Kubernetes

Aspects Administratifs

- https://moncompte.dawan.fr
  - Besoins/Attentes/Niveau d'entrée
  - Émargement bi-quotidien:
  - Évaluation
  - Niveau de sortie
  - -> Attestation

INTRODUCTION

- Mise en place de l'infrastructure
  - Debian-12 (avec Docker)
  - CPU : 2+
  - RAM : 6Go (4Go)
  - Réseau :
    - Carte 1 : NAT
    - Carte 2 : private network, IP : 192.168.50.4
  - Compte
    - vagrant (ssh auto + sudo auto)
    - vagrant ssh

INTRODUCTION

- Kubernetes est un orchestrateur de conteneurs
  - Ordonnancer des conteneurs sur des nodes -> **Scheduler**
  - Tolérance de pannes -> Vérification de l'état de santé du cluster (niveau node, niveau pod)
  - Répartition de charge
  - Scalabilité
    - Manuelle
    - Automatique : Nodes, Pods
  - Mises à jour
    - Plusieurs stratégies de mises à jour
    - Historisation des mises à jour
    - Possibilité de Rollback

INTRODUCTION

- Différents orchestrateurs
  - Docker Swarm
  - Nomad (Hashicorp)
  - Kubernetes - K8s (CNCF : Cloud Native Computing Foundation)
  - Openshift (RedHat)
    - Okd
  - Mesos (Apache)

Structure d'un orchestrateur

- Un cluster est composé de nodes (composants matériels) :
  - Machines physiques
  - Machines virtuelles
  - (Conteneurs -> Minikube et Kind)

- Nodes de contrôles
  - Manager (Swarm)
  - Master (Openshift)
  - Control-Plane (K8s)
  - Par configuration, on ne peut créer de ressources sur un control-plane ou un master.
  - Il est vivement conseillé de disposer d'au moins 3 nodes de contrôles
- Nodes de travail -> Worker

Structure d'un orchestrateur

- Distributions Kubernetes
  - Kubeadm (dsitribution cannonique)
  - Kubespray (~ kubeadm piloté par ansible)
  - RKE2 (kubernetes dans l'écosystème Rancher)
  - K3S (distrib pour arm / IoT, Etcd est remplacée par sqlite)
  - MicroK8s : (distribution pour prod simplifiée)
  - Minikube : (distrib sur un seul node => maquette)
  - Kind : (distrib sur une machine docker et les nodes sont des containers => maquette)

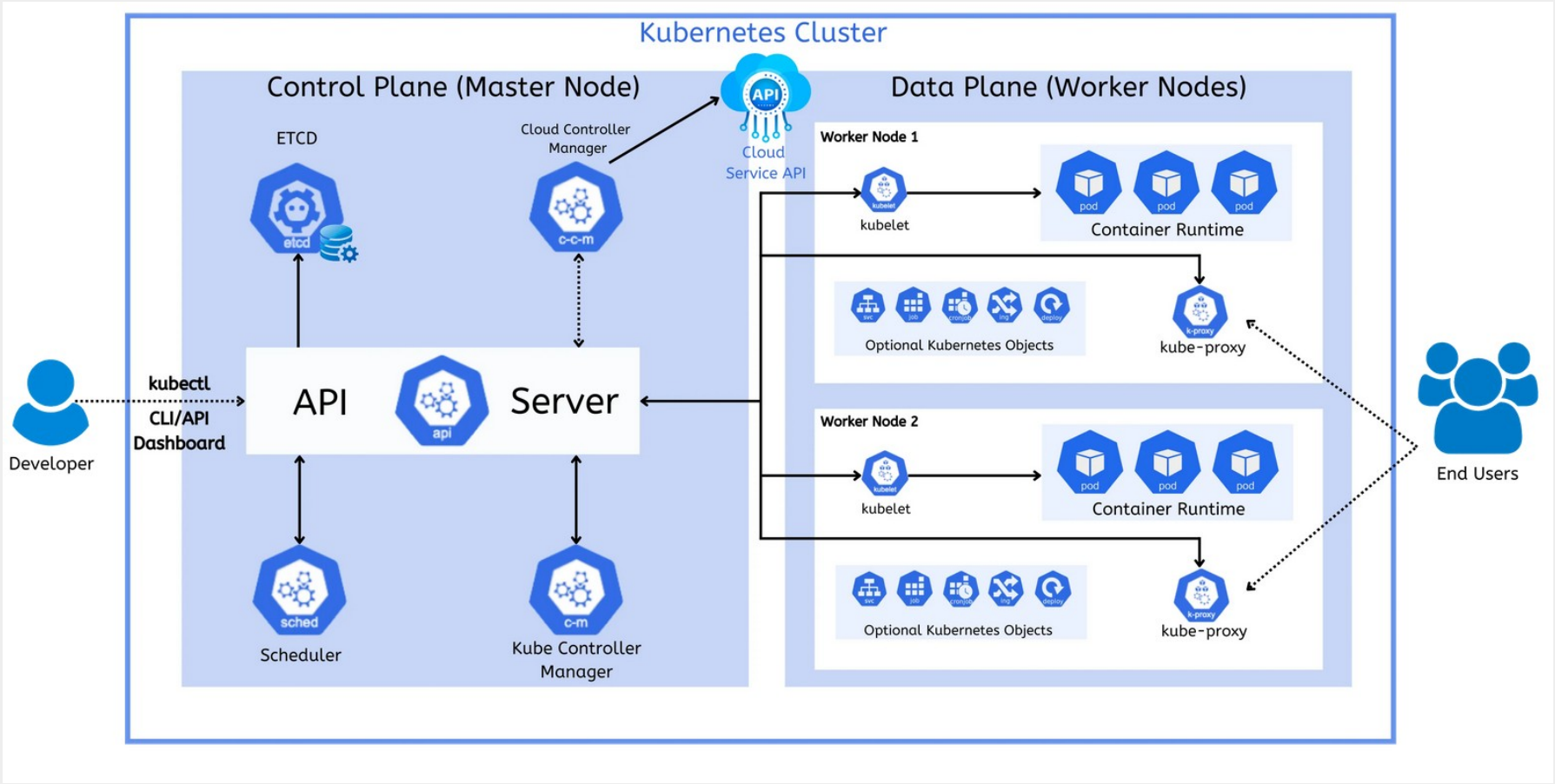
Structure d'un orchestrateur

- Limitations de Kubernetes (1.30)
  - 5000 nodes
  - 110 pods par nodes
  - 300 000 conteneurs
  - 150 000 pods au total (limitation K8s)

Structure d'un orchestrateur

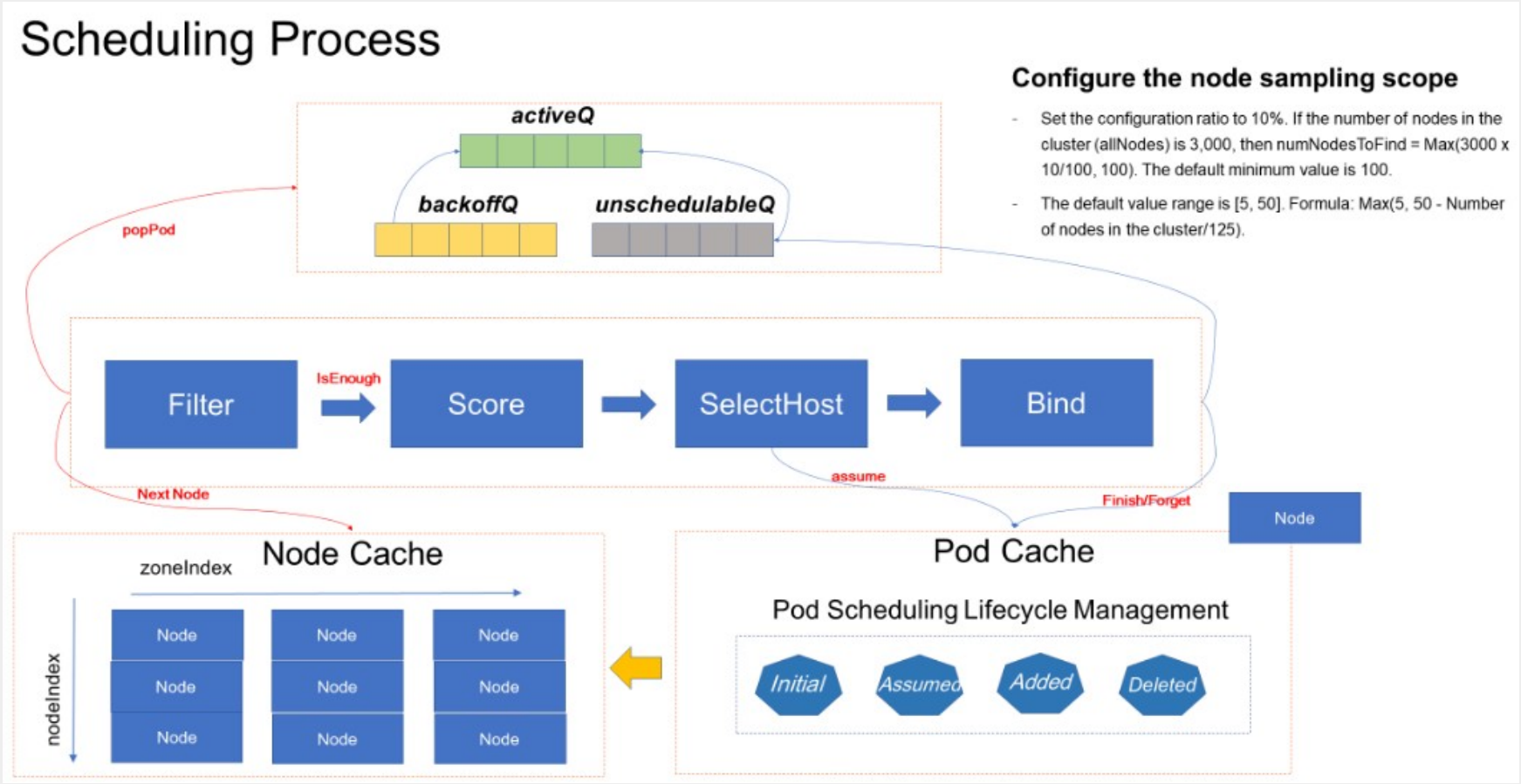
- Composants principaux
  - Machines maîtres : « **control plane** »
    - **etcd** : base de donnée de type « keystore » : magasin de clés / valeurs distribué => stocke l'état du cluster
    - **kube-controller-manager** : démon récolte tous les flux k8s => détermine l'état désiré du cluster
    - **scheduler** : processus responsable de l'installation des composants applicatifs « pods » sur les nœuds du cluster => ordonnance les flux de travail déterminés par le contrôleur
    - **apiServer** : démon d'API REST => centralise les communications dans k8s
  - Machines exécutants : « **data plane** » composé de noeuds « **workers** »
    - **kubelet** : démon recevant les ordres du scheduler => installe et gère les applicatifs conteneurisés « pods »
    - **kube-proxy** : démon responsable des règles réseaux bas niveau dans un nœud (cf infra « services »)

