécution d'agents de collection de logs comme fluentd ou logstash
  - L'exécution de pilotes pour du matériel comme nvidia-plugin
  - L'exécution d'agents de supervision comme NewRelic agent ou Prometheus node exporter

KUBERNETES : DAEMONSET

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

KUBERNETES : STATEFULSET

- Similaire au Deployment
- Les pods possèdent des identifiants uniques.
- Chaque replica de pod est créé par ordre d'index
- Nécessite un Persistent Volume et un Storage Class.
- Supprimer un StatefulSet ne supprime pas le PV associé

KUBERNETES : STATEFULSET

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

KUBERNETES : JOB

- Crée des pods et s'assurent qu'un certain nombre d'entre eux se terminent avec succès.
- Peut exécuter plusieurs pods en parallèle
- Si un noeud du cluster est en panne, les pods sont reschedulés vers un autre noeud.

KUBERNETES : JOB

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1
  completions: 1
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure
```

KUBERNETES: CRON JOB

- Un CronJob permet de lancer des Jobs de manière planifiée.
- La programmation des Jobs se définit au format Cron
- Le champ jobTemplate contient la définition de l'application à lancer comme Job.

KUBERNETES : CRONJOB

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
          -  name: pi
             image: perl
             command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

KUBERNETES

Networking

KUBERNETES : NETWORK PLUGINS

- Kubernetes n'implémente pas de solution de gestion de réseau par défaut.

- Le réseau est implémenté par des solutions tierces :

- Calico : IPinIP + BGP
- Cilium : eBPF
- Weave : VXLAN
- Bien d'autres

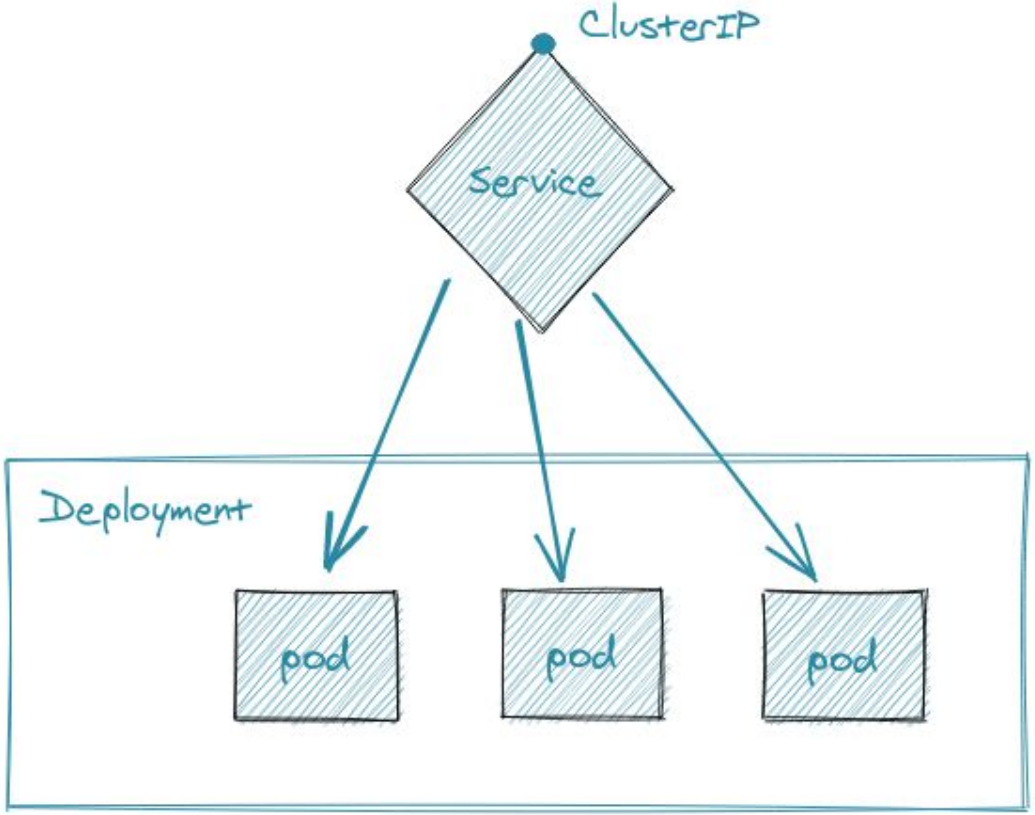
KUBERNETES : CNI

- Container Network Interface
- Projet dans la CNCF
- Standard pour la gestion du réseau en environnement conteneurisé
- Les solutions précédentes s'appuient sur CNI

KUBERNETES : SERVICES

- Abstraction des Pods sous forme d'une IP virtuelle de Service
- Rendre un ensemble de Pods accessibles depuis l'extérieur ou l'intérieur du cluster
- Load Balancing entre les Pods d'un même Service
- Sélection des Pods faisant parti d'un Service grâce aux labels

KUBERNETES : SERVICES



KUBERNETES : SERVICES CLUSTERIP

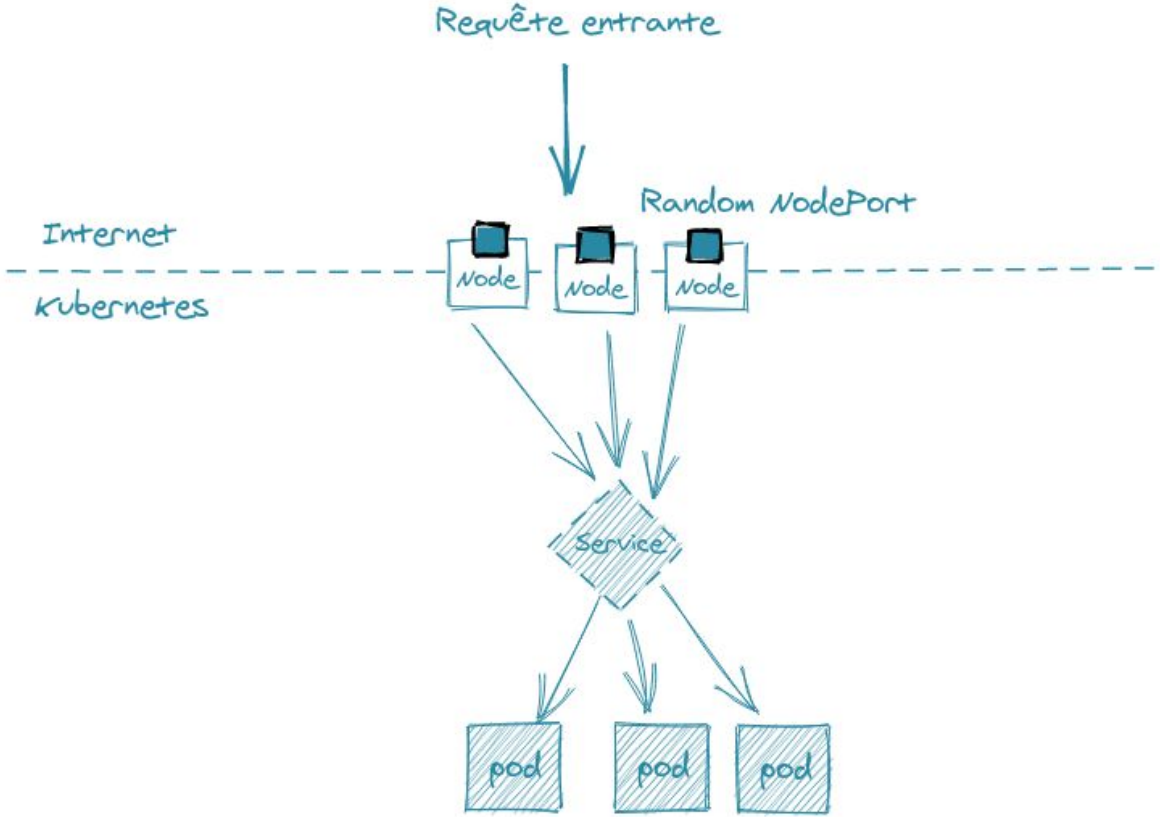
Exemple de Service (on remarque la sélection sur le label et le mode d'exposition):

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    app: guestbook
```

