Chapitre 6 : Approfondissement Kubernetes

Docker et Kubernetes

CCI Strasbourg

Chapitre 6 : Approfondissement Kubernetes

Objectifs

En suivant ce chapitre, nous allons appronfondir Kubernetes avec les modules suivants :

- 6.1 Réseau dans Kubernetes : Concepts et solutions CNI
- 6.2 Stockage persistant avec Persistent Volumes et Persistent Volume Claims
- 6.3 Auto-scaling et gestion des ressources
- 6.4 Sécurisation des clusters avec les politiques de réseau et RBAC

Chapitre 6 : Approfondissement Kubernetes

Module 6.1
Réseau dans Kubernetes Concepts et solutions CNI

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Réseau dans Kubernetes Concepts et solutions CNI

Sommaire :

- Les concepts réseaux dans Kubernetes
- Les spécifications CNI
- Les plugins (solutions CNI)
- Ingress Controller
- Mise en place Nginx Ingress Controller (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Réseau dans Kubernetes Concepts et solutions CNI

Les concepts réseaux dans Kubernetes

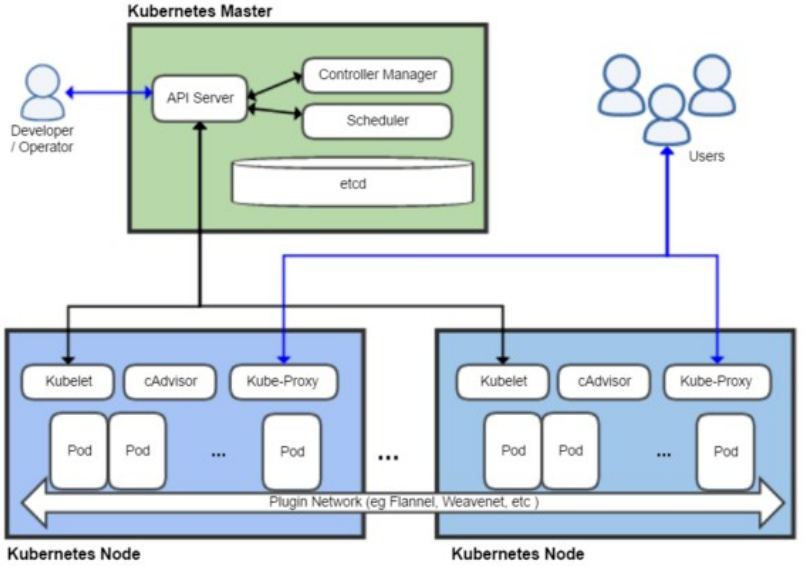
Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Rappels et généralités : architecture

Un cluster Kubernetes est composé d'un master et de plusieurs nœuds (exception pour les clusters de type test ou l'ensemble peut se trouver sur une seule machine).

- La communication entre ces machines est effectuée à travers :
  - L'API Server et les agents Kubelet
  - La couche réseau est **en partie** gérée par le comp **Kube-proxy** qui permet de :
    - Assurer la connectivité dans un cluster K8s
    - Gérer les règles réseaux
    - Permettre l'accès aux services (dans le sens K8s) exposés



Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Rappels et généralités : réseau Kubernetes

Quelques rappels :

- Un Pod dispose d'une IP unique
- Au sein d'un pod, les ports sont partagés avec tous les conteneurs (= un port ne peut être utilisé qu'une fois)
- Par défaut, tous les Pods peuvent communiquer entre eux. Si on doit restreindre ou effectuer des règles plus poussés pour isoler les Pods, il faudra le faire via des NetworkPolicies.
- Chaque nœud dispose d'un range d'IP qui est configurable :
  - IPAM va délivrer des adresses Ips aux Pods
  - Des adresses IP sont réservés pour les services

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Rappels et généralités : réseau et conteneur

Un conteneur n'est pas un OS, mais un ensemble de processus isolés via des namespaces et limité par les cgroups (limite les ressources CPU, mémoire,etc..).

Au niveau réseau, un conteneur n'a pas de carte réseau physique, la définition de son réseau va donc se réaliser à travers les namespaces linux.

Plusieurs namespaces sont disponibles pour la gestion des processus : pid, mnt, ipc, **net**, uts, uset, user, cgroup et time.

Pour la gestion du réseau c'est bien le namespace **net** qui est utilisé.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Rappels et généralités : réseau et conteneur

Donc pour définir un réseau sur un conteneur, il faut définir de nombreux éléments et voici un exemple pour connecter des conteneurs entre eux sur un « switch virtuel »

Heureusement, Docker (ou un autre moteur de conteneur) fait tout ça pour nous lorsque l'on précise la configuration du réseau.

Et dans Kubernetes ? C'est une des missions du CNI (**Container Network Interface**) que l'on verra plus tard !!