Structure d'un orchestrateur



Nb impair de Control Planes répliqués

Structure d'un orchestrateur



Structure d'un orchestrateur

- Notion de ressources
  - Tout est ressources avec Kubernetes.
  - La liste des ressources disponibles sur un cluster, à un moment donné, est donnée par la commande
  - kubectl **api-resources**
  - kubectl **explain pod.metadata**

- Attributs principaux d'une resource
  - **name**: hello-5f4ddcf477 => Unique
  - **uid**: b7dc0cd2-cc1f-4d97-b2c0-bd11703a4e5d => Unique
  - **metadata.labels** : {app : hello, ...} : organiser, sélectionner les objets par labels
  - **metadata.annotations** : {…} : utiliser pour documenter un objet via des client (CLI)
  - **spec.selector.matchLabels** : sélectionner des objets bas-niveau dans une resource + haut-niveau
    - ex : sélectionner **par labels** les **pods** à répliquer dans un **Deployment / ReplicaSet**

Réseau - K8S

- Réseau k8s « flat »
  - Par défaut, tous les éléments applicatifs « Pods » sont accessibles par leur IP, dans tout le cluster

    => On parle de réseau « flat »
  - Ips des nodes (cas KIND)

    - docker network inspect kind --format '**{{range .Containers }} {{ .IPv4Address }} {{ end }}**'
  - gamme d'adresses ip par défaut pour les ips des pods

    - k get nodes -o jsonpath='**{.items[*].spec.podCIDR}**'

Réseau - K8S CNI

- CNI : « Container Network Interface »
  - Spécifications de mise en réseau complexe des pods dans k8s (intra & inter nodes)
  - Plugins réseau de k8s chargés du contrôle du trafic intra & inter applications
  - Exemples : Flannel, Calico, Cilium, ...
  - 2 types de modèles :

    - overlay i.e encapsulation (ex : VXLAN, IPIP,…)

      + le sous réseeau sous jacent ne connaît pas les ip de pods (Software Design Network, config flexible)

      - l'encapsulation demande + de CPU et des trames plus grandes (headers pour les tunnels ...)

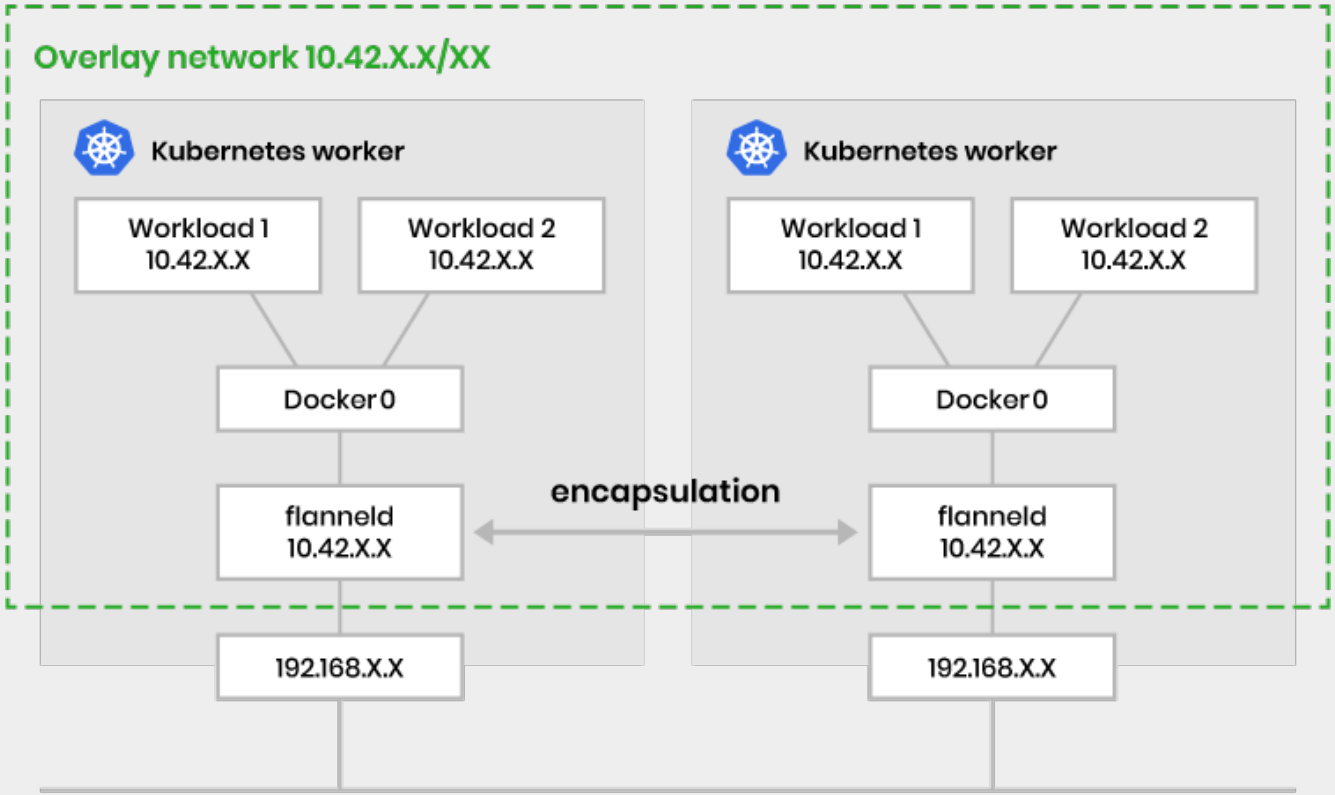
    - underlay (algortihme de routage seuls)

      + pas d'encapsulation : meilleures performances

      - configurations plus complexes à mesure que le réseau grossisse (scalablité)

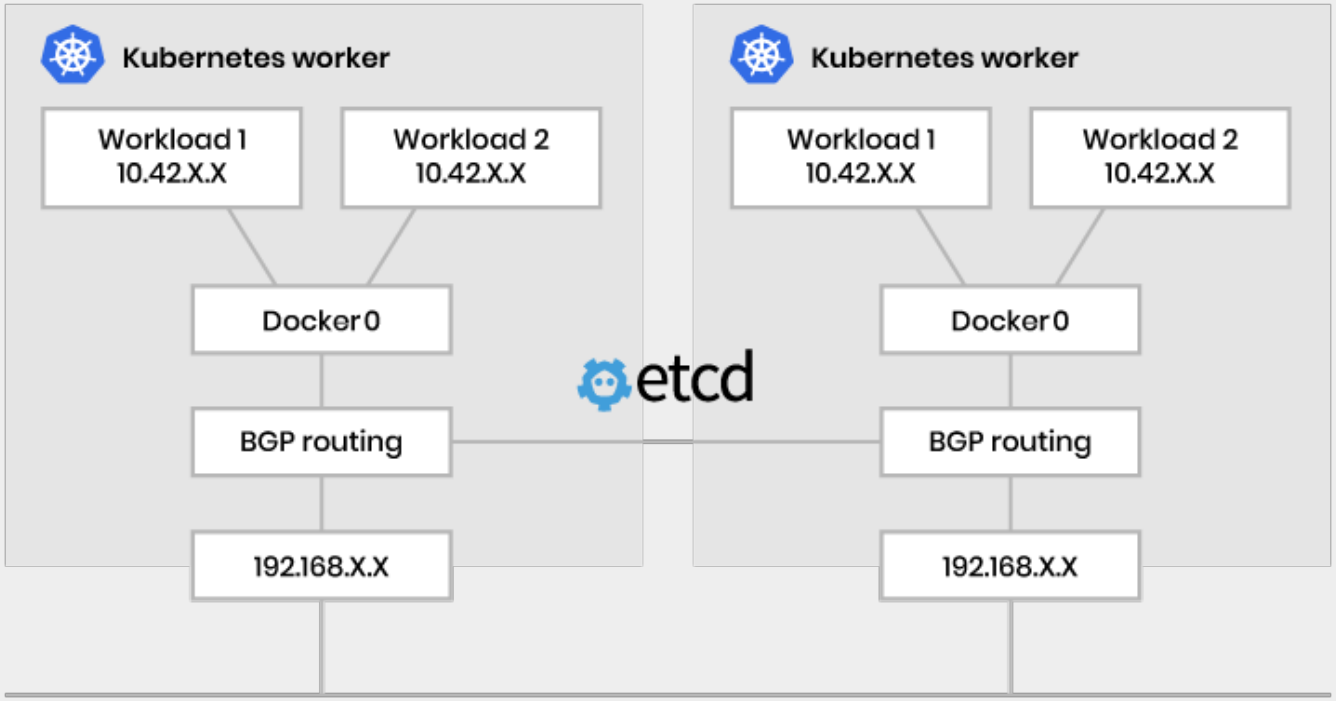
Réseau - K8S CNI

- Ex : Overlay Flannel (ou Calico with IPIP)