KUBERNETES : SERVICE NODEPORT

NodePort :

Chaque noeud du cluster ouvre un port statique et redirige le trafic vers le port indiqué

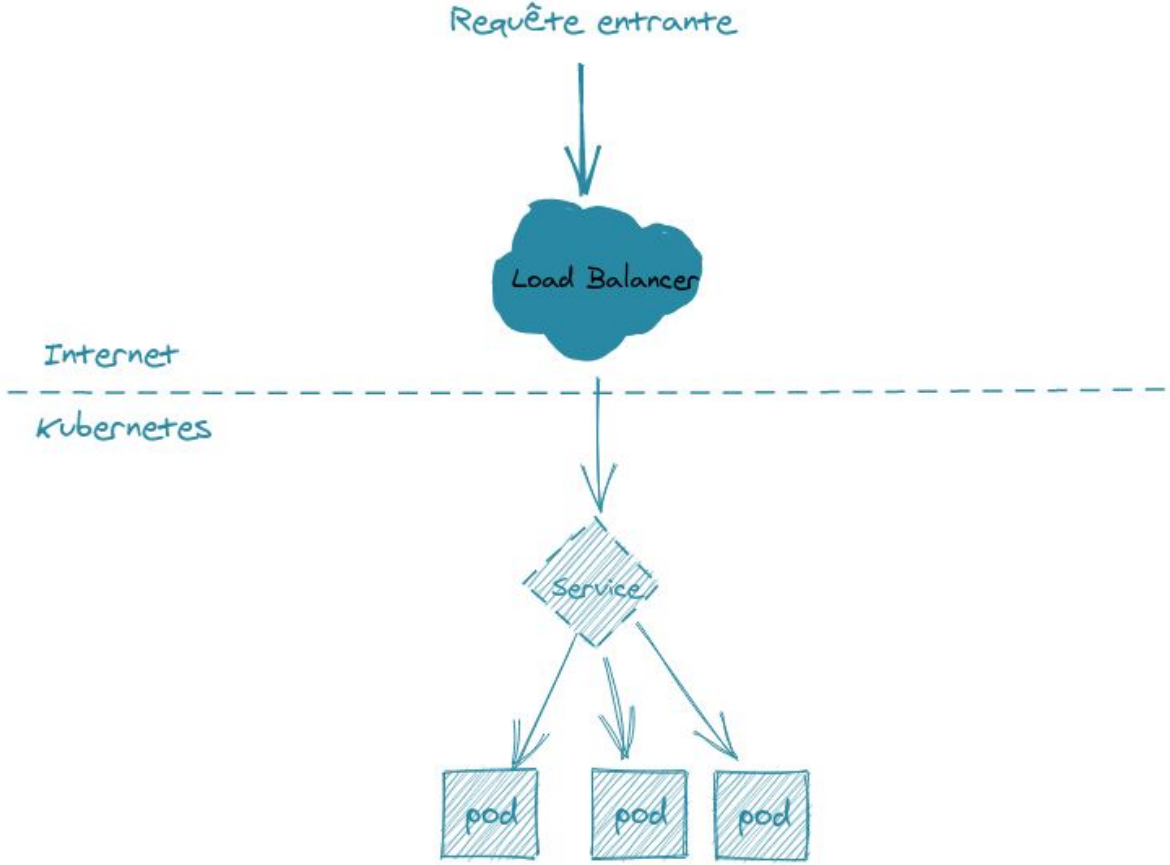


KUBERNETES : SERVICE LOADBALANCER

LoadBalancer : expose le Service en externe en utilisant le loadbalancer d'un cloud provider

- AWS ELB/ALB/NLB
- GCP LoadBalancer
- Azure Balancer
- OpenStack Octavia

KUBERNETES : SERVICE LOADBALANCER



KUBERNETES : SERVICES

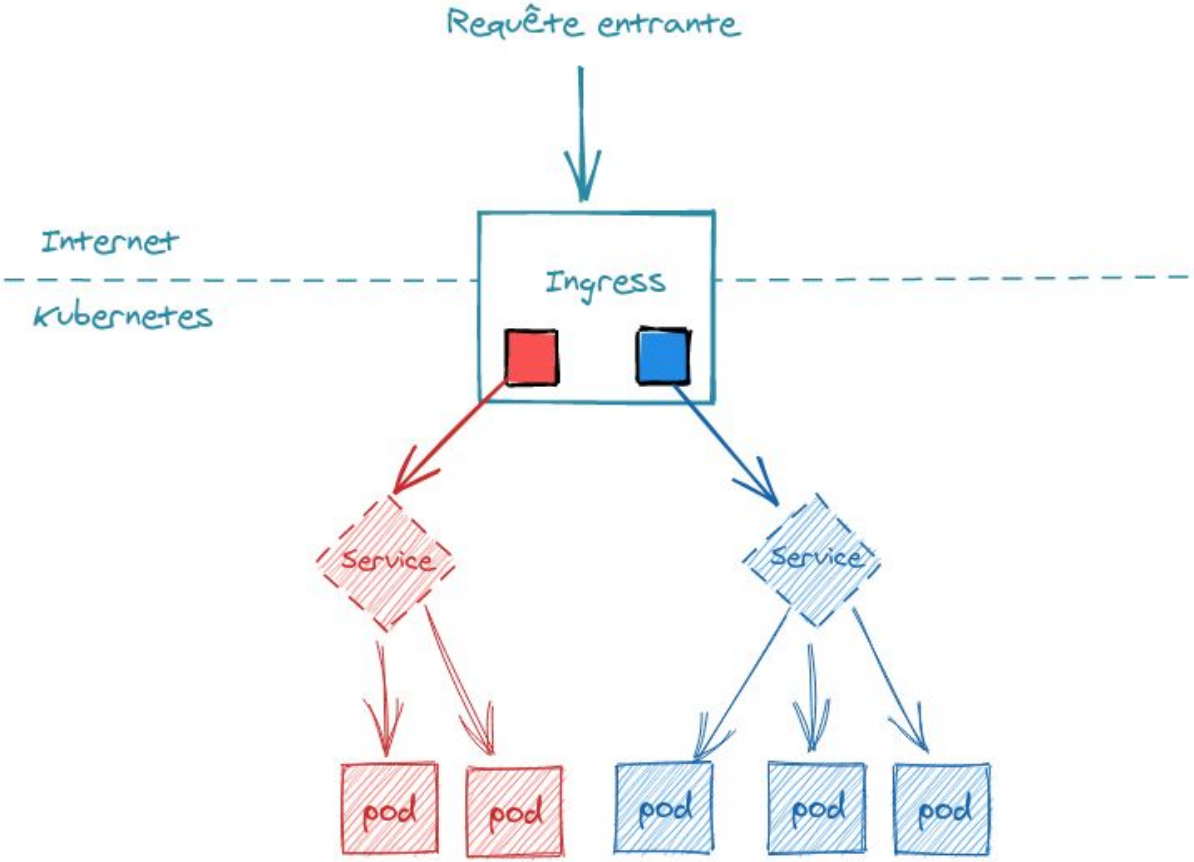
Il est aussi possible de mapper un Service avec un nom de domaine en spécifiant le paramètre spec.externalName.

```
apiVersion: v1

kind: Service

metadata:

  name: my-service

  namespace: prod

spec:

  type: ExternalName

  externalName: my.database.example.com
```

KUBERNETES: INGRESS

- L'objet Ingress permet d'exposer un Service à l'extérieur d'un cluster Kubernetes
- Il permet de fournir une URL visible permettant d'accéder un Service Kubernetes
- Il permet d'avoir des terminations TLS, de faire du Load Balancing, etc...