1/ Création net namespaces des conteneurs
ip netns add c1
ip netns add c2
ip netns add c3

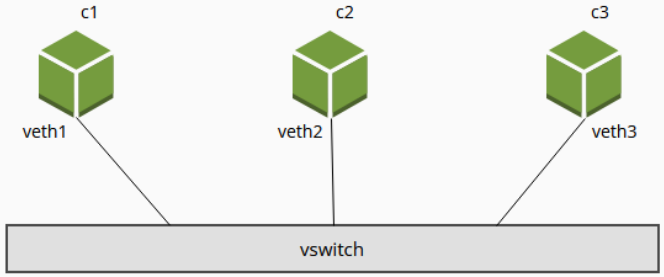
2/ Création d'un "switch virtuel"
ip link add vswitch type bridge
ip link set dev vswitch up

3/ Création des "câbles virtuels"
ip link add vethc1 type veth peer vethc1-vswitch
ip link add vethc2 type veth peer vethc2-vswitch
ip link add vethc3 type veth peer vethc3-vswitch

4/ Activation des cables (exemple pour 1er conteneur c1)
ip link set vethc1 netns c1
ip link set vethc1-vswitch master vswitch
ip link set vethc1-vswitch up

5/ Configuration des IP (exemple pour premier conteneur c1)
ip netns exec c1 ip addr add 172.16.0.20 dev vethc1
ip netns exec c1 ip link set vethc1 up

6/ Définir les routes réseaux (exemple pour conteneur c1)
ip netns exec c1 ip route add 172.16.0.0/24 dev vethc1



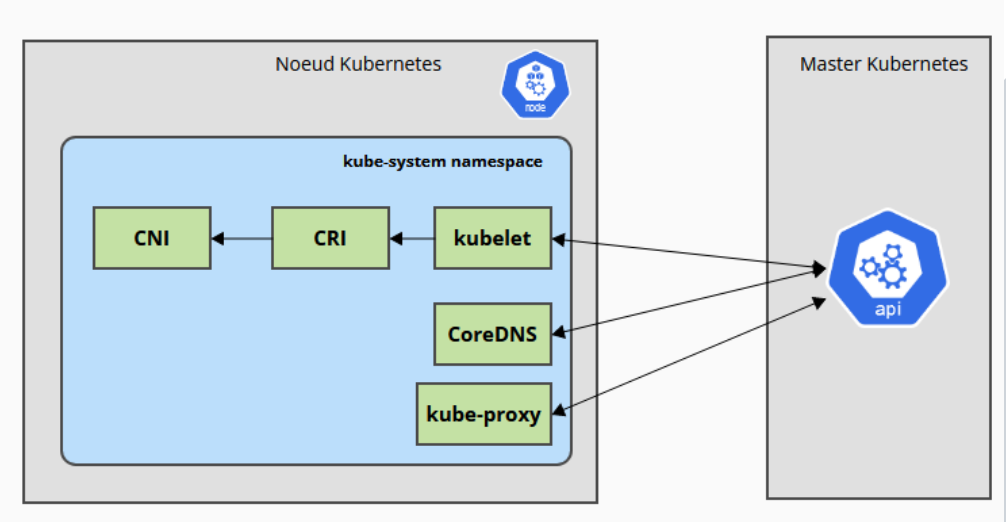
Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Architecture Kubernetes : le réseau

Ce schéma illustre les principaux composants qui interviennent dans la gestion du réseau.

CoreDNS : Serveur DNS qui permet de définir des entrées DNS
Kube-proxy : définit les règles de routage à travers Iptable (par défaut) afin que les services puissent atteindre les destinations
CRI (container runtime interface) : spécifications pour le moteur d'exécution (dans notre cas Docker)
CNI (container network interface) : spécifications pour la gestion réseau des conteneurs (piloté par le CRI)



Les spécifications CNI

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les spécifications CNI

CNI : présentation

Le **Container Network Interface (CNI)** est une spécification standardisée qui définit comment les plugins réseau doivent interagir avec les conteneurs et les orchestrateurs de conteneurs, tels que Kubernetes.

CNI a intégré le **CNCF (Cloud Native Computing Foundation)** le 23/05/2017 ce qui démontre la maturité de ces spécifications et la volonté de diffuser ce standard.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les spécifications CNI

CNI : présentation

CNI a pour mission de standardiser la configuration du réseau (assignation d'IP, configuration des interfaces, etc.) et l'interopérabilité entre les différentes solutions d'orchestrations (Kubernetes, OpenShift, Amazon ECS, etc.)

CNI est donc un ensemble de standards, mais n'est pas une **solution d'implémentation**

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Réseau dans Kubernetes Concepts et solutions CNI

Les plugins (solutions CNI)

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Les solutions (plugins) CNI

Il existe plusieurs solutions CNI, il faut noter que le choix d'une solution CNI dépend des besoins de votre cluster.

En effet une solution CNI s'étudie sur différents points :

- simplicité de mise en place,
- de la performance attendue,
- Les fonctionnalités avancées ou spécifiques qu'elle apporte (NetworkPolicies).