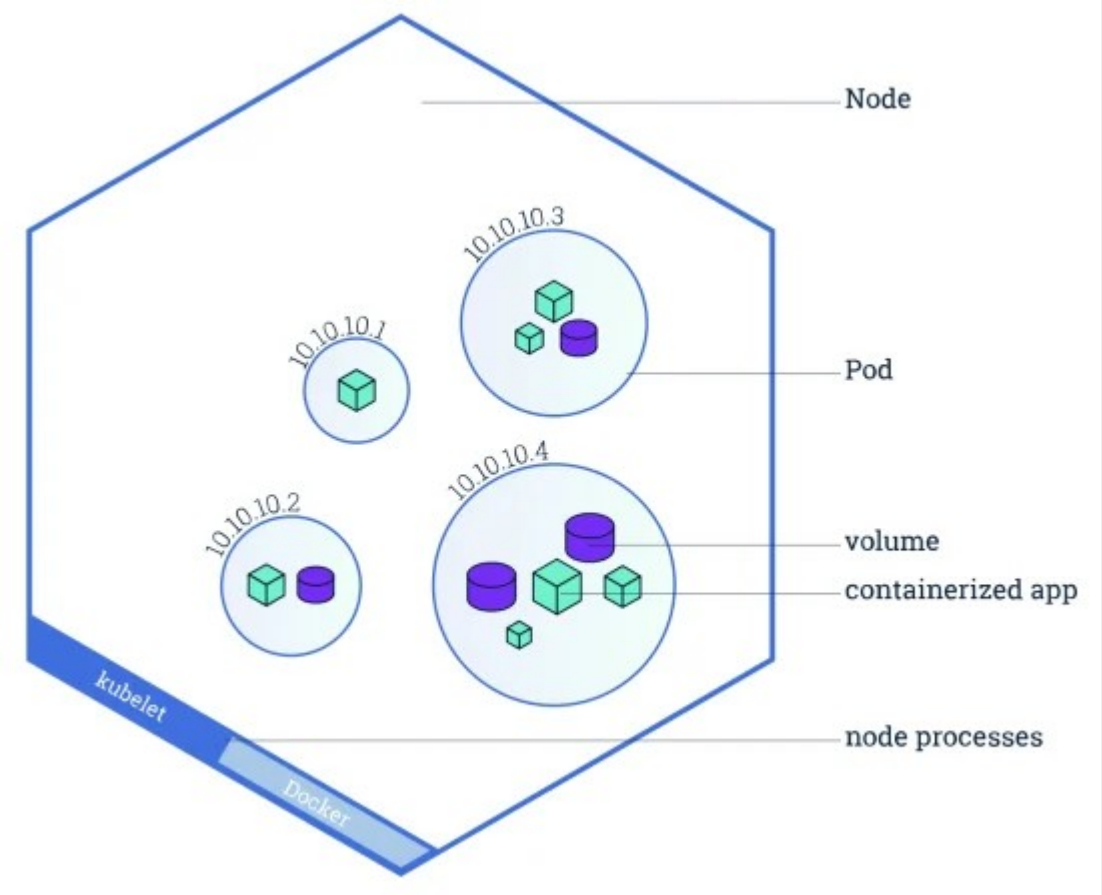
Réseau - K8S CNI

- Ex : Underlay k8s calico BGP



Pods - K8S

- Pod
  - Unité logicielle composée
    - d'un ou plusieurs conteneurs
    - et / ou des volumes de données
  - Partageant les namespaces linux:
    - net (iface lo et ip « publique »)
    - ipc
    - uts (hostname)
  - « Pseudo VM »



Pods - K8S

- Manipulation CLI (à la docker CLI)
  - kubectl run <pod_name> **--image** <image_name> [**--** <cmd>]
  - kubectl exec -it <pod_name> **--** sh
  - kubectl **delete** pod <pod_name> (~ rm)
  - kubectl **describe** pod <pod_name> (~ inspect)
  - kubectl **get pods** <pod_name> **-o jsonpath='{…}'**

Pods - K8S

- Manipulation via à l'IaC k8S
  - En règle générale, on manipule les **ressources k8s** par le biais de fichiers YAML appelés **manifestes**
  - k run <pod_name> --image <image_name> **--dry-run=client -o yaml** (> pod_name.yml)

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: <pod_name>
  name: <pod_name>
spec:
  containers:
  - image: <image_name>
    name: <ctn_name>
  restartPolicy: Always
```

création via manifeste : **k apply -f pod_name.yml**

champs réquis d'un manifeste :

- **apiVersion** : version de l'api k8s ou autre
- **kind** : type de ressource
- **metadata** : identification de l'objet (name, UID, namespace …)
- **specs** : état voulu pour la ressource

Pods - K8S

- Manipulation de pod multi container
  - Ex : logs : k **logs** <pod_name> **-c <ctn_name2>**
  - Ex : cp : k **cp** <pod_name>:<path/to/file> <path/to/file> **-c** <ctn_name>

```
...
spec:
  containers:
  - image: <image_name>
    name: <ctn_name>
  - image: <image_name2>
    name: <ctn_name2>
  restartPolicy: Always
```

Pods - K8S

- Manipulation de pod : attacher des labels
  - Les labels sont des paires clé / valeurs associées à nombreuses ressource k8s en particulier les pods

- k get pod **--show-labels**
- k get pod **-L <label_name>** (affichage des labels dans les listes de ressources)
- k get pod -L <label_name> **-l <label_name>** (filtrage par nom de label)
- k get pod -L <label_name> **-l <label_name>=<label_value>** (// valeur //)
- k **label pod <pod_name>** <label_name>=<label_value> (ajout d'un label dans une ressource)
- k label pod <pod_name> <label_name>=<label_value> **--overwrite=true** (écrasement de la valeur)
- k label pod <pod_name> <label_name>**-**

Pods - K8S

- Limites des pods
  - Pas de scalabilité
  - Pas de mise à jour et d'historisation des mises à jour
  - Pas de mise en réseau

- Limites à la manipulation directe des pods
  - On ne gère pas directement les pods, on les pilote via des objets dits de **« workload »** (cf infra)
  - De même on n'adresse jamais un pod par l'IP qui lui est affectée,

    - celle-ci étant aussi volatile que le pod

    k get pods <pod_name> -o jsonpath='{**.status.podIP**}'

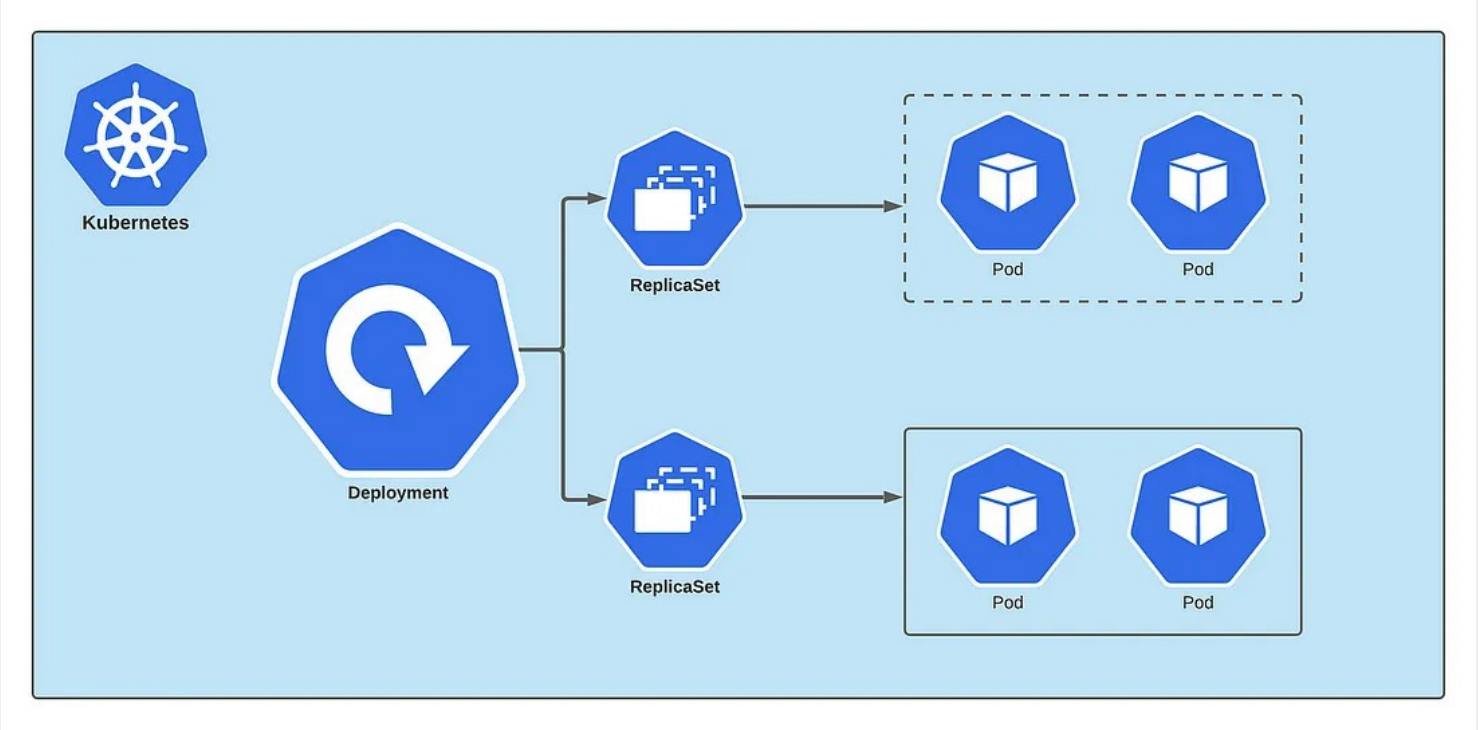
  - On fait appel pour cela à une ressource de type **« service »** (cf infra)
  - éventuellement complétée par une ressource de type **« ingress »** (cf infra)

WorkLoad - K8S

- Resources de + haut niveau enrichissant le pod
  - **ReplicaSet** : Pods + gestion de la **mise à jour** et la **réplication** (configurable)
  - **Deployment** : ReplicaSet + **mise en réseau**
  - **DaemonSet** : mise en réseau != ReplicaSet, **réplication fixe avec 1 replica / node**
  - **StateFulSet** : gestion des applications d'état pour volumes persistants (bdd / stockage)
  - **Job** : pod éphèmère, à usage unique **« one-shot »**
  - **CronJob** : Job exécuté de manière périodique par un **crontab**

WorkLoad - K8S

- Déploiement



WorkLoad K8S

- Deploiement : réplication & selection des pods à répliquer / déployer

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: <dpl_name>
  name: <dpl_name>
spec:
  replicas: 2
  selector:
    MatchLabels: # match conditions ET
      <pod_label_key>: <pod_label_value>
      <pod_label_key2> : <pod_label_value2>
  template: # pod description
    metadata:
      labels:
        <pod_label_key>: <pod_label_value>
    Spec:  # pod containers
```

Description de l'objet « Deployment »

k **get pods -l** app(!)=hello,k=v,… # « equaly based »

// -l **'**app **(not)in** (hello)**'** # « set based »

// -l app # exists « set based »

matchLabels

vs **matchExpressions**

- { key: <label_key>, operator: In, values: [<label_val>] }
- { key: <label_key2>, operator: Exists }

WorkLoad - K8S

- Deploiement : mise à jour
  - Maj du nb de réplicas : k **scale** deploy <dpl_name> --replicas <nb> **=> ne créé pas de REVISION**
  - Maj des propriétés des pods :

    ex image : k **set image** deploy <dpl_name> <pod_name>=<img:tag>
  - Maj directe via la config YAML : k **edit** deployment hello + <edit_file> + <save_file>

- Contrôler les mises à jour
  - État d'une Maj : k **rollout status** deploy <dpl_name>
  - Rollback : k rollout **undo** deploy <dpl_name> [**--to-revision=<revision_num>**]
  - Historique des révisions : k rollout **history** deploy <dpl_name>
  - Documenter une Maj : k **annotate** deploy <dpl_name> kubernetes.io/change-cause="<msg>" --overwrite=true

WorkLoad - K8S

- Interrompre et débloquer le mécanisme **kubectl rollout**
  - k rollout **pause** deploy <dpl_name>
  - À partir de cette commande,
    - les mises à jour ne sont plus effectives
    - le **undo**, **status** est désactivé
  - On peut gérer plusieurs changements successifs, vérifier la validité de l'état en pause
    - k apply -f <file.yml> | k set image … | …
    - et regarder le retour
    - les **révisions** ont été ajoutés normalement
    - mais l'état effectif n' pas changé dans le **describe**
  - Pour rendre les modification effectives : k rollout **resume** deploy <dpl_name>

WorkLoad - K8S

- Deploiement : Stratégie de Maj
  - **spec.strategy.type** :
    - **RollingUpdate** : (Default) ajouts & suppressions graduels en // des nouveaux / anciens pods
    - **Recreate** : toutes les suppressions puis tous les ajouts
  - **spec.strategy.maxSurge**: (25% Default)
    - nb max de pods (transitoires nouveaux + anciens) au dessus du nb désiré pendant la Maj
  - **spec.strategy.maxUnavailable**: (25%)
    - nb de pods indisponibles pendant la Maj
  - D'autres stratégies
    - « in-place » : k **patch** deploy --patch-file patch-file.yaml => (**spec.template.spec**)
      maj en place des pods (containers) sans contôle de l'env. (+ rapide, - sûre)
    - blue-green : Maj **progressive** ex : 20 => 50 => 80 => 100 % green (new) / 80 => 50 => 20 => 0 % legacy
      entre **deux déploiements** via un **Load Balancer** externe
    - canary : Idem en **ciblant le public** voyant la nouvelle version via des **routeurs**
    - ….