KUBERNETES: INGRESS



KUBERNETES: INGRESS

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: orsys
spec:
  rules:
  - host: orsys.fr
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

KUBERNETES: INGRESS CONTROLLER

Pour utiliser un Ingress, il faut un Ingress Controller. Un Ingress permet de configurer une règle de reverse proxy sur l'Ingress Controller.

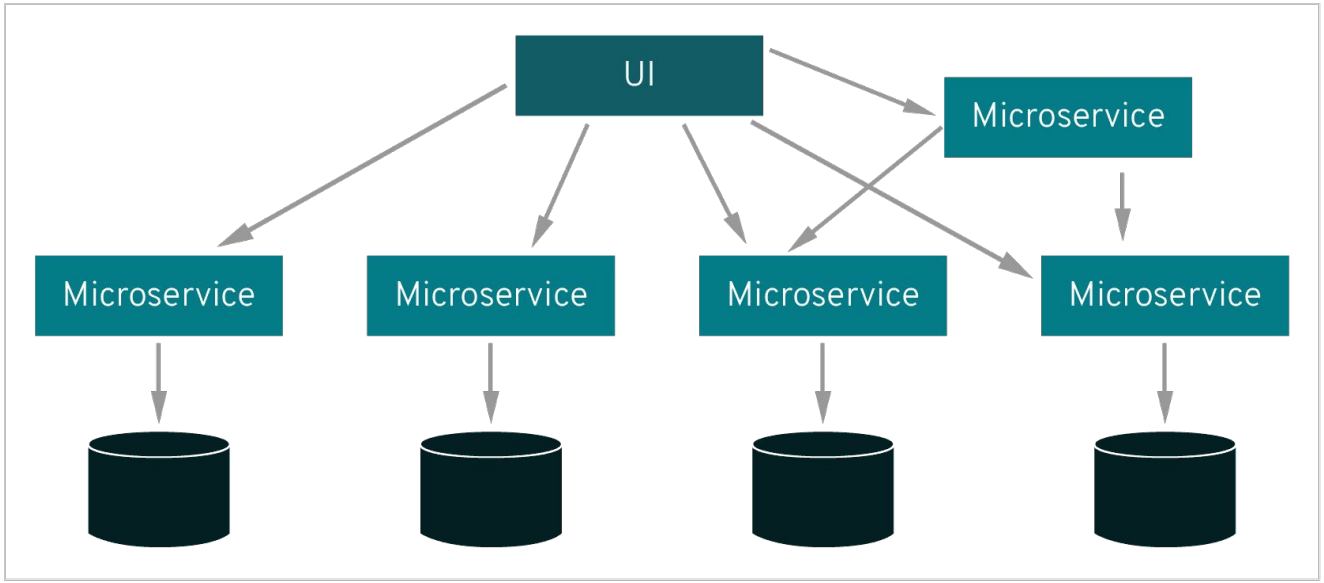
- Nginx Controller : https://github.com/kubernetes/ingress-nginx
- Traefik : https://github.com/containous/traefik
- Istio : https://github.com/istio/istio
- Linkerd : https://github.com/linkerd/linkerd
- Contour : https://www.github.com/heptio/contour/

KUBERNETES

Service Mesh

PROBLÉMATIQUE

- Découpage des applications en micro services
- Communication inter service (est-west) accrue



PROBLÉMATIQUE

- Sans service mesh la logique de communication est codée dans chaque service
- Problématiques de sécurité ? Comment implémenter TLS entre les micro services ?
- Chaque micro service doit implémenter la logique métier ainsi les fonctions réseau, sécurité et fiabilité
- Augmente la charge sur les équipes de développement
- Disparité des langages de programmation : implémentation de la même logique N fois

PROBLÉMATIQUE

L'augmentation du nombre du nombre de micro services peut provoquer :

- Une latence entre les services
- Peu de visibilité sur la santé des services individuels
- Peu de visibilité sur l'inter dépendance des services

KUBERNETES : SERVICE MINIMUM

- L'objet Service supporte uniquement TCP/UDP ainsi que de la répartition de charge basique
- L'objet Ingress utilise un point central de communication : l'ingress contrôleur
- Pas d'objets natifs pour faire du routage applicatif poussé
- Nécessité d'augmenter les fonctionnalités par un service tiers

SERVICE MESH

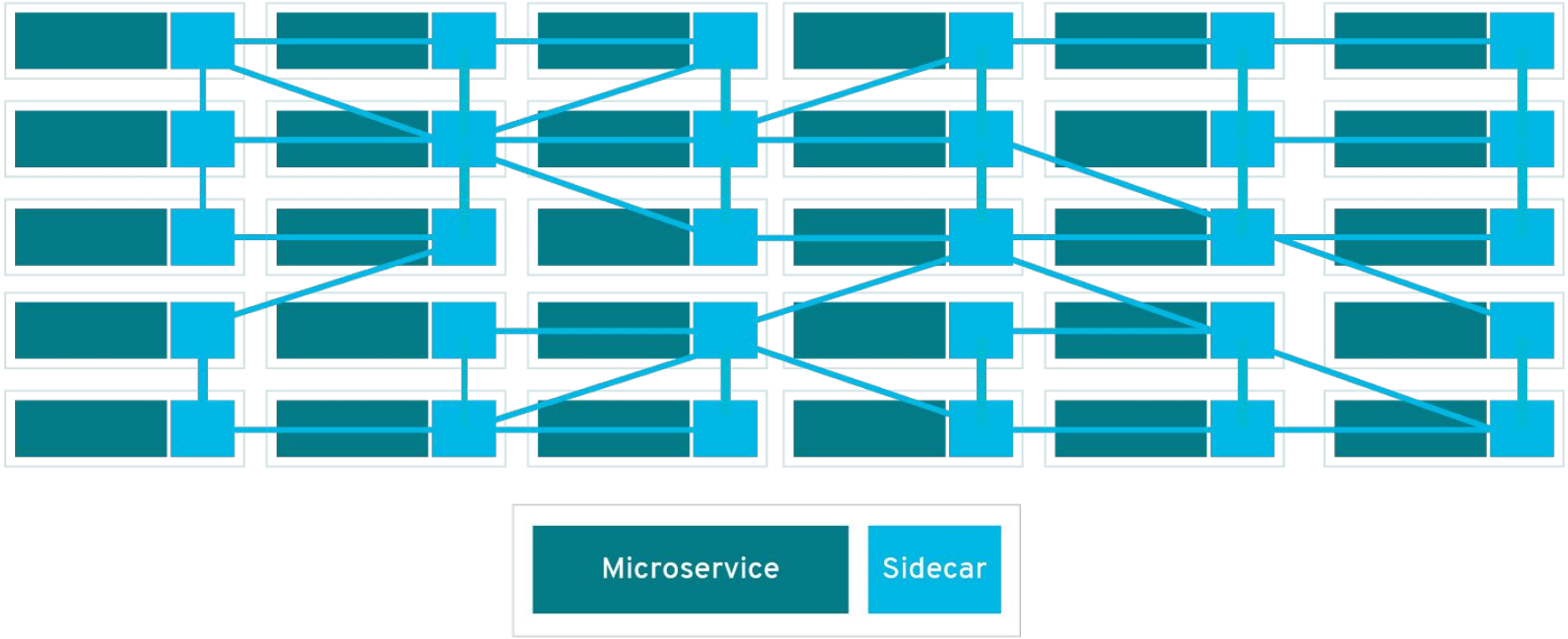
Les service mesh déportent la logique de communication au niveau de l'infrastructure et non plus au niveau de l'application. Les service mesh sont en général composés de deux plans:

- Un plan de données : effectue la communication entre les micro services.
- Un plan de contrôle : Programme le data plane et fournit des outils de configuration et de visualisation (CLI/Dashboard)

SERVICE MESH : PLAN DE DONNÉES

- Se basent sur un réseau de proxy
- Ne modifient pas l'application
- S'exécutent "à coté" : concept de sidecars
- Ces sidecars s'exécutent dans les même pods que l'application mais dans un conteneur différent

SERVICE MESH : PLAN DE DONNÉES



SERVICE MESH : PLAN DE CONTRÔLE

- Pilote le plan de donnée
- Permet la configuration de règles de routage applicatives
- Cartographie la communication entre les micro services
- Fourni des métriques applicatives :

  Latences

  Défaillances

  Logique de retry (désengorgement du réseau)

SERVICE MESH : AVANTAGES

- Permettent aux développeurs de se focaliser sur l'application et non pas sur la communication des services
- Couche commune de communication qui facilite le débogage et l'identification des problèmes
- Évite automatiquement certaines défaillances grace à du routage intelligent

SERVICE MESH : INCONVÉNIENTS

- Ajoute une surcouche à Kubernetes qui est déjà complexe
- Difficulté de migrer d'un service mesh à un autre
- Augmentent la nombre de conteneurs et de ressources consommées sur un cluster
- Paradoxalement les proxy rajoutent de la latence

SERVICE MESH : LES SOLUTIONS

Aujourd'hui les solutions sont multiples et offrent toutes plus ou moins les même fonctionnalités :

- Gestion du TLS mutuel et des certificats
- Authentification et autorisation
- Monitoring et traçage applicatif
- Routage du trafic via des règles applicatives

SERVICE MESH : LES SOLUTIONS

Istio : Plus connu et le plus complexe, open sourcé par Google

Linkerd : Hébergé dans la CNCF en incubation, open sourcé par Twitter

Consul : Développé par Hashicorp

Traefik Maesh : Développé par Traefik Lab

KUBERNETES

STOCKAGE

KUBERNETES : VOLUMES

Fournir du stockage persistent aux pods

Fonctionnent de la même façon que les volumes Docker pour les volumes hôte :

- EmptyDir ~= volumes docker
- HostPath ~= volumes hôte

Support de multiples backend de stockage :

- GCE : PD
- AWS : EBS
- GlusterFS / NFS
- Ceph
- iSCSI

KUBERNETES : VOLUMES

On déclare d'abord le volume et on l'affecte à un pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-persistent-storage
      mountPath: /data/redis
  volumes:
  - name: redis-persistent-storage
    emptyDir: {}
```

KUBERNETES : STORAGE CLASS

- Permet de définir les différents types de stockage disponibles

- Utilisé par les Persistent Volumes pour solliciter un espace de stockage au travers des Persistent Volume Claims

KUBERNETES : STORAGE CLASS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: us-east-1d, us-east-1c
  iopsPerGB: "10"
```

KUBERNETES : PERSISTENTVOLUMECLAIMS

- Ressource utilisée et vue comme une requête pour solliciter du stockage persistant
- Offre aux PV une variété d'options en fonction du cas d'utilisation
- Utilisé par les StatefulSets pour solliciter du stockage (Utilisation du champ volumeClaimTemplates)

KUBERNETES : PERSISTENTVOLUMECLAIMS

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: storage-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: "slow"
```

KUBERNETES : PERSISTENTVOLUME

- Composant de stockage dans le cluster kubernetes
- Stockage externe aux noeuds du cluster
- Cycle de vie indépendant du pod qui le consomme
- Peut être provisionné manuellement par un administrateur ou dynamiquement grâce à une StorageClass

KUBERNETES : PERSISTENTVOLUME

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: persistent-volume-1
spec:
  storageClassName: slow
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

KUBERNETES : CSI

- Container Storage Interface
- Équivalent de CNI mais pour les volumes
- Avant Kubernetes 1.13, tous les drivers de volumes étaient in tree
- Le but de la séparation est de sortir du code du core de Kubernetes
- GA depuis Kubernetes 1.13

KUBERNETES : CSI

La plupart des volumes supportés dans Kubernetes supportent maintenant CSI :

Amazon EBS

Google PD

Cinder

GlusterFS

La liste exhaustive est disponible ici

KUBERNETES

GESTION DE LA CONFIGURATION DES APPLICATIONS

KUBERNETES : CONFIGMAPS

- Objet Kubernetes permettant de stocker séparément les fichiers de configuration
- Il peut être créé d'un ensemble de valeurs ou d'un fichier resource Kubernetes (YAML ou JSON)
- Un ConfigMap peut sollicité par plusieurs pods

KUBERNETES : CONFIGMAP ENVIRONNEMENT (1/2)

```
apiVersion: v1
data:
  username: admin
  url: https://api.particule.io
kind: ConfigMap
metadata:
  name: web-config
```

KUBERNETES : CONFIGMAP ENVIRONNEMENT (2/2)

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: USERNAME
          valueFrom:
            configMapKeyRef:
              name: web-config
              key: username
        - name: URL
          valueFrom:
            configMapKeyRef:
              name: web-config
              key: url
  restartPolicy: Never
```

KUBERNETES : CONFIGMAP VOLUME (1/2)

```
apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  name: redis-config
```

KUBERNETES : CONFIGMAP VOLUME (2/2)

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "head -v /etc/config/*" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: redis-config
  restartPolicy: Never
```

KUBERNETES : SECRETS

- Objet Kubernetes de type secret utilisé pour stocker des informations sensibles comme les mots de passe, les tokens, les clés SSH...
- Similaire à un ConfigMap, à la seule différence que le contenu des entrées présentes dans le champ data sont encodés en base64.
- Il est possible de directement créer un Secret spécifique à l'authentification sur une registry Docker privée.
- Il est possible de directement créer un Secret à partir d'un compte utilisateur et d'un mot de passe.

KUBERNETES : SECRETS

- S'utilisent de la même façon que les ConfigMap
- La seule différence est le stockage en base64
- 3 types de secrets:

- Generic: valeurs arbitraire comme dans une ConfigMap
- tls: certificat et clé pour utilisation avec un serveur web
- docker-registry: utilisé en tant que imagePullSecret par un pod pour pouvoir pull les images d'une registry privée