On pourra citer quelques solutions : Flannel, Calico, Weave, Cilium et Kube-router.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Comparaison des solutions

Feature	Calico	Flannel	Weave	Cilium
Networking model	Layer 3	Layer 3 overlay	Layer 2 mesh	eBPF
Performance	High	Good	Good	High
Scalability	High	High	High	High
Security features	Advanced	Basic	Basic	Advanced
Ease of use	Good	Easy	Easy	Moderate
Maturity	Mature	Mature	Mature	Mature

source : https://www.devopsschool.com/blog/compare-the-differences-between-calico-flannel-weave-and-cilium/
Networking model (modèle OSI) : https://fr.wikipedia.org/wiki/Mod%C3%A8le_OSI

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Comparaison des solutions

- https://github.com/containernetworking
- https://github.com/weibeld/cni-plugin-comparison
- https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Réseau dans Kubernetes Concepts et solutions CNI

Ingress Controller

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Ingress Controller

Un **Ingress Controller** dans Kubernetes est un composant responsable de la gestion du trafic entrant vers les services au sein du cluster. Il agit comme un point d'entrée pour les requêtes HTTP et offre une solution de routage et de gestion du trafic basée sur des règles.

Il répond a des besoins qu'un service Kubernetes (clusterIP, nodePort, externalName et LoadBalancer) ne peut pas offrir, dont notamment :

- routage basée sur des nom DNS : si on souhaite atteindre le pod1 avec un nom comme http://montest.local, cela n'est pas possible
- URL rewriting : certains Ingress Controller peuvent effectuer de la réécriture d'URL

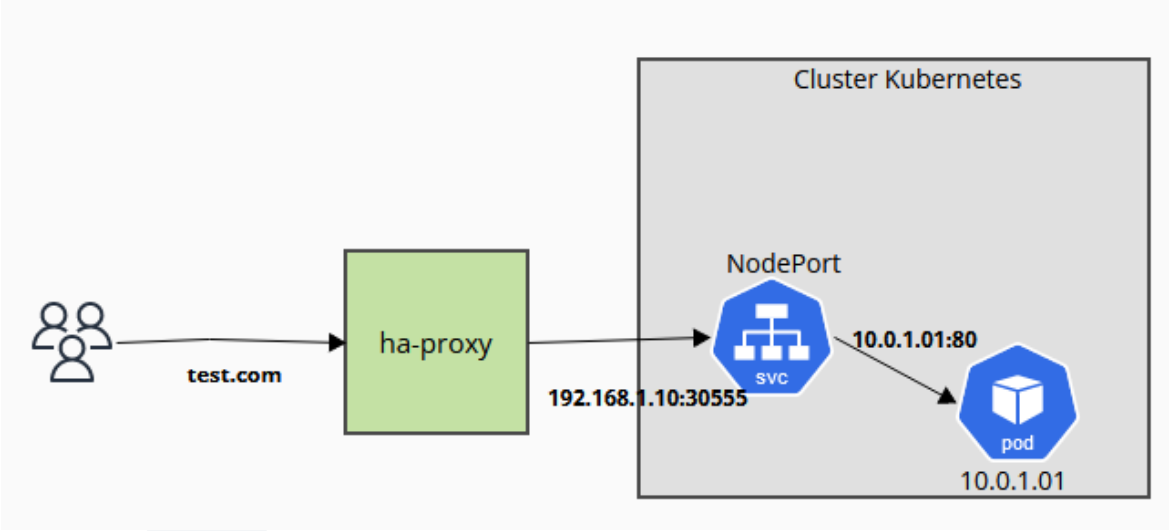
Il va agir en point d'entrée comme un reverse-proxy.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Ingress Controller

Une solution alternative à utiliser un Ingress Controller serait de mettre en place une solution en externe du cluster comme un ha-proxy en amont, mais cela ne serait pas le plus adapté (car ce composant serait en dehors du cluster).



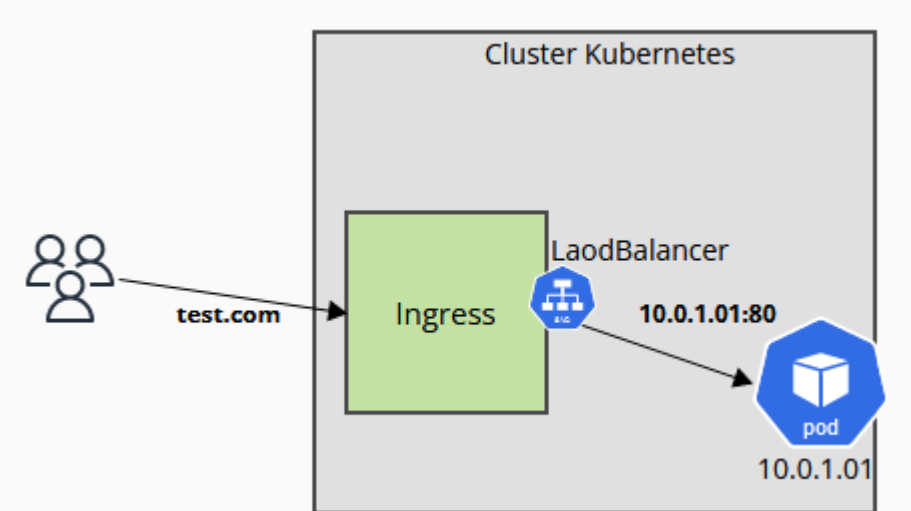
Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Ingress Controller

Le schéma suivant illustre le comportement avec un Ingress Controller.