WorkLoad - K8S

- Ordre de déploiement: Les **InitContainers**
  - se lancent avant les containers "applicatifs"
  - Rôles :

    - ajouter des utilitaires sans modifier les dockerfiles des conteneurs applicatifs du Pod

    - préparer ou retarder l'exécution des conteneurs applicatifs tant qu'on ait vérifié certaines conditions

    - un **initContainer régulier est éphèmère**, il se stoppe quand le(s) conteneur(s) sont en exécution

    - en ajoutant un **restartPolicy: Always**, on l'appelle alors **sideCar container** pour supporter les conteneurs applicatifs en permanence

```
...
spec:
  initcontainers:
    - image: <init_image_name>
      name: <inti_ctn_name>
      restartPolicy: Always
      command : …
  containers:
```

WorkLoad - K8S

- StatefulSet
  - Contrairement aux Deployments, les StatefulSets garantissent **un ordre** de
    - déploiement,
    - mise à l'échelle
    - suppression des Pods
  - ainsi qu'un nom d'hôte stable et un stockage persistant (cf PersistentVolume).
  - Chaque pod StatefulSet a un hostname de type { pod_name**-i** } avec i ≥ 0 dans l'ordre des pods
  - Finalité : on utilise ces objets pour déployer des applications qui permettent **de sauvegarder les données écrites dedans**

    => ce faisant modifiant l'état de l'application => **Stateful** => bases de données, stockage ,...

    != **Stateless** => server web, logique serveur (php, java, ...)

WorkLoad - K8S

- Exposer des ports pour communiquer
  - Par défaut, l'exposition des ports est gérée dans les images de container
    - Ex : EXPOSE dans un Dockerfile

    => on peut faire remonter ou faire partager cette information au niveau De k8s
  - Ex :

```
containers:
  - …
    ports:
      - name: http
        containerPort: 80
```

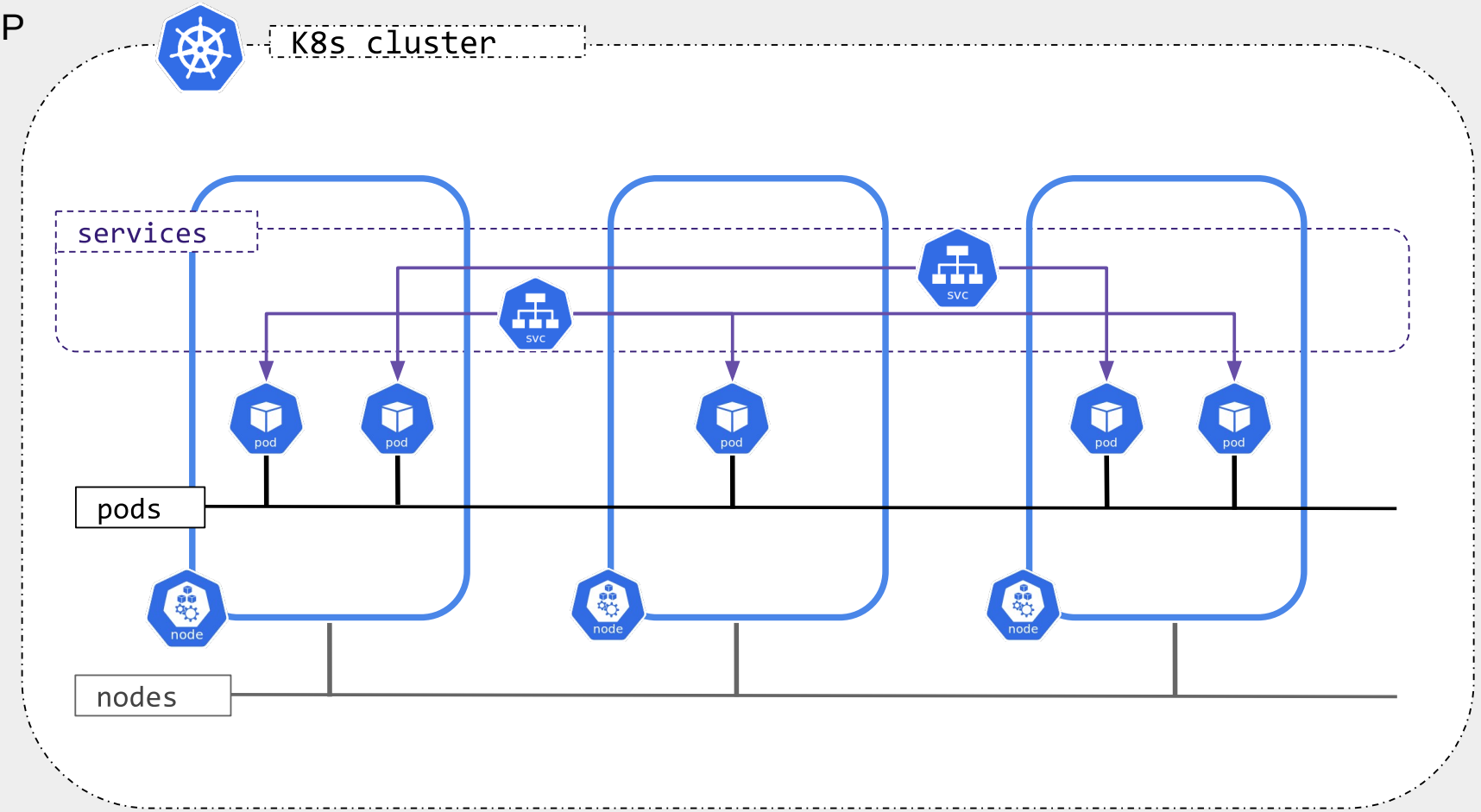
Services - K8S

- Mise en réseau des déploiements
  - **== exposition** i.e ajouter une ressource de type **service** sur un workload => trois types d'exposition :
  - **ClusterIP**: association d'une IP à un nom. L'IP n'est utilisable qu'à l'intérieur du cluster.
  - **NodePort** : association d'un port dans - l'intervalle [30000-32767] - aux IP de chacun des nodes. Cela permet ainsi des communications depuis l'extérieur
  - **LoadBalancer** : Associe une IP à un déploiement, IP permettant les accès externes en s'affranchissant -- du moins en apparence -- du NodePort.

```
k expose deploy <dpl_name> \
   --port <service_port> \
   --target-port <pod_port> \
   --type= ClusterIp | NodePort | LoadBalancer
   --dry-run=client -o yaml > <dpl_name>-svc.yml
```