```
kubectl create secret generic monSuperSecret --from-literal=username='monUser'
--from-literal=password='monSuperPassword"
```

KUBERNETES : SECRETS

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Les valeurs doivent être encodées en base64.

KUBERNETES

GESTION DES RESSOURCES

PODS RESOURCES : REQUEST ET LIMITS

- Permettent de gérer l'allocation de ressources au sein d'un cluster
- Par défaut, un pod/container sans request/limits est en best effort
- Request: allocation minimum garantie (réservation)
- Limit: allocation maximum (limite)
- Se base sur le CPU et la RAM

PODS RESOURCES : CPU

- 1 CPU est globalement équivalent à un cœur
- L'allocation se fait par fraction de CPU:
- 1 : 1 vCPU entier
- 100m : 0.1 vCPU
- 0.5 : 1/2 vCPU
- Lorsqu'un conteneur atteint la limite CPU, celui ci est throttled

PODS RESOURCES : RAM

- L'allocation se fait en unité de RAM:
- M : en base 10
- Mi : en base 2
- Lorsqu'un conteneur atteint la limite RAM, celui ci est OOMKilled

PODS RESOURCES : REQUEST ET LIMITS

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: "password"
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: wp
    image: wordpress
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

HORIZONTAL AUTOSCALING

- Permet de scaler automatiquement le nombre de pods d'un deployment
- Métriques classiques (CPU/RAM): En fonction d'un % de la request CPU/RAM
- Métriques custom (Applicative)

HORIZONTAL AUTOSCALING

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

LIMITRANGES

- L'objet LimitRange permet de définir les valeurs minimales et maximales des ressources utilisées par les containers et les pods
- L'objet LimitRange s'applique au niveau du namespace
- Les limites spécifiées s'appliquent à chaque pod/container créé dans le namespace
- Le LimitRange ne limite pas le nombre total de ressources disponibles dans le namespace

LIMITRANGES

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-example
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256 Mi
    type: Container
```

RESOURCEQUOTAS

- Un objet ResourceQuota limite le total des ressources de calcul consommées par les pods ainsi que le total de l'espace de stockage consommé par les PersistentVolumeClaims dans un namespace.

- Il permet aussi de limiter le nombre de pods, PVC et autres objets qui peuvent être créés dans un namespace

RESOURCEQUOTAS

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-and-ram
spec:
  hard:
    requests.cpu: 400m
    requests.memory: 200Mi
    limits.cpu: 600m
    limits.memory: 500Mi
```

KUBERNETES

SCHEDULING

AFFINITÉ / ANTI-AFFINITÉ

2 types de règles :

- Affinité de nodes
- Affinité / Anti-affinité de pod

AFFINITÉ DE NODES

- Permets de scheduler des workloads sur un nœud en particulier

- Paramétrable en fonction de label

AFFINITE DE NODES

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

AFFINITE / ANTI-AFFINITE

- Permet de scheduler des pods en fonction des labels
- Sur un même nœud (collocation)
- Sur des nœuds différents

AFFINITE / ANTI-AFFINITE

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 100
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security
              operator: In
              values:
              - S2
          topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
    image: k8s.gcr.io/pause:2.0
```

TAINTS ET TOLERATIONS

- Une taint permet l'inverse d'une affinité
- Permet à un nœud de refuser des pods
- Utilisé pour dédier des nœuds à un certain usage

```
kubectl taint nodes node1 key=value:NoSchedule
```

TAINTS ET TOLERATIONS

Aucun pod ne pourra être schedulé sur ce nœud à moins de tolérer la taint:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  tolerations:
  - key: "key"
    operator: "Exists"
    effect: "NoSchedule"
```

KUBERNETES

Service Mesh

SONDES : READINESS AND LIVENESS

- Permettent à Kubernetes de sonder l'état d'un pod et d'agir en conséquence
- 2 types de sonde : Readiness et Liveness
- 3 manières de sonder :
- TCP : ping TCP sur un port donné
- HTTP: http GET sur une url donnée
- Command: Exécute une commande dans le conteneur

SONDES : READINESS

- Gère le trafic à destination du pod
- Un pod avec une sonde readiness NotReady ne reçoit aucun trafic
- Permet d'attendre que le service dans le conteneur soit prêt avant de router du trafic
- Un pod Ready est ensuite enregistré dans les endpoints du service associé

SONDES : LIVENESS

- Gère le redémarrage du conteneur en cas d'incident
- Un pod avec une sonde liveness sans succès est redémarré au bout d'un intervalle défini
- Permet de redémarrer automatiquement les pods "tombés" en erreur

SONDES : EXEMPLE

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

KUBERNETES

UTILISATION ET DEPLOIEMENT DES RESSOURCES

KUBERNETES : KUBECTL

- Le seul (ou presque) outil pour interagir avec des clusters Kubernetes
- Utilise un fichier de configuration (kubeconfig) pour communiquer avec l'API de Kubernetes
- Le(s) fichier(s) se trouve(nt) par défaut dans ~/.kube/config
- Le fichier de config. contient :
- L'adresse(URI) de l'APIServer
- Les chemins des certificats TLS utilisés pour l'authentification
- Fichier kubeconfig peut être passé en paramètre de kubectl avec le flag --kubeconfig

KUBECONFIG

Un seul fichier pour gérer tous ses clusters avec trois informations :

- Serveurs (IP, CA Cert, Nom)
- Users (Nom, Certificat, Clé)
- Context, association d'un user et d'un serveur

Stocké par défaut dans ~/.kube/config

KUBERNETES : KUBECTL

Afficher la liste des ressources API supportées par le serveur:

```
$ kubectl api-resources
NAME                              SHORTNAMES   APIGROUP                       N
configmaps                        cm
limitranges                       limits
namespaces                        ns
nodes                             no
persistentvolumeclaims            pvc
persistentvolumes                 pv
pods                              po
secrets
services                          svc
daemonsets                        ds           apps
deployments                       deploy       apps
replicasets                       rs           apps
statefulsets                      sts          apps
horizontalpodautoscalers          hpa          autoscaling
cronjobs                          cj           batch
jobs                                           batch
```

KUBERNETES : KUBECTL

- Afficher les noeuds du cluster :
  kubectl get nodes

- Ces commandes sont équivalentes:
  kubectl get no
  kubectl get nodes

- Afficher les namespaces
  kubectl get ns
  kubectl get namespaces

- Par défaut, kubectl utilise le namespace default. Il est possible de sélectionner un namespace avec l'option -n ou --namespace

  kubectl -n kube-system get pods

KUBERNETES : KUBECTL

Afficher les pods (pour le namespace default):

```
kubectl get pods
kubectl get pod
```

Afficher les services (pour le namespace default):