Paradoxalement, même si les services K8s ne permettent pas de répondre aux besoins que nous avions mentionnés, Ingress Controller va utiliser un service de type LoadBalancer (par défaut) ou NodePort en « sous-marin » pour atteindre les pods.



Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Ingress Controller : les implémentations

Il existe plusieurs solutions pour implémenter un Ingress Controller sur Kubernetes.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Ingress Controller : les implémentations

Une fois, une solution d'Ingress Controller installé, il faut ensuite définir une ressource de type Ingress pour définir une règle de routage, par exemple :

- Une règle va rediriger les requêtes vers le host php.devx vers le service « service-mon-php » sur le port 8080 (le port clusterIP du service)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-service-php
spec:
  rules:
  - host: php.devx
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service-mon-php
            port:
              number: 8080
  ingressClassName: nginx
```

Mise en place nginx Ingress Controller (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Les concepts réseaux dans Kubernetes

Mise en place nginx Ingress Controller (TP)

Suivre le TP fournit par le formateur.

Chapitre 6 : Approfondissement Kubernetes

Module 6.1 – Réseau dans Kubernetes Concepts et solutions CNI

Chapitre 6 : Approfondissement Kubernetes

Module 6.2

Stockage persistant avec Persistent Volumes et Persistent Volume Claims

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Sommaire :

- Présentation des persistent volumes
- Gestion des persistent volumes
- Manipulation des volumes (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les volumes

Avant de mentionner les persistent volumes, il faut d'abord reprendre le concept de volumes.

Les volumes vont être un espace de stockage qui va être défini en dehors du conteneur, cela aura pour effet de :

- pouvoir partager un espace entre plusieurs conteneurs
- pouvoir reprendre des données si le conteneur est supprimé.

Chapitre 6 : Approfondissement Kubernetes

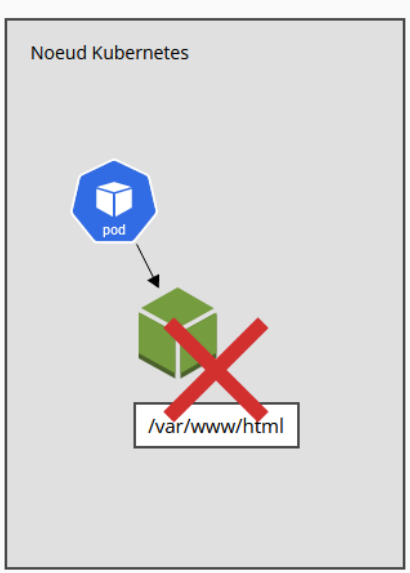
Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

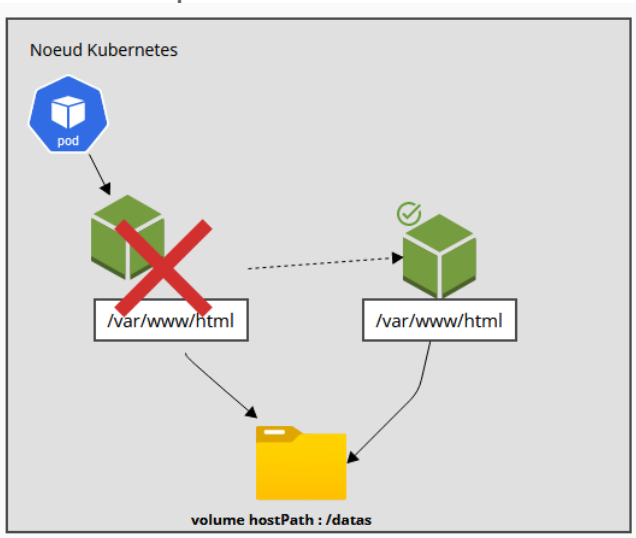
Les volumes

Un volume « traditionnel » va permettre la reprise des données si un conteneur va être supprimé.

Exemple sans volume



Exemple avec volume



Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les volumes

Exemple de manifeste YAML qui permet de monter un volume.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-volume
  labels:
    app: nginx-volume
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: html-volume
      mountPath: /usr/share/nginx/html
  volumes:
  - name: html-volume
    hostPath:
      path: /run/desktop/mnt/host/c/Temp
      type: Directory