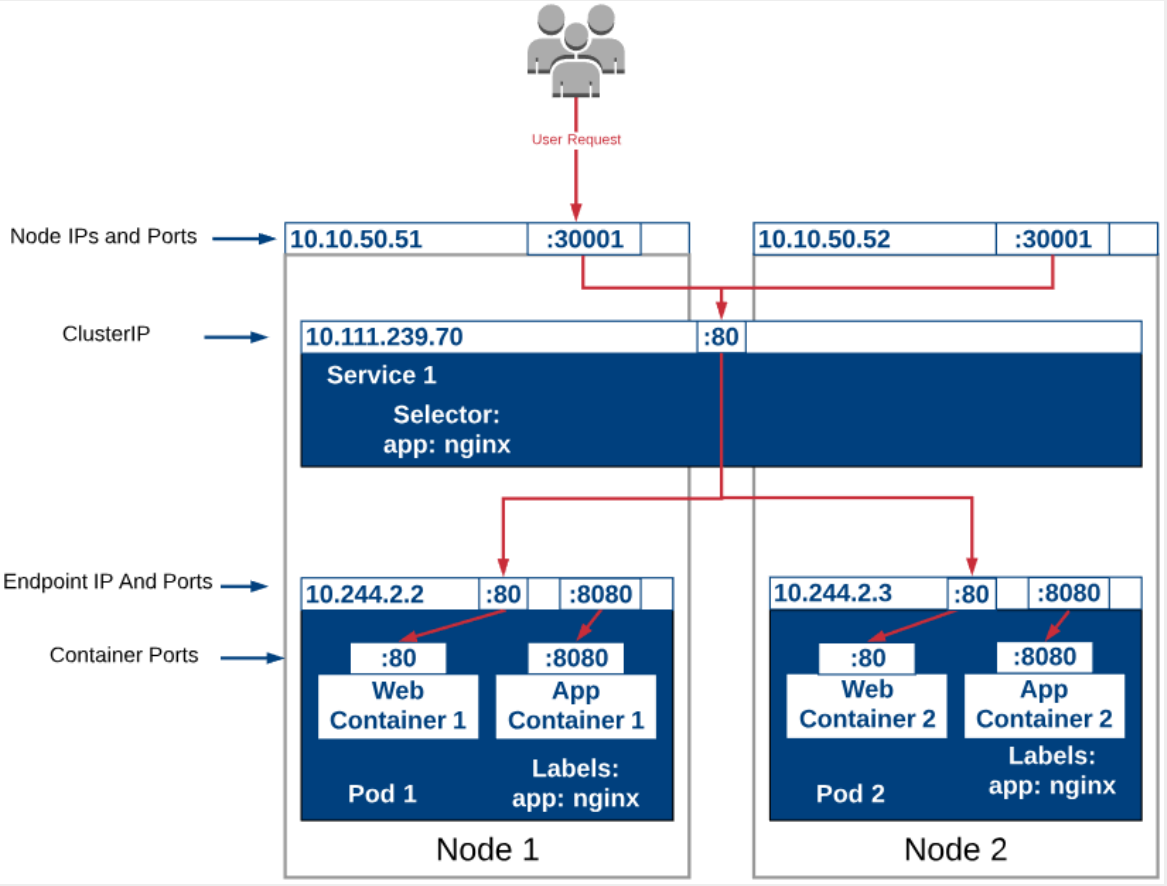
Services - K8S

- ClusterIP



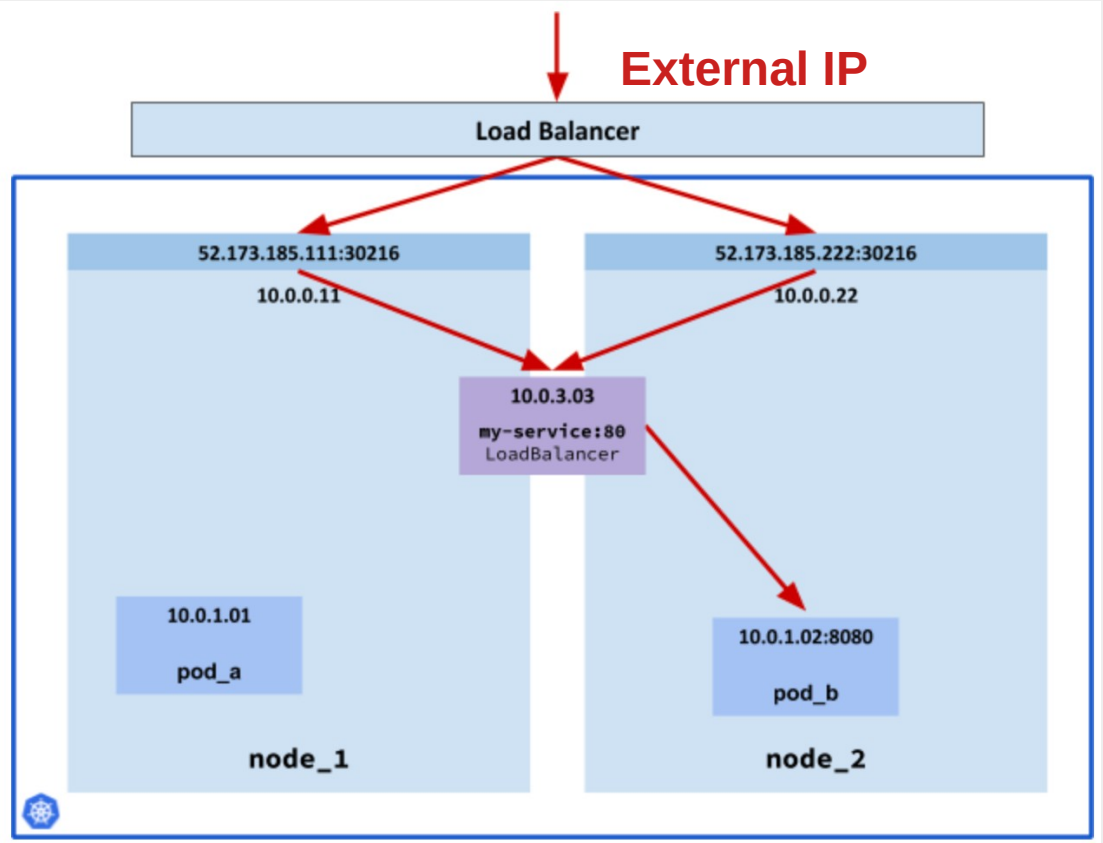
Services - K8S

- NodePort



Services - K8S

- LoadBalancer



Services - K8S

- Noms de domaines générés dans k8s:
  - Nom d'hôtes « Fully Qualified Domain Name » **FQDN**
  - Le cluster : par défaut **cluster.local**
  - Un Service : **<service_name>.<namespace_name>.svc.**cluster.local
  - Un Pod : (IP) **<xxx-yyy-zzz-ttt>**.<namespace_name>**.pod.**cluster.local

Services - K8S

- StatefulSet : « headless »service
  - Un service spécifiant la clé **spec.clusterIP: None**

    permet de créer des enregistrements DNS individuels pour chaque Pod,

    => permettant ainsi aux clients d'accéder **directement** aux Pods en utilisant **leur nom d'hôte**.
  - REM : On les voie avec « **None** » dans « **CLUSTER-IP** » lorsque l'on fait un « **k get svc »**
  - Pod (StatefulSet + headless) : **<pod_name>-i**.<service_name>.<namespace_name>.svc.cluster.local
  - La clé **spec.serviceName** du statefulSet permet de donner un dns court pour le pod

    <pod_name>-i.<service_name>

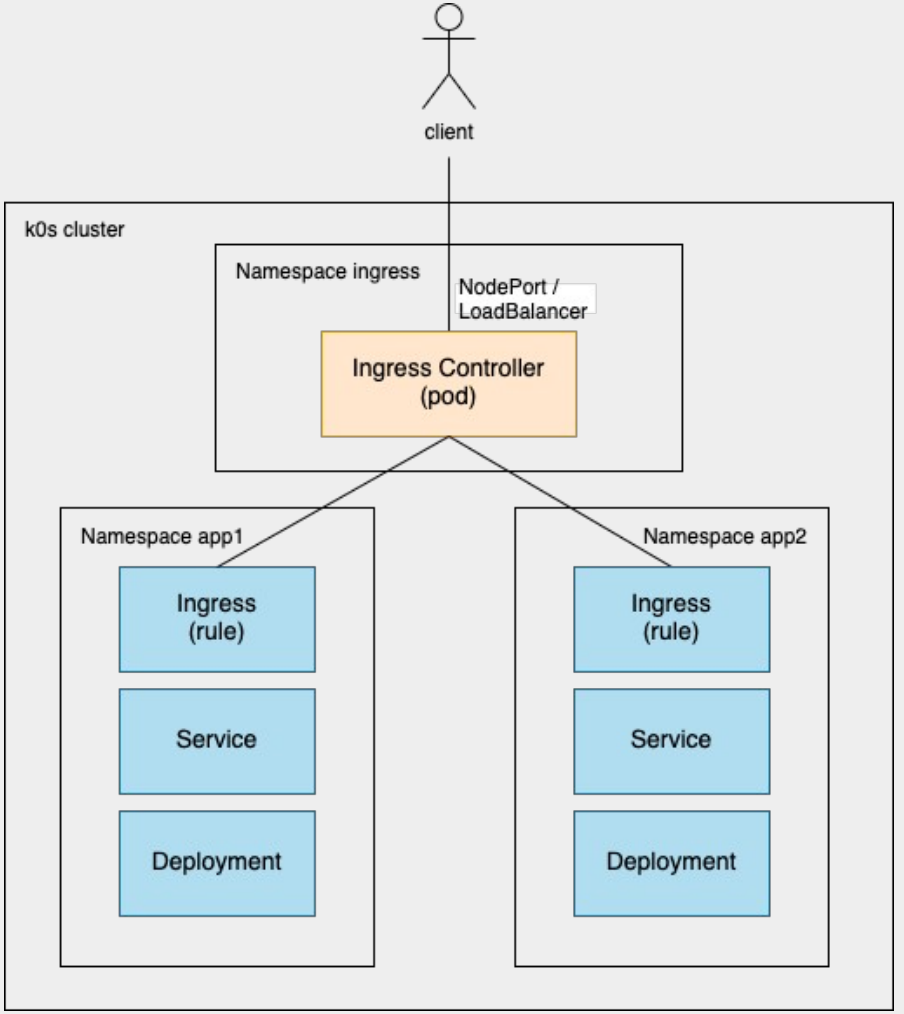
Services - K8S

- Ingress Controller
  - Un contrôleur d' « ingress » i.e de flux entrant est une **ressource externe** de k8s mais compatible
    - éditeurs indépendants : nginx, traefik, Haproxy
    - fournisseurs de cloud GCE (Google Compute Engine), aws-load-balancer-controller
  - Il a plusieurs fonctions dans une infra k8s
    - **exposer le trafic http et https** depuis l'extérieur dans des services du cluster
    - contrôler la **sécurité TLS** du trafic de bout en bout
    - **répartir la charge** sur les réplicas des sercives
    - gérer des noms d'hôtes virtuels public pour accéder aux services (**virtualHost**)
  - L'ingress est une alternative favorite **- pour le trafic http et https**, aux services de type
    - nodePorts
    - loadBalancer

Services - K8S

- Ingress Controller

  - La ressource k8S « ingress controller » est
    - associé à un **dpl** via un **svc clusterIP**
    - il faut aussi une ressource k8s LoadBalancer
  - La ressource k8s « ingress » gère
    - les règles du trafic entrant
    - nom d'hôte / chemin / protocol etc.



Persistance - K8S

- Accès & Persistance des données : **volumes**
  - Les pods K8s sont tout autant **éphèmères** que les containers dockers
  - Il est souvent nécessaire qu'un conteneur puisse accéder à des données :

    - de configuration → ini, json, xml, yaml, ...

    - d'exploitation → html, php, png, sql...

    - brutes → csv, json, xml, yaml...

  - Types de volumes :

    - **emptyDir** : point de montage créé en tant que dossier vide, permet la communication inter container dans un pod, détruit avec le pod

    - **hostPath** : point de montage créé entre le hôte (i.e les nœuds) et les pods

    - nfs

    - **configMap** : point de montage créé entre une ressource ConfigMap liée à un fichier de config et un pod

    - ... **ici**

Persistance - K8S

- Forme générale d'utilisation de volumes
  - 2 clés : spec.template.spec.volumes ET spec.template.spec.containers[].volumeMounts

```
volumes:
  - name: data_ed
    emptyDir:

  - name: data_hp
    hostPath:
      path: /data  # on nodes
      type: Directory

  - name: data_nfs
    nfs: # config nfs
      server: xxx.yyy.zzz.ttt
      path: /
```

```
volumeMounts:
  - name: data_ed
     mountPath: /data # on container

  - name: data_hp
    mountPath: /data # on container
    subPath: subdir # from source


  - name: data_nfs
    mountPath: /data # on container
    subPath: subdir # from source
```

Persistance - K8S

- Limitations du type hostPath
  - Le dossier associé doit exister **sur tous les noeuds** susceptibles d'héberger les pods.
  - On est dépendant du système de fichiers des nodes
  - Tant qu'on reste avec des accès en lecture, il n'y pas de problème de concurrence entre plusieurs conteneurs sur un même node.
  - En cas d'accès en écriture, peuvent se poser des problèmes d'accès concurrents
    - nécessité d'un système de fichiers gérant ces accès
    - utiliser une infra de réplication primaire/secondaire
    - utiliser une infra multi-maître -> indépendance des volumes