```
kubectl get services
kubectl get svc
```

KUBERNETES : CRÉATION D'OBJETS KUBERNETES

- Les objets Kubernetes sont créés sous la forme de fichiers JSON ou YAML et envoyés à l'APIServer
- Possible d'utiliser la commande kubectl run, mais limitée aux Deployments et aux Jobs
- L'utilisation de fichiers YAML permet de les stocker dans un système de contrôle de version comme git, mercurial, etc...
- La documentation de référence pour l'API Kubernetes https://kubernetes.io/docs/reference/#api-reference

KUBERNETES : CRÉATION D'OBJETS KUBERNETES

- Pour créer un object Kubernetes depuis votre fichier YAML, utilisez la commande kubectl create :

  kubectl create -f object.yaml

- Il est possible de créer des objets Kubernetes à partir d'une URL :

  kubectl create -f https://raw.githubusercontent.com/kubernetes/examples/master/guestbook/frontend-deployment.yaml

- Pour les supprimer exécuter simplement :

  kubectl delete -f object.yaml

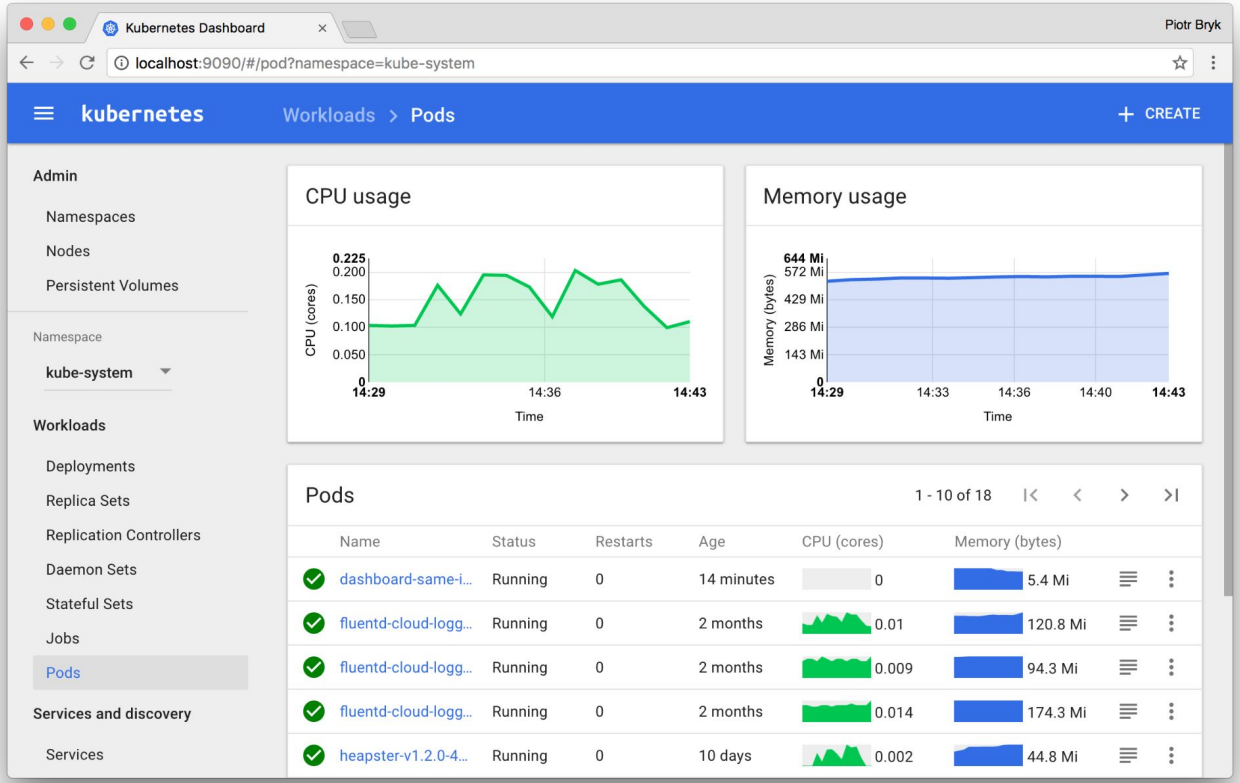
- Mettre à jour un objet Kubernetes en écrasant la configuration existante:

  kubectl replace -f object.yaml

KUBERNETES : KUBERNETES DASHBOARD

- Interface graphique web pour les clusters Kubernetes
- Permet de gérer les différents objets Kubernetes créés dans le(s) cluster(s).
- Installé par défaut dans minikube

KUBERNETES : KUBERNETES DASHBOARD



KUBERNETES : KUBERNETES DASHBOARD

- Pour déployer le Dashboard, exécuter la commande suivante:

$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended/kubernetes-dashboard.yaml

- Pour accéder au Dashboard, il faut établir une communication entre votre poste et le cluster Kubernetes :

$ kubectl proxy

- L'accès se fait désormais sur :

http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/

KUBERNETES

HELM

QU'EST-CE QUE HELM ?

- Outil de packaging d'application Kubernetes
- Developpé en Go
- Actuellement en v3
- Projet graduated de la CNCF
- https://github.com/helm/helm

POURQUOI HELM ?

- Applique le principe DRY (Don't Repeat Yourself)

    Mécanisme de templating (Go templating)

    Variabilisation des ressources générées

- Facilité de versionnement et de partage (repository Helm)
- Helm permet d'administrer les Releases

    Rollbacks / upgrades d'applications

CONCEPTS

Concept	Description
Chart	Ensemble de ressources permettant de definir une application Kubernetes
Config	Valeurs permettant de configurer un Chart (`values.yaml`)
Release	Chart deployé avec une Config

COMPARAISON AVEC DES MANIFESTS YAML

- Permet de mettre en place le DRY (Don't Repeat Yourself)
- customisation via fichier de configuration YAML
- Définition d'une seule source de vérité
- Les ressources sont packagées
- Packages déployés via des Releases

STRUCTURE D'UN CHART

- Chart.yaml pour définir le chart ainsi que ses metadatas
- values.yaml sert à definir les valeurs de configuration du Chart par défaut
- crds/: Dossier qui recense les CRDs
- templates/: les templates de manifeste Kubernetes en YAML

CHART.YAML

Le fichier de configuration du Chart dans lequel sont définies ses metadatas.

```
---
apiVersion: v2
description: Hello World Chart.
name: hello-world-example
sources:
  - https://github.com/prometheus-community/helm-charts
version: 1.3.2
appVersion: 0.50.3
dependencies: []
```

STRUCTURE DU VALUES.YAML

Chaque attribut est ensuite disponible au niveau des templates

```
---
### Provide a name in place of kube-prometheus-stack for `app:` labels
##
applicationName: ""

### Override the deployment namespace
##
namespaceOverride: ""


### Apply labels to the resources
##
commonLabels: {}
```

SURCHARGE DU VALUES.YAML

```
---
# values-production.yaml
commonLabels:
  env: prod
```

tree

```
.
├── Chart.yaml
├── templates
│   ├── application.yaml
│   ├── configuration.yaml
│   └── secrets.yaml
├── values-production.yaml
├── values-staging.yaml
└── values.yaml
```

TEMPLATES

Helm permet de variabiliser les manifestes Kubernetes, permettant de créer et configurer des ressources dynamiquement. Le langage Go Template est utilisé.

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: {{ .Chart.Name }}
  labels:
    app.kubernetes.io/managed-by: "Helm"
    chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    release: {{ .Release.Name | quote }}
    version: 1.0.0
spec:
  containers:
  - image: "{{ .Values.helloworld.image.name }}:{{ .Values.helloworld.image.tag }}"
    name: helloworld
```

GESTION DES REPOSITORIES

- Un Repository Helm permet de distribuer et versionner des Charts
- Contient un index.yaml listant les Charts packagés disponibles par version
- Deux méthodes de déploiement possibles
    - Via HTTP en tant que fichiers statiques
    - Via OCI en utilisant une Registry (depuis Helm v3)

COMMANDES COMMUNES

```
$ helm repo add stable https://charts.helm.sh/stable
"stable" has been added to your repositories
$ helm repo update
$ helm install stable/airflow --generate-name
helm install stable/airflow --generate-name
NAME: airflow-1616524477
NAMESPACE: defaul
...

$ helm upgrade airflow-1616524477 stable/airflow
helm upgrade airflow-1616524477 stable/airflow
Release "airflow-1616524477" has been upgraded. Happy Helming!
$ helm rollback airflow-1616524477
Rollback was a success! Happy Helming!
$ helm uninstall airflow-1616524477
release "airflow-1616524477" uninstalled
```

KUBECTL : ADVANCED USAGE

- Il est possible de mettre à jour un service sans incident grâce ce qui est appelé le rolling-update.
- Avec les rolling updates, les ressources qu'expose un objet Service se mettent à jour progressivement.
- Seuls les objets Deployment, DaemonSet et StatefulSet support les rolling updates.
- Les arguments maxSurge et maxUnavailabe définissent le rythme du rolling update.
- La commande kubectl rollout permet de suivre les rolling updates effectués.