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les volumes

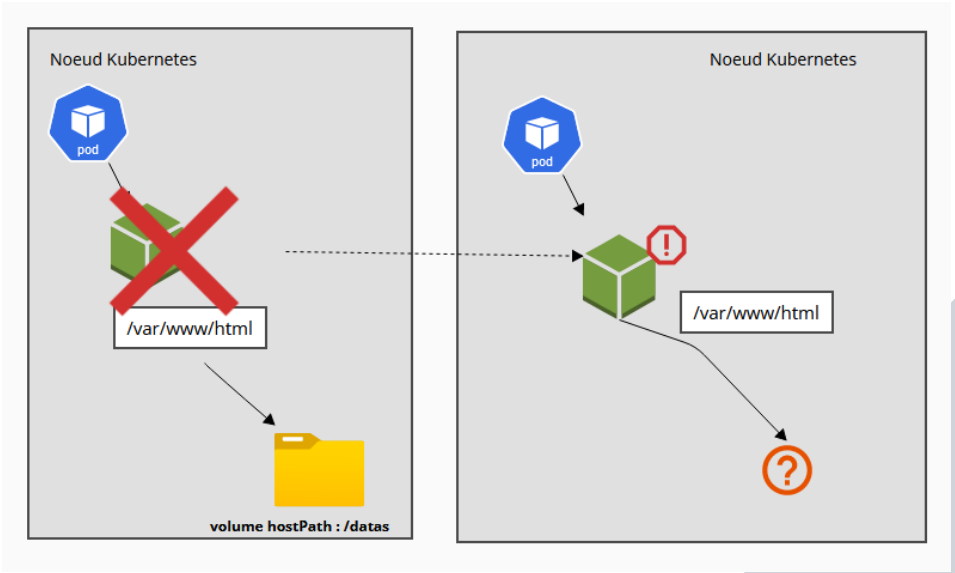
C'est bien au niveau d'un nœud, mais si on se trouve dans un cluster K8s productif avec plusieurs noeuds ?

Même exemple mais dans un cluster K8s où le conteneur est recrée sur un autre nœud.

Dans ce cas, on se retrouvera avec un problème : le volume ne sera pas présent car il est défini au sein du nœud !

C'est dans ce cadre que les persistent volumes viennent en aide avec des provisions sur des ressources réseaux comme :

- un partage NFS,
- amazon Elastic Block Store volume,
- google Compute Engine persistent disk
- un NAS



Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les Persistent Volumes (PV)

Les Persistent Volumes (PV) dans Kubernetes sont des ressources abstraites qui représentent du stockage réseau ou local dans un cluster. Ils permettent de séparer la gestion du stockage de l'application elle-même. Cela va **permettre le stockage et la persistance des données**.
Les Persistent Volumes vont donc **fournir** du stockage.

Les Persistent Volumes Claims (PVC)

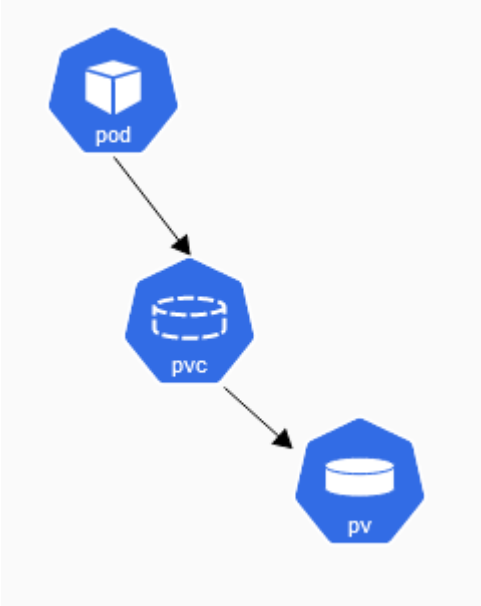
Les Persistent Volume Claims (PVC) sont des demandes émises par les applications pour obtenir de l'espace de stockage. Les PVC sont utilisés pour réserver un accès spécifique aux ressources de stockage définies par les PV.
Les persistent Volumes Claims (PVC) vont alors **consommer** du stockage en pouvant attribuer des quota aux pods.

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Un pod exprime un besoin (PVC) et le système va affecter un volume adapté (PV)



Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Cette séparation entre le besoin applicatif et la gestion des volumes permet de séparer également les responsabilités :

- besoin applicatif = développeur / architecte applicatif
- gestion des volumes = ingénieur infrastructure

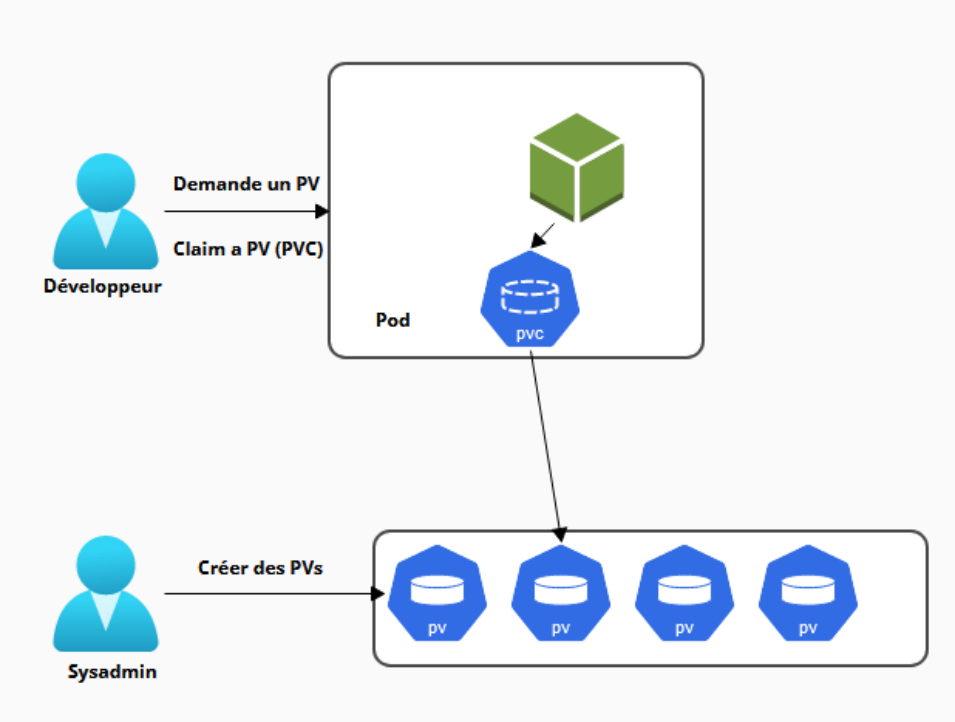
Un besoin applicatif va être par exemple, pour ce Pod j'ai besoin de 15Go très rapide sur SSD (donc coûteux à sauvegarder).

Dans la gestion des volumes, nous allons par exemple définir des volumes sur SSD et / ou sur HDD

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes



Chapitre 6 : Approfondissement Kubernetes

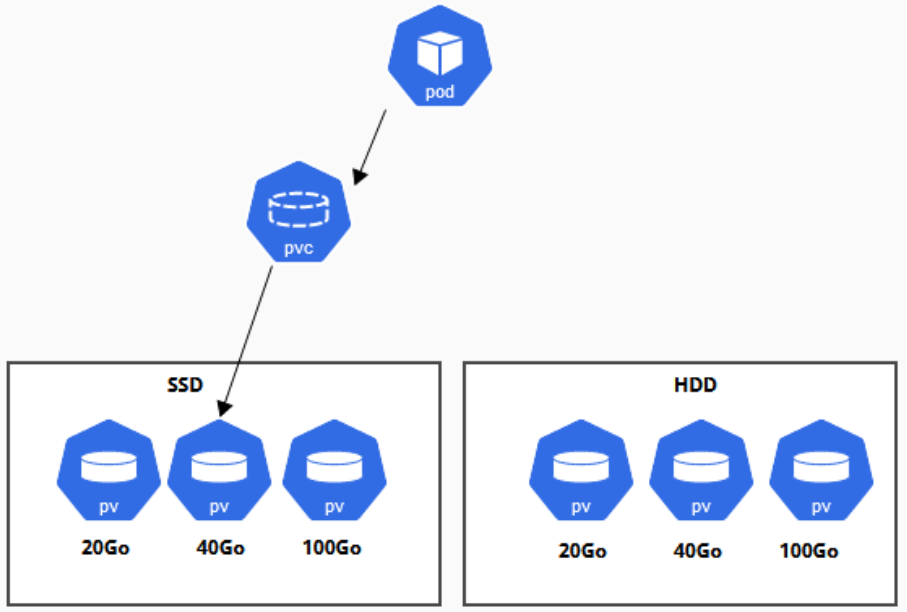
Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Ce pod a besoin d'un volume de type SSD de 30Go.

Le Pod va émettre une requête à travers un objet PVC, ensuite K8s à travers un algorithme et en analysant le **storageClassName** va alors attribuer le PV adéquat :

- SSD
- Nécessite 30 Go
  - → le PV de 20Go est trop petit
  - → le PV de 100Go est trop grand par rapport au 40G



Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Le **storageClassName** est un objet qui va permettre de catégoriser les PV afin que les développeurs puissent préciser des demandes plus pertinentes.