Persistance - K8S

- Les Volumes Persistants
  - La ressource k8s « **PersistentVolume** » représente
    - la configuration d'un volume tel que renseigné dans un objet Workload
  - Mais en ajoutant plusieurs propriétés inhérentes aux volumes tels que
    - la **capacité** (~ la taille) des données
    - le mode de stockage : **FileSystem** | **Block** | **Object**
    - le mode d'accès : Read [**Only** | **Write**] [**Once** | **Many**]

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: <pv_name>
spec:
  capacity:
    storage: 10Mi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
```

Once : le montage n'est monté que sur un seul node

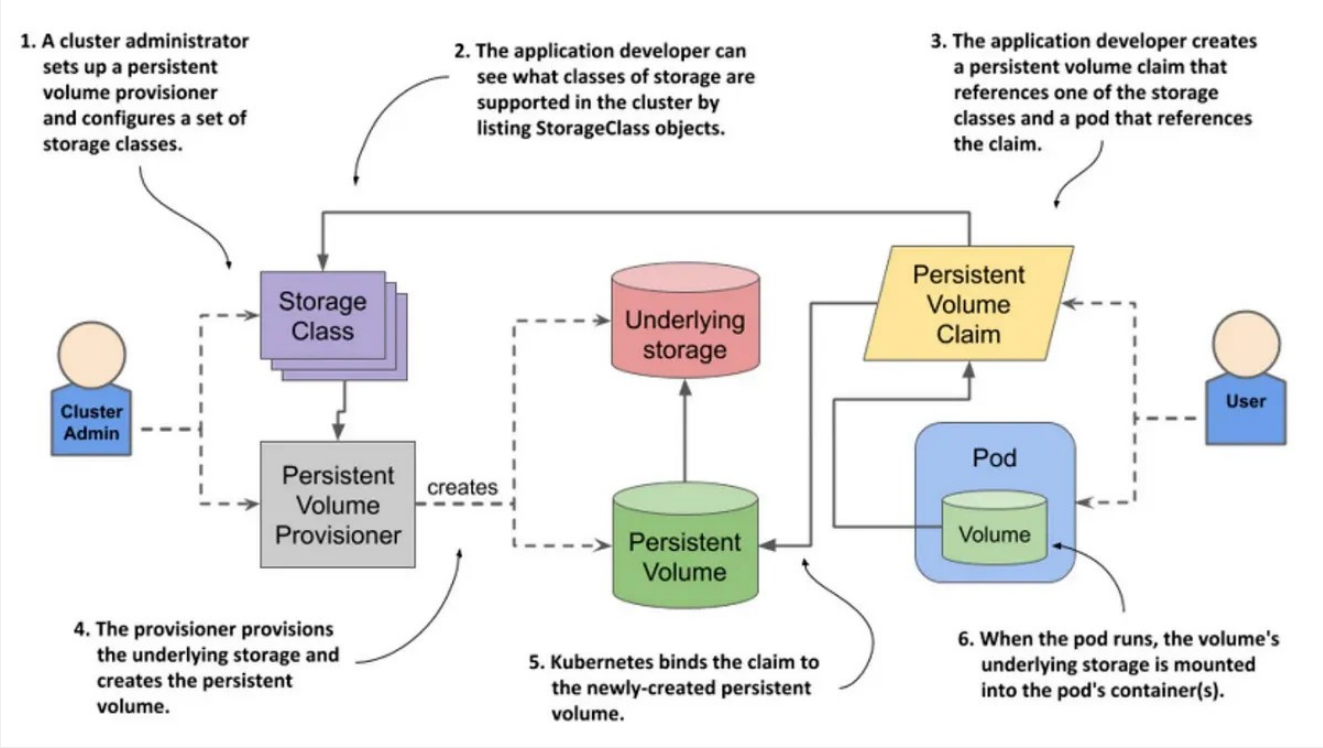
OncePod : le montage n'est monté que sur un seul pod

Persistance - K8S

- Accès dynamique aux Volumes Persistants :
  - La ressource k8s « **PersistentVolumeClaim** » représente une requête d'accès à un PV
  - Selon de nombreux critères **manuels** dont
    - selecteurs / labels
    - capacity min., Accès de modes, modes de stockage
  - La classe de stockage : objet **générique** renseignant les mêmes critères
  - La **politique de « Reclaim »** : quoi faire du PV quand un PVC lié est supprimé
    - Retain : suppression manuelle par l'administration
    - Delete : suppression immédiate avec le PVC

Persistance - K8S

- Les classes de stockage
  - Configuration génériques : permettant de manipuler des PV en fonctions de critères



Persistance - K8S

- ConfigMap: création
  - permet de propager un contenu sur l'ensemble du cluster, de manière plus naturelle qu'un volume.
  - Contenus gérés : **fichiers**, paires **clé / valeur**

```
k create configmap <cm_name> \
--from-file <file_path> \

--from-literal KEY=VAL \

--from-env-file <key_val_file_path> \

--dry-run=client -o yaml > cm_name.yml
```

```
volumes:
  - name: <config_vol>
    configMap:
      name: <cm_name>
```

Persistance - K8S

- ConfigMap: Usage pour arrimer les fichiers de configuration

```
containers :
  - ...
    volumeMounts :
      - name: <config_vol>
        mountPath: <mount_path_dir>


volumes:
  - name: <config_vol>
    configMap:
      name: <cm_name>
```

Environnement - K8S

- Gestion des variables d'environnements
  - Ajout direct de variables d'env. dans les conteneurs des pods

```
containers:
  - ...
    env:
      - name: <KEY>
        value: <value>
```

  - Chargement des variables depuis un configMap

```
containers:
  - ...
    envFrom:
      - configMapRef:
          name: <cm_name>
        prefix: PRFX_
```

OU

seulement des valeurs

```
env:
  - name: <KEY>
    valueFrom:
      configMapKeyRef:
        name: <cm_name>
        key: <CM_KEY>
```

Environnement - K8S

- Gestion des secrets
  - Ressource similaire à une configMap mais dédiée aux **données sensibles**.
    - 3 sous commandes : « **generic** », « docker-registry » et « tls »
    - nombreux types de secrets « **Opaque** » par défaut, « basic auth », « ssh auth », « tls auth », …

```
k create secret generic <secret_name> \
--from-literal SECRET=azerty \
--dry-run=client -o yaml > <secret_name>.yml
```

  - **WARN !!**: La valeur cachée n'est qu'**encodée en base64 non chiffrée** dans l'objet Secret

Environnement - K8S

- Usage des secrets
  - Chargement des secrets OU seulement des valeurs depuis un secret

```
containers:
  - ...
    envFrom:
      - secretRef:
          name: <secret_name>
```

```
env:
  - name: <KEY>
    valueFrom:
      secretKeyRef:
        name: <secret_name>
        key: SECRET
```

Environnement - K8S

- Injecter les métadonnées k8s dans l'application
  - Exemple

```
env:
  - name: NSPC
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

Contrôle de l'état - K8S

- Cycle de vie des pods : **k get pod <pod_name> -o yaml**
  - Phases d'un pod : **status.phase**
    - **Pending** : pod **accepté** par K8S donc pas de pb de syntaxe, mais un ou plus conteneurs **n'est pas démarré**
    - **Running** : le pod est associé à un nœud, les conteneurs sont en exécution
    - **Succeeded** : si les processus du pod (conteneurs) s'arrêtent sans erreur sans redémarrage (Job, cronJob)
    - **Failed** : un des processus du pod (conteneurs) s'arrête en erreur
    - **Unknown** : l'état ne peut pas être obtenu. Souvent lié à un pb de communication node / pod (kubelet)
  - États d'un conteneur dans k8s : **status.containerStatuses[].state**
    - **Running** : en exécution
    - **Terminated** : arrêté en succès ou en erreur
    - **Waiting** : ! Running || ! Terminated

Contrôle de l'état - K8S

- Cycle de vie des pods : **k get pod <pod_name> -o yaml**
  - conditions d'un pod : **status.conditions**
    - en particulier la notion de **« Ready »,** i.e prêt à accepter des connexions en entrée
  - Etats contrôlés d'un conteneur dans k8s :
    - **status.containerStatuses[].started** : démarrage du conteneur « **Startup** »
    - status.containerStatuses[]**.ready** : prêt à accepter des connexions en entrée « Ready »
  - Cycle d'un crash
    - Initial Crash : au moment du démarrage
    - Repeated crashes : k8s établit un **délai d'attente** (configurable cf infra.) entre crashes
    - CrashLoopBackOff state : politique de redémarrage continu avec une **boucle crash / redémarrage**
    - Backoff reset : si un état en succès restauré permettra la **désactivation du délai d'attente**

Contrôle de l’état - K8S

- Sonder les conteneurs des pods :
  - Évènements à sonder :
    - si le conteneur est démarré => **containers[].startupProbe**
    - si le conteneur accepte le trafic en entrée (services) => containers[].**readinessProbe**
    - si le conteneur tourne correctement sinon on redémarre => containers[].**livenessProbe,**
  - Délai d’attente
    - **probe.initialDelaySeconds** : délai après le startup
    - probe.**periodSeconds** : délai entre sondage
    - probe.**(success)failureThreshold** : nb de tentatives pour vérifier un crash ou un succès

      Délai total = (success)failureThreshold * periodSeconds
    - probe.**timeoutSeconds**

Contrôle de l'état - K8S

- Techniques de sondage :
  - Exécution d'une commande dans le conteneur : **probe.exec.command[]**
  - Exécution d'une requête HTTP sur un endpoint et un port : **probe.httpGet**
  - Exécution d'un socket TCP sur un port : **probe.tcpSocket.port**

  - **REM :** le choix de la technique et de la commande / URL / port doivent être **judicieux sinon on peut mettre un pod en erreur anormalement !!**

Partitions - K8S

- Les espaces de noms :
  - Un **namespace** est une forme de compartiment/cloisonnement

    dans lequel on ne peut avoir deux ressources de même type de même nom
  - Finalité :

    - **Isolation** des ressources
    - **Autorisation** d'accès
    - **Découpage** du cluster en sous-cluster.