KUBECTL : ADVANCED USAGE

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: frontend
  replicas: 2
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      name: nginx
      labels:
        app: frontend
    spec:
      containers:
        - image: nginx:1.9.1
          name: nginx
```

KUBECTL : ADVANCED USAGE

$ kubectl create -f nginx.yaml --record

deployment.apps/nginx created

- Il est possible d'augmenter le nombre de pods avec la commande kubectl scale :

  kubectl scale --replicas=5 deployment nginx

- Il est possible de changer l'image d'un container utilisée par un Deployment :

  kubectl set image deployment nginx nginx=nginx:1.15

KUBECTL : ADVANCED USAGE

Dry run. Afficher les objets de l'API correspondant sans les créer :

```
kubectl run nginx --image=nginx --dry-run
```

Démarrer un container en utiliser une commande différente et des arguments différents :

```
kubectl run nginx --image=nginx  --command -- <cmd> <arg1> ... <argN>
```

Démarrer un Cron Job qui calcule π et l'affiche toutes les 5 minutes :

```
kubectl run pi --schedule="0/5 * * * ?" --image=perl --restart=OnFailure -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

KUBECTL : ADVANCED USAGE

Se connecter à un container:

kubectl run -it busybox --image=busybox -- sh

S'attacher à un container existant :

kubectl attach my-pod -i

Accéder à un service via un port :

kubectl port-forward my-svc 6000

KUBECTL : LOGGING

Utiliser kubectl pour diagnostiquer les applications et le cluster kubernetes :

kubectl cluster-info

kubectl get events

kubectl describe node <NODE_NAME>

kubectl logs [-f] <POD_NAME>

KUBECTL : MAINTENANCE

Obtenir la liste des noeuds ainsi que les informations détaillées :

kubectl get nodes

kubectl describe nodes

KUBECTL : MAINTENANCE

Marquer le noeud comme unschedulable (+ drainer les pods) et schedulable :

kubectl cordon <NODE_NAME>

kubectl drain <NODE_NAME>

kubectl uncordon <NODE_NAME>

KUBERNETES

COMMENT DEPLOYER ?

BARE METAL, PRIVATE ET PUBLIC CLOUDS

Managed K8S (AKS, EKS, GKE) - Kops - Kubespray - Kubeadm

Cluster Deployment

Cluster Lifecycle

INFRASTUCTURE AS CODE

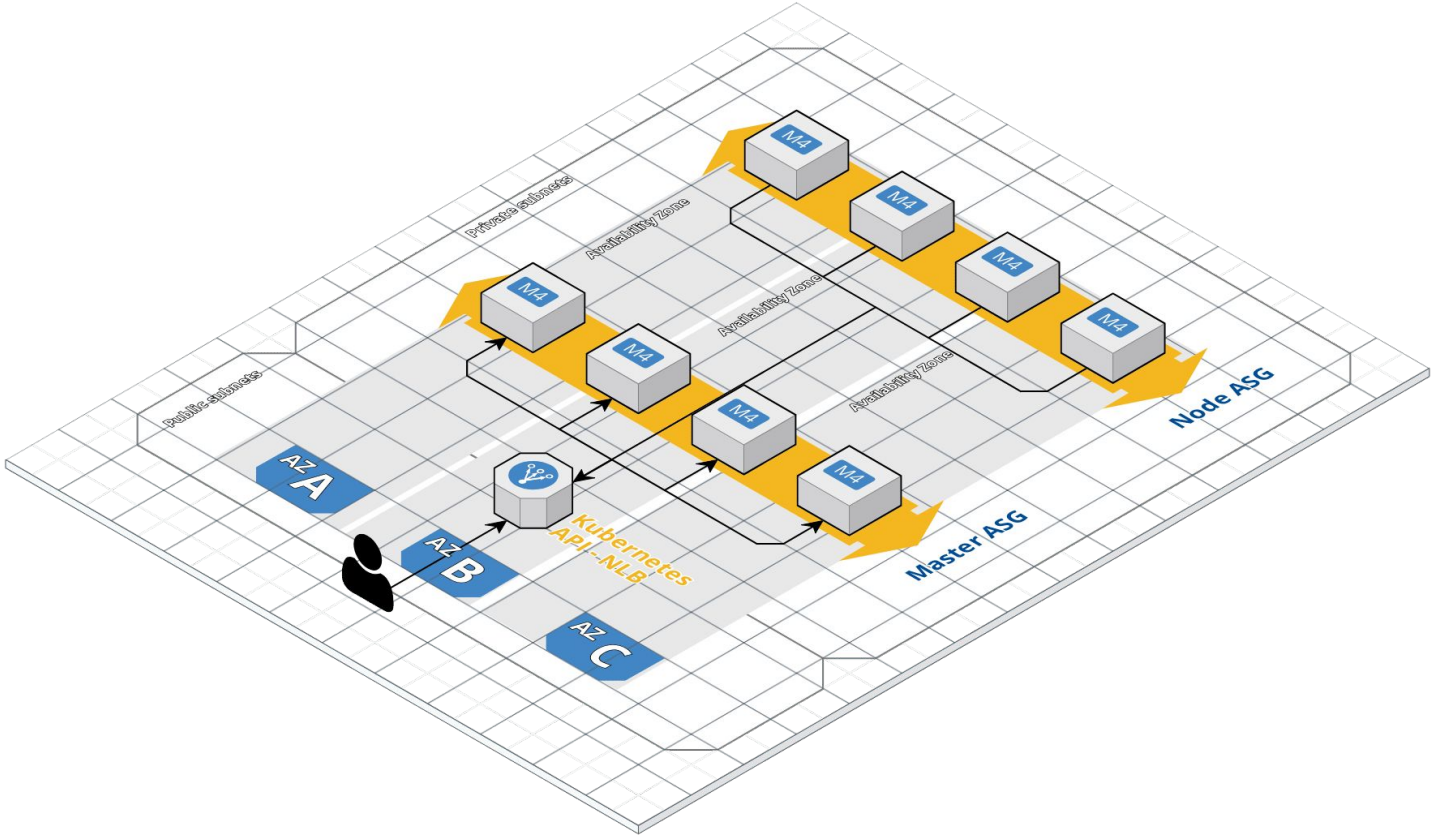
IaC : Infrastructure as Code

Terraform - CloudFormation - Cloud Deployment Manager - OpenStack Heat - Azure Resource Manager

Infrastructure déclarée

Infrastructure immuable

IMPLÉMENTATION DE RÉFÉRENCE



QUE CHOISIR ?

Je veux utiliser Kubernetes

Cloud ?

Cloud public ou privé ?

Configuration particulière ?

Multiple cloud providers ?

Homogénéité des outils ?