Par exemple, les sysadmin pourraient catégoriser ce genre de storageClassName :

- NFS_SSD
- NFS_HDD
- NAS
- EBS_azure
- Local
- Local_SSD
- Etc…

Cela implique également que le **storageClassName** doit pouvoir accéder et piloter à la ressource de stockage via un protocole de communication adapté **(un Provisioner)**.

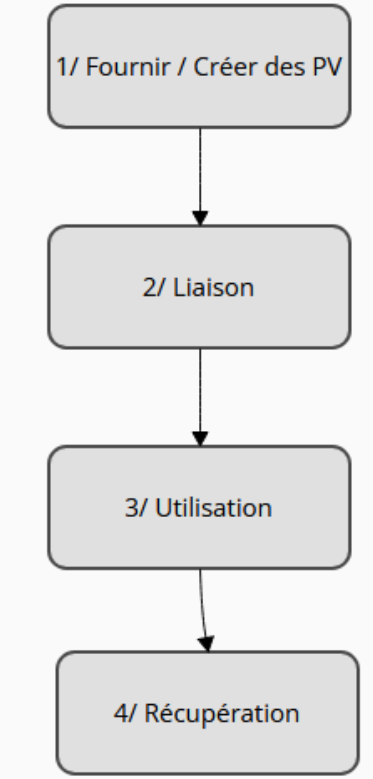
Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Le cycle de vie des PV / PVC

- **1/ Créer des Pvs** : la fourniture (provisioning) est réalisée de manière manuelle ou dynamique
  - Manuelle : un administrateur créer des Pvs et sont ensuite disponibles à la consommation
  - Dynamique : le cluster K8s peut si il est paramétré créer automatiquement des PV au besoin des PVC si le stockage le permet.
- **2/ Liaison** : en fonction des besoins (un PVC), si les conditions sont remplies alors un PV est attribuée
- **3/ Utilisation** : le pod utilise le PVC en tant que volume, une fois que le PV est lié et utilisé alors il appartient à l'utilisateur tant qu'il en a besoin.
- **4/ Récupération** : Lorsque l'utilisateur n'a plus besoin du volume, alors il peut supprimer l'objet PVC. A ce moment-là, le « reclaim policy » du PV va indiquer au cluster ce qu'il doit faire du volume après l'avoir relâché.
  - Retain : le volume n'est pas supprimé, il est considéré comme « released » et un sysadmin peut ensuite le supprimer ou traiter des données avant suppression.
  - Delete : supprime le PV
  - Recycle : permet de supprimer le contenu du PV et le rendre à nouveau disponible pour d'autres PVC



Chapitre 6 : Approfondissement Kubernetes

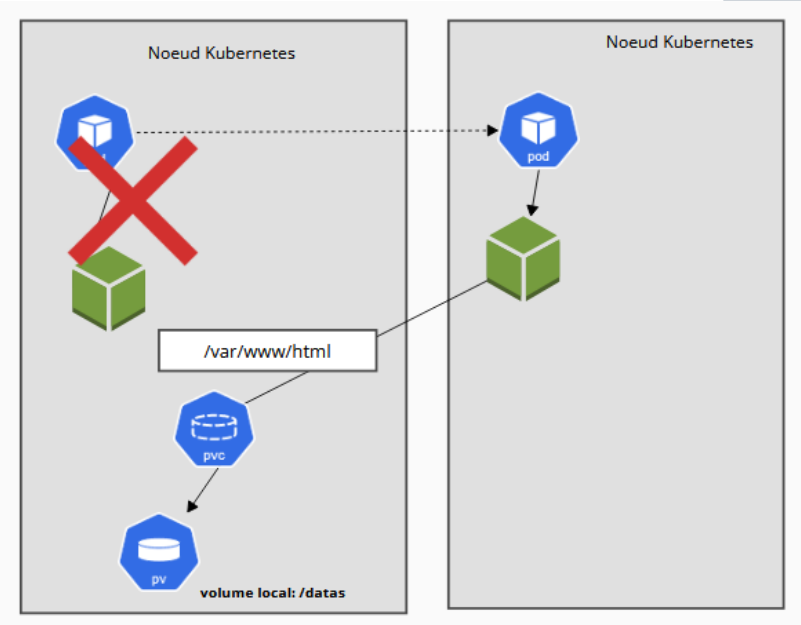
Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les persistent volumes et PVC

En conclusion, que se passe-t-il si tout un Pod venait à tomber, pourra-t-il reprendre son PVC ? La réponse courte est oui, mais c'est plutôt comment (et cela dépend du type de PV) ?

- Le type **hostPath** que nous avons vu précédemment devrait être utilisé uniquement pour les systèmes single node (test).
- Un type ressemblant est **local,** il va créer également un volume sur le nœud où se trouve le Pod, la différence est que le cluster va conserver en mémoire le nœud sur lequel il doit recréer le Pod pour qu'il puisse accéder au volume.



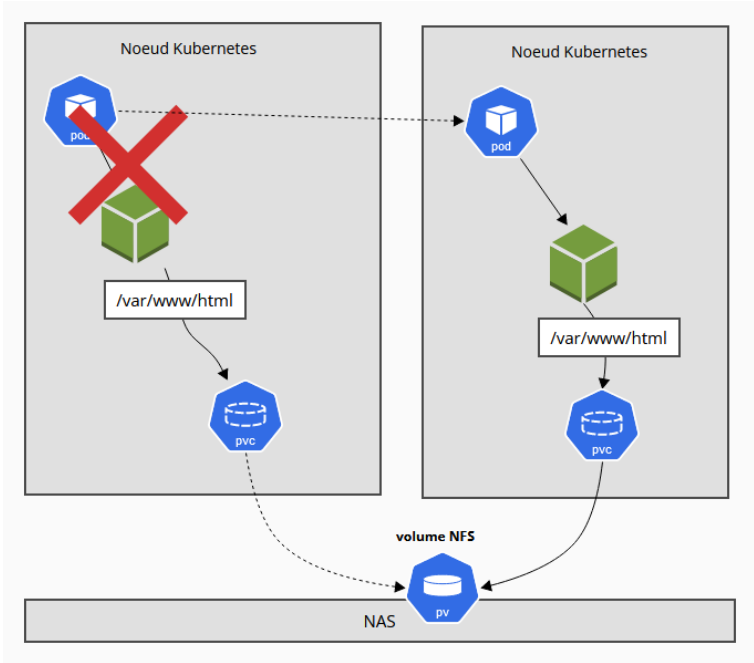
Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Les persistent volumes et PVC

- Sur d'autres type qui vont par exemple monter des shares sur des systèmes de stockage en dehors du cluster, le volume sera toujours accessible peu importe le nœud K8s.



Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Présentation des persistent volumes

Pour aller plus loin