  - Par défaut, la communication entre namespaces est possible avec les **noms FQDN**

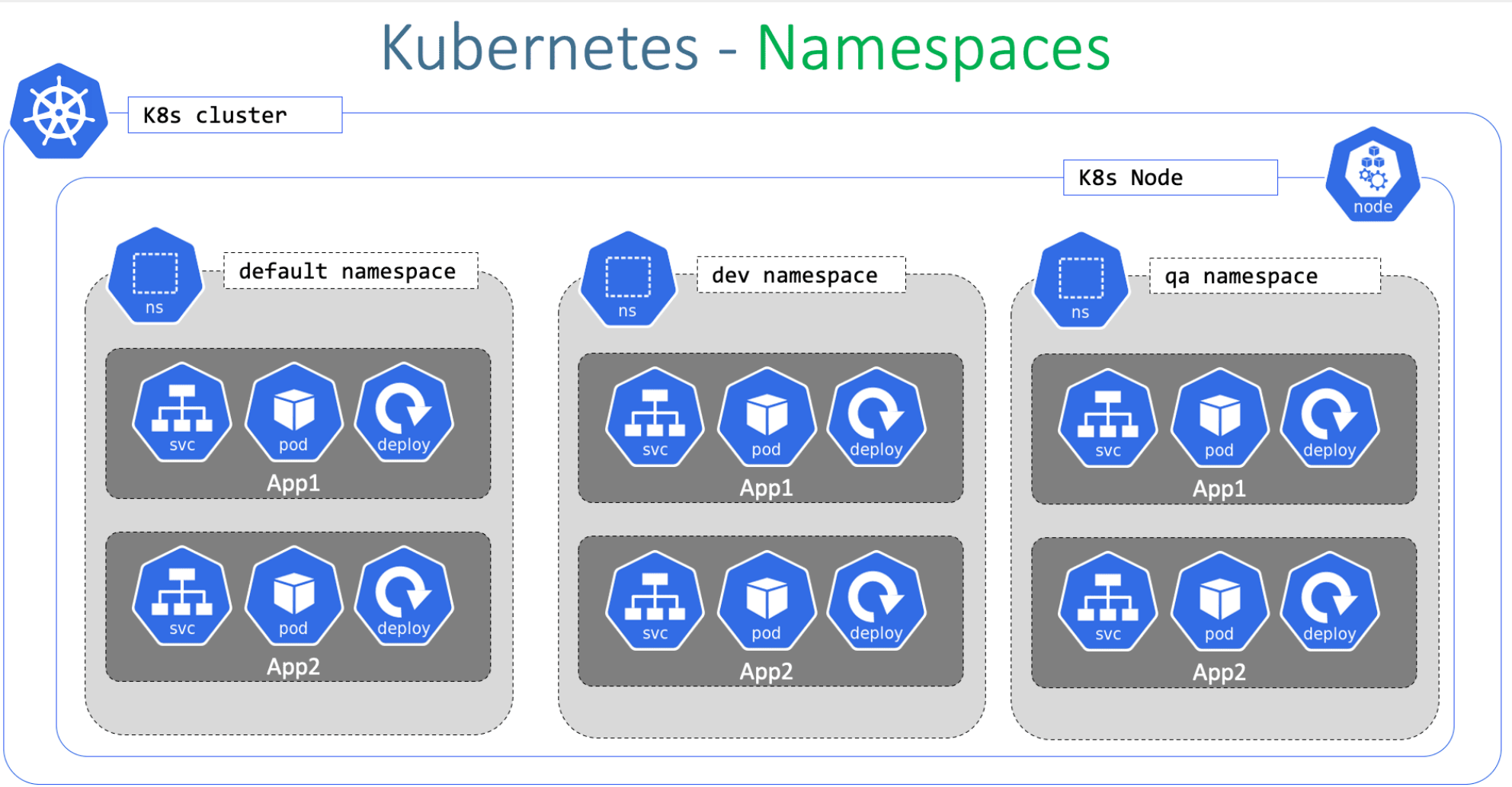
    - ces communications peuvent être restreintes par le moyen de politiques de réseau **« network policies »**

Partitions - K8S

- Gérer / Affecter un espace de nom:
  - Gestion (Ex : pod)
    - k create **namespace <ns_name>**
    - k get ns
    - k get pod **-n <ns_name>** [--create-namespace (si le ns n'existe pas)]
    - k get pod **-A** (pour tous les namespaces)
  - Affectation (Ex déploiement)

```
...
metadata:
  ...
  namespace: <ns_name>
# OU
k apply -f <file.yml> -n <ns_name>
```

Partitions - K8S



Partitions - K8S

- Politiques de réseau:
  - Les « netpols » permettent de définir des **autorisations** pour un ensemble de pods.
  - Par défaut, toutes les communications sont permises
  - Quand une **autorisation particulière** pour un **namespace** et / ou une **sélection de pods** est définie,

    => toutes les autres communications sont **interdites**

    => dans les 2 sens du point de vue d'un pod, en entrée **« Ingress »** et / ou en sortie **« Egress »**

    => dans un scope donné, **namespace** et / ou des **sélection de pods** et / ou des **blocs de IPS**

  - **REM :** SI les netpols sont des ressources par défaut du cluster, leur implémentation dépend du CNI

    => Ce n'est pas le cas de tous les CNI :

    - Calico (OK) | Clilium (OK) | Canal (OK) | Flannel (NO) | Kindnet (NO)

Partitions - K8S

- netpols remarquables:

```
# deny-all
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  PodSelector: {}
  policyTypes:
  - Egress
  - Ingress

# deny-egress
…
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

```
# allow-all-ingress
….
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Composition - K8S

- Kustomize :
  - Outil de
    - **centralisation**
    - **standardisation**
    - **customisation** des ressources IaC k8s formant des microservices
  - Ressources habituelles
    - workload (Déploiement, StatefulSet, ...)
    - services
    - configMap, secret
    - volumes persistents
    - ...

Composition - K8S

- Kustomize : configuration de base

```
# fichier kustomization.yml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace : <nspc>
commonLabels:
  run: hello
resources:
  - app-deploy.yml
  - app-svc.yml
  - app-cm.yml
  - app-secret.yml
  - …

# execution dans le dossier comprenant le fichier  kustomization.yml et les ressources
k create ns <nspc> # kustomize ne créé pas un namespace
k kustomize  # vérification
k apply -k .          #    k get -k .  |   k describe -k . | k delete -k .
```

Composition - K8S

- Kustomize : génération de configuration

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
configMapGenerator:
  - name: <prefix>-cm-<suffix>
    envs:
      - .env
      - file.properties
      - file.ini
  - name: <prefix>-cm-<suffix2>
    literals:
      - FOO=Bar
  - name: <prefix>-cm-<suffix3>
    files:
      - file.conf
      - file.txt
# optionnel
generatorOptions:
  disableNameSuffixHash: true
```

```
secretGenerator:
  - name: <prefix>-secret-<suffix>
    envs :
      - secret.env
```

<suffix> : généré par kustomize par défaut

fixer les suffixes avec

```
generatorOptions:
  disableNameSuffixHash: true
```

Composition - K8S

- Kustomize : customisation à partir d'une base

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - ../app_base
namespace: <new_ns>
namePrefix: <new_prfx>-
replicas:
  - name: <workload_name>
    count: 2
images:
  - name: <current_image>
    newName: <new_image>
    newTag: <new_tag>
configMapGenerator:
  - name: <current_cm>
    behavior: replace | merge
    envs:
      - <new_data>.env
```

Gestion des ressources - K8S

- Serveur de métriques
  - Si les kubelet récupèrent un certains nombre d'informations de **ressources**,
  - celles-ci ne sont par défaut pas disponibles
  - sauf à installer un **serveur de métriques**
  - Par Ex **metrics-server**

```
+ k apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
+ k patch -n kube-system deployment metrics-server --type=json \
  -p '[{"op":"add","path":"/spec/template/spec/containers/0/args/-","value":"--kubelet-insecure-tls"}]'
```

  - k **top nodes**
  - k **top pods** [<pod_name>] [-n <nspc> | -A]

Gestion des ressources - K8S

- Réguler directement l'accès aux ressources
  - 2 types accès : les **limites** (supérieures) et les **requêtes** (valeur min garanties)
  - Pour les types de ressources classiques : **cpu**, **memory**, storage (cf PersistentVolumes)
  - Pour le trafic résau d'un pod on utilise l'annotation **kubernetes.io/ingress-bandwidth**: 10M

  - k set **resources** deployment <dpl_name> [-c <ctn_name>] **--limits | --requests** cpu=200m

    // pod <pod_name> // // memory=500Mi

Gestion des ressources - K8S

- Réguler les ressources d'un namespace
  - La description d'un namespace indique l'usage de 2 « ressources k8s » permettent de gérer les ressources

    k describe ns <nspc>

  - Les **ResourceQuotas** spécifient les métriques de ressources globales à un namespace donné

  - Les pods d'un namespace doté d'un ResourceQuota doit spécifier toutes les ressources du RQ dans sa config

    => sinon les **pods non pourvus** ne seront **pas ordonnancés**
  - Les **LimitRanges** permettent de **définir des intervalles de valeurs** pour des ressources **par défaut**

Gestion des ressources - K8S

- ResourceQuota : exemple

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota
  namespace: <nspc>
spec:
  hard:
    pods: 10 # nb max / nspc
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "500m" # ou 0.5
    limits.memory: 2G
```

unité cpu entière => calcul CPU sur x CPU phys. ou virtuel

unité memory Xi => en puissance de 2 ex : Ki = 1024 octets

unité cpu float ou « m » => proportion du temps CPU total

unité memory X => en puissance de 10 ex : K = 1000 octets

Gestion des ressources - K8S

- LimitRange : règles

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limitrange
  namespace: <nspc>
spec:
  limits:
  - default:
      cpu: 500m
    defaultRequest:
      cpu: 300m
    max:
      cpu: 800m
    min:
      cpu: 200m
    type: Container | Pod |
PersistentVolumeClaim
```

ex CPU :

1/ default & defaultRequest désigne les **limites** et **requêtes** pour un type de « ressource k8s » **non pourvu, par défaut**

2/ max & min : permet de vérifier si les ressources spécifiées pour un type de « ressource k8s » sont

≥ min && ≤ max,

Sinon la « ressource k8s » n'est pas ordonnancée

3/ si les valeurs default & defaultRequest sont absentes,

Elles seront évaluée **par défaut à max**

Gestion des ressources - K8S

- Réguler les ressources au niveau d'un node
  - L'agent **kubelet**, est responsable de
    - l'enregistrement d'un serveur comme nœud dans un cluster k8s
    - la gestion du cycle de pod / workloads à partir des consignes configurées dans la base etcd du control pane (pour les services voir kube-proxy)
  - Sa configuration spécifie plusieurs options liées aux ressources
    - dans le fichier **/var/lib/kubelet/config.yaml** dans les noeuds
    - **podPidsLimit: x** (-1 par défaut), le nb max de processus à exécuter dans un pod
    - cette métrique permet de prévenir une attaque de type **« fork bomb »**
    - ne pas oublier : **systemctl restart kubelet**

Gestion des ressources - K8S

- Priorité des pods
  - La ressource k8s PriorityClass permet

    - d'affecter un **ordre de priorité** dans l'ordonnancement des pods, en tant que **32-bit entier < 1000000000**
    - de **préempter** ou non les ressources nécessaires à l'exécution des pods