LOCAL KUBERNETES

Minikube: Machine virtuelle locale

Kind: Kubernetes in Docker

k3s: Kubernetes léger

Docker for Mac/Windows

AWS EKS

Control plane managé par AWS

Amazon Linux / Ubuntu

CloudFormation / Terraform / eksctl

GKE

Control plane managé par GCP

Premier sur le marché

COS / Ubuntu

Terraform / Google Cloud SDK

AKS

Control plane managé par Azure

KUBEADM

Outil officiel de la communauté

Stable depuis v1.13.0

Ne provisionne pas de machine

Facilement personnalisabe

Respect des best practices

Peut être utilisé par d'autres outils

KUBESPRAY

Basé sur Ansible

Dense, permet d'installer un nombre important de plugins

Multiples OS

Support Kubeadm

KOPS

Déploie sur AWS/GCP/OpenStack et Digital Ocean

Cycle de release lent

Facilement personnalisable

Multiples OS

Supporte Cloudformation and Terraform

KUBERNETES

SECURITE ET CONTROLE D'ACCES

AUTHENTICATION & AUTORISATION

RBAC (Role Based Access Control)

ABAC (Attribute-based access control)

WebHook

Certificates

Token

RBAC

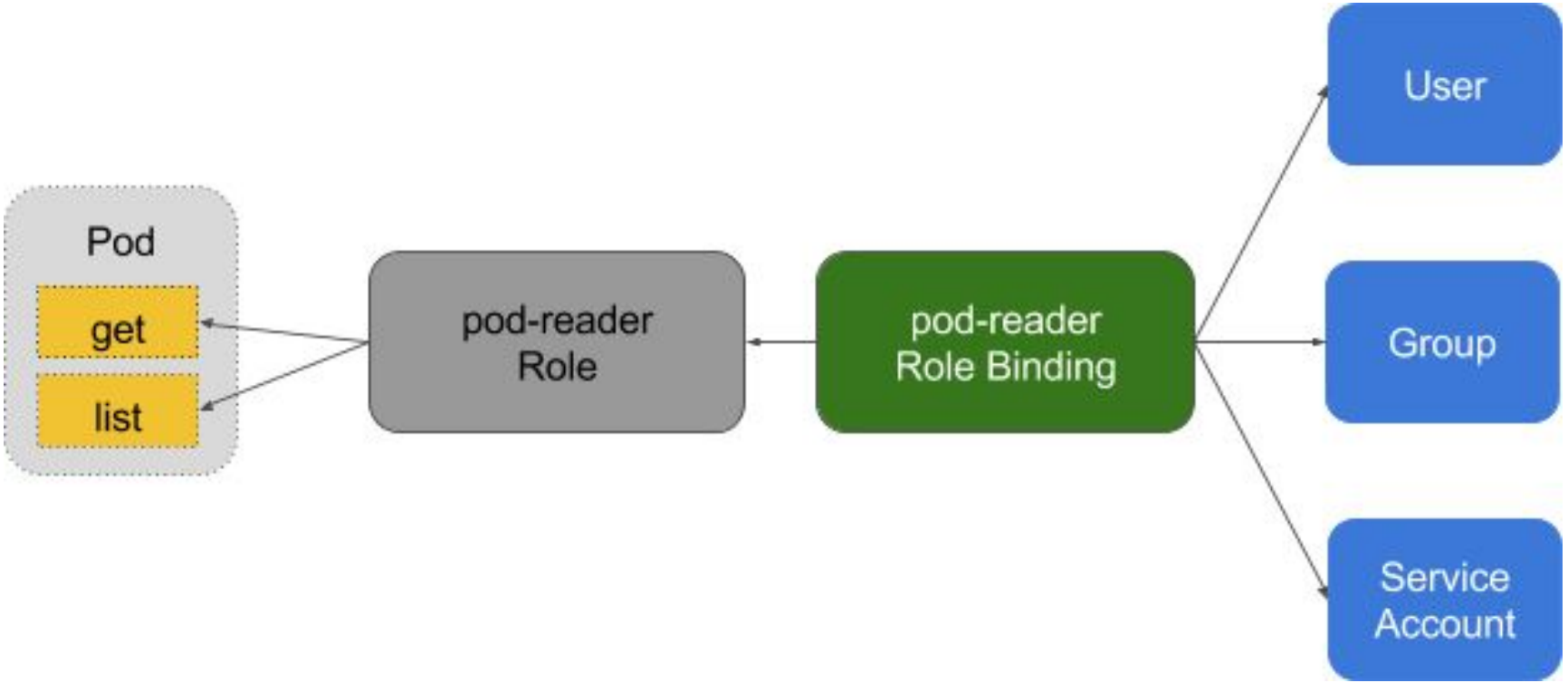
3 entités sont utilisées :

Utilisateurs représentés par les Users ou les ServiceAccounts

Ressources représentées par les Deployments, Pods, Services, etc...

les différentes opérations possibles : create, list, get, delete, watch, patch

RBAC


Pod
get
list
pod-reader
Role
pod-reader
Role Binding
User
Group
Service
Account

SERVICE ACCOUNTS

Objet Kubernetes permettant d'identifier une application s'éxecutant dans un pod

Par défaut, un ServiceAccount par namespace

Le ServiceAccount est formatté ainsi :

```
system:serviceaccount:<namespace>:<service_account_name>
```

SERVICE ACCOUNTS

```
apiVersion: v1

kind: ServiceAccount

metadata:

    name: default

    namespace: default
```

ROLE

L'objet Role est un ensemble de règles permettant de définir quelle opération (ou verbe) peut être effectuée et sur quelle ressource

Le Role ne s'applique qu'à un seul namespace et les ressources liées à ce namespace

ROLE

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ROLEBINDING

L'objet RoleBinding va allouer à un User, ServiceAccount ou un groupe les permissions dans l'objet Role associé.

Un objet RoleBinding doit référencer un Role dans le même namespace.

L'objet roleRef spécifié dans le RoleBinding est celui qui crée le liaison

ROLEBINDING

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

CLUSTERROLE

L'objet ClusterRole est similaire au Role à la différence qu'il n'est pas limité à un seul namespace

Il permet d'accéder à des ressources non limitées à un namespace comme les nodes

CLUSTERROLE

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

CLUSTERROLEBINDING

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-reads-all-pods
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

NETWORKPOLICIES

La ressource NetworkPolicy est une spécification permettant de définir comment un ensemble de pods communiquent entre eux ou avec d'autres endpoints

Le NetworkPolicy utilisent les labels pour sélectionner les pods sur lesquels s'appliquent les règles qui définissent le trafic alloué sur les pods sélectionnés

Le NetworkPolicy est générique et fait partie de l'API Kubernetes. Il est nécessaire que le plugin réseau déployé supporte cette spécification

NETWORKPOLICIES

- DENY tout le trafic sur une application
- LIMIT le trafic sur une application
- DENY le trafic all non alloué dans un namespace
- DENY tout le trafic venant d'autres namespaces
  exemples de Network Policies : https://github.com/ahmetb/kubernetes-network-policy-recipes

NETWORKPOLICIES

Exemple de NetworkPolicy permettant de blocker le trafic entrant :

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

PODSECURITYPOLICIES

- Permet de contrôler les privilèges d'un pod
- Permet de définir ce qui est autorisé pendant l'exécution du pod
- A utiliser dans un contexte multi-tenant et quand les pods ne viennent pas d'un tiers de confiance
- Peut-être combiné avec le RBAC
- Attention: Activer cette fonctionnalité peut endommager votre environnement
- Il faut une PSP par défaut

PODSECURITYPOLICIES

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot'
  readOnlyRootFilesystem: false
```

ADMISSION CONTROLLERS

Interceptent les requêtes sur l'API Kubernetes

Peut effectuer des modifications si nécessaires

Conception personnalisée possible

ADMISSION CONTROLLERS

DenyEscalatingExec

ImagePolicyWebhook

NodeRestriction

PodSecurityPolicy

SecurityContextDeny

ServiceAccount

FIN