Que se passe-t-il avec des replicas ? En effet, nous avons vu qu'un volume ne peut être lié qu'à un seul Pod, donc comment faire avec la notions de réplicas qui va multiplier le nombre de Pods?

Dans ce cas, il s'agit de la notion de **Volume Claim Template** qui va venir nous aider. Le principe est que si nous effectuons une mise à l'échelle du pod, alors le cluster va prendre en compte le template pour dynamiquement créer un PVC (une requête de stockage donc) et l'affecter au Pod.

L'ensemble est géré via un objet **StatefulSet**.

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Gestion des persistent volumes

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Gestion des persistent volumes

Création d'un Persistent volume

kind : PersistentVolume, le type de ressource à créer
spec : définit les caractéristiques du pv
  capacity :
    storage : la taille du disque
  storageClassName : la catégorie
  volumeMode : Filesystem → le volume est monté dans le Pod comme un répertoire (valeur par défaut)
  accessModes : définit le mode d'accès du PV
    ReadWriteOnce -- le volume peut être monté en lecture-écriture par un seul nœud
    ReadOnlyMany -- le volume peut être monté en lecture seule par plusieurs nœuds
    ReadWriteMany -- le volume peut être monté en lecture-écriture par de nombreux nœuds
  hostPath :
    path : chemin de l'hôte qui représente le volume
  persistentVolumeReclaimPolicy : Recycle → définit que le volume sera recyclé

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: hostpath-pv
  labels:
    disk: hdd
spec:
  capacity:
    storage: 3Gi
  storageClassName: hddlocal
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /run/desktop/mnt/host/c/Temp
  persistentVolumeReclaimPolicy: Recycle
```

```
kubectl get pv
```

```
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
hostpath-pv   3Gi        RWO            Recycle          Available           hddlocal               3m23s
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Gestion des persistent volumes

Création d'un Persistent volume claim

kind : PersistentVolume, le type de ressource à créer
spec : définit les caractéristiques du pv
  capacity :
    storage : la taille du disque
  storageClassName : la catégorie
  volumeMode : Filesystem → le volume est monté dans le Pod comme un répertoire (valeur par défaut)
  accessModes : définit le mode d'accès du PV
    ReadWriteOnce -- le volume peut être monté en lecture-écriture par un seul nœud
    ReadOnlyMany -- le volume peut être monté en lecture seule par plusieurs nœuds
    ReadWriteMany -- le volume peut être monté en lecture-écriture par de nombreux nœuds
  hostPath :
    path : chemin de l'hôte qui représente le volume
  persistentVolumeReclaimPolicy : Recycle → définit que le volume sera recyclé

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: hddlocal
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