      => s'il y a **préemption**, des pods de **priorité plus faible seront arrêtés**

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-nonpreempting
value: 1000000
preemptionPolicy: Never ( ou pas de clé )
globalDefault: false
description: "blabla"
```

1/ une seule classe de priorité peut avoir l'attribut **globalDefault**

Pour affecter sa valeur aux pods sans priorité spécifiée

2/ on attribue une classe de priorité à un pod avec la clé

spec.**priorityClassName** pour (Pod)

spec.template.spec.**priorityClassName** (Deployment)

- Voir les priorités avec k get pods --sort-by='**.status.priority**' -o wide

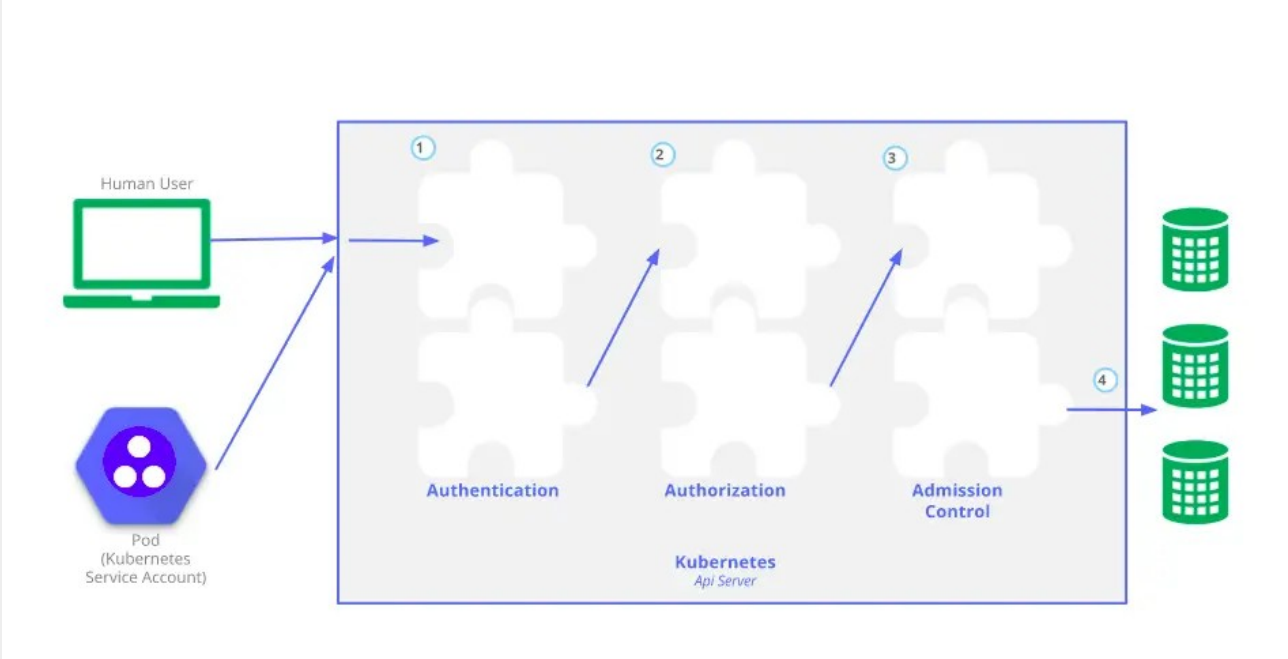
Gestion des ressources - K8S

- Mise à l'échelle pilotée par les resources
  - La resource **HorizontalPodAutoscaling**
  - Définir le nb de replicas idoine d'un workload lié à des ressources définies

```
k autoscale deployment <dpl_name> \
--cpu-percent xx \
--min i \ # min replicas
--max j \ # min replicas
--dry-run=client -o yaml > <dpl_name>-hpa.yml
```

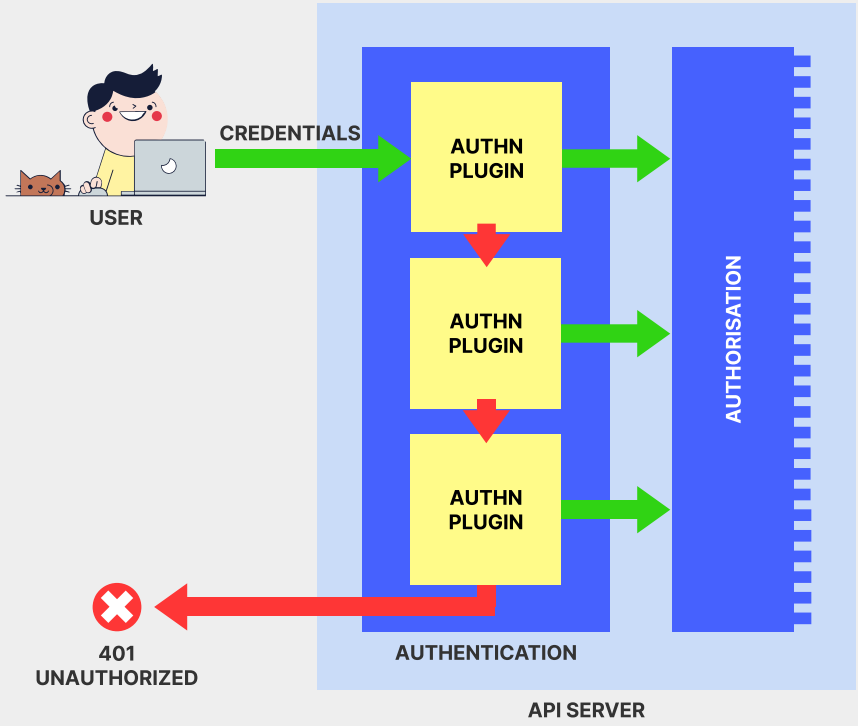
Accès- K8S

- Différents d’accès au cluster
  - Comptes « utilisateurs » liés à des **identifiants**, et des **contextes**
  - Comptes de services « **ServiceAccounts** » liés à un **jeton JWT** - par défaut, et des **namespaces**



Accès - K8S

- Modes d'authentification « **Plugins** »
  - Gérés par k8s par défaut
    - Jeton « **Bearer: <hash>** » dans le header **HTTP Authorization** (Oauth 2.0)
    - Certificats **x.509**
    - OIDC (OpenID Connect)
  - Utilisables ...
    - **LDAP**, AD, Kerberos, ...
  - via des **extensions**
    - WebHook Token, Authenticating Proxy
  - Si un plugin d'auth fonctionne avec le certif => OK !



Accès- K8S

- Configuration du client kubectl
  - Fichier **~/.kube/config** ou k **config view**

    - liste des **clusters** cibles de kubectl : nom, accès au serveur d'API k8s, authentification
    - liste des « **utlisateurs** » : enregistrements associés à une cnx TLS (certif + pKey)
    - liste des **contextes** : collections cohérentes d'informations pour l'accès :
      + cluster
      + utilisateur
      + namespace
    - quel contexte courant est utilisé
    - préférences …

  - La variable d'environnemet **KUBECONFIG** peut être créée pour changer l'remplacement du fichier de configuration de kubectl

Accès - K8S

- Créer un « compte utilisateur »
  - Générer un certificat X.509 dûment signé avec **openssl**
    - créer une clé privé symétrique « <user>**.key** »
    - en déduire une requête de signature de certificat « <user>**.csr** »
    -le sujet « **CN=<user>/O=<group>** » définit l'utilisateur et le groupe liés aux **autorisations RBAC** (cf infra.)
    - en déduire une ressource k8s **CertificateSigningRequest** **ici** (le csr doit être encodé en base64)
    - approuver la ressource k8s CSR : **k certificate approve <csr_name>**
    - rechercher le certificat TLS « **.crt** » dans le CSR : k get csr <csr_name> -o jsonpath=**'{...}'**
      (le crt doit être décodé en base64)
  - En déduire un utilisateur au sens kubectl et un context
    - k config **set-credentials <user>** --client-certificate <user>.crt --client-key <user>.key
    - k config **set-context** <ctx_name> --cluster <cluster_name> **--user=<user>**
  - Changer de contexte
    - k config get-contexts ; k config **use-context** <ctx_name>

Accès - K8S

- ServiceAccount :
  - Accès privilégié au cluster (l'API Serveur) pour les procédures automatisées depuis
    - un pipeline **CI / CD** => Ex : déploiement
    - un outil **cloud** => Ex : mise en échelle, provisionnement de stockage
    - depuis un **pod** => communications authentifiées / chiffrées entre pods
  - Un compte de service par défaut nommé « **default** »
    - est créé avec tout nouveau **namespace**
    - est utilisé par défaut à tout **pod** de ce namespace
    - mais n'autorise **aucun accès** aux enpoints de l'API Serveur !!!
    - pour ajouter des permissions, on ajoute un nouveau objet « serviceAccount » custom

Accès - K8S

- Créer un ServiceAccount :

```
k create sa <obj>-sa \
--namespace <nspc> \
--dry-run=client -o yaml > obj-sa.yml
```

  - Pour ajouter des permission au serviceAccount

    - créer ou utiliser des **rôles k8s**, puis **lier** des rôles au serviceAccount (cf infra.)

    - puis renseigner la clé **spec.template.spec.serviceAccountName: <obj>-sa** dans l'objet workload cible

Accès - K8S

- Configurer des ressources RBAC
  - **R**ole **B**ased **A**ccess **C**ontrol
  - On va associer

    - des ressources k8s « **Roles** »
    - à des comptes « utilisateurs / groupes » ou des ServicesAccounts
    - grâce à des ressources k8s « **RoleBindings** »
  - Les Roles / RoleBindings sont liés à un **namespace**
  - Les **Cluster**Roles / **Cluster**RoleBindings ne sont pas //

Accès - K8S

- Structure d'un (Cluster)Rôle

```
k create role <role_name> \
--resource <rsc> \
--verb=get,list,create,patch,update,delete,deletecollection,watch
--dry-run=client -o yaml > <role_name>-role.yml
```

- Ressources k8s : pod, deployment, services, namespaces, ….
- Le Verbe **watch** : voir les mises à jour des ressources associées en temps réel
  - induit le verb list => kubectl get <rsc>
  - induit le verb get => kubectl get <rsc> <rsc_obj>
  - => kubectl get <rsc> [<rsc_obj>] **-w**

Accès - K8S

- Structure d'un (Cluster)RoleBinding

```
k create rolebinding <rb_name> \
--role <role_name> \
[ --group <group> ] \
[ --user <user> ] \
[ --serviceaccount <nspc:sa_name> ] \
[ --namespace <nspc> ] \
--dry-run=client -o yaml > <rb_name>-role-binding.yml
```

- **<user>** référence l'utilisateur renseigné dans les contextes => dans la section **/CN=** du sujet du **CSR**
- **<group>** référence le group renseigné dans la section **/O=** sujet du CSR
- **<nspc:sa_name>** désigne un serviceAccount lié à son namespace natif
- **<nspc>** n'est pas spécifié s'il s'agît d'un ClusterRoleBinding