```
kubectl get pvc
```

```
NAME     STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   AGE
my-pvc   Bound    hostpath-pv  3Gi        RWO            hddlocal       55s
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Gestion des persistent volumes

Utilisation d’un PVC pour un Pod

dans la section **volumes** du Pod, nous allons simplement préciser le pvc qui a été crée précédemment.

persistentVolumeClaim:
  claimName: my-pvc

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-pvc
  labels:
    app: nginx-use-pvc
spec:
  containers:
  - name: nginx-container
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: nginx-html-volume
      mountPath: /usr/share/nginx/html
  volumes:
  - name: nginx-html-volume
    persistentVolumeClaim:
      claimName: my-pvc
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Gestion des persistent volumes

Exemple avec un StatefulSet

Avec les statefulSet, nous pouvons :

- définir le nombre de replicas comme un deployment
- définir des volumeClaimTemplates afin de donner la possibilité aux pods de créer des request (PVC) dynamique lors de la créatic des replicas

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # doit correspondre à .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # est 1 par défaut
  template:
    metadata:
      labels:
        app: nginx # doit correspondre à .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: registry.k8s.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
          storage: 1Gi
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Gestion des persistent volumes

Les commandes utiles

- **kubectl get pv** : lister les PV disponibles
- **kubectl describe pv <nomPV>** : afficher les détails du volume persistent, dont notamment son status pour déterminer où il se trouve dans son cycle de vie, son type, quel est le PVC qui est lié etc…
- **kubectl get pvc** : lister les PVC disponibles
- **kubectl describe pvc <nomPVC>** : afficher les détails du volume persistent claim dont notamment le pod qui utilise ce PVC

Manipulation des volumes (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Manipulation des volumes (TP)

Suivre le TP fournit par le formateur.

Chapitre 6 : Approfondissement Kubernetes

Module 6.2 – Stockage avec persistent volumes et persistent volumes claims

Chapitre 6 : Approfondissement Kubernetes

Module 6.3
Auto-scaling et gestion des ressources

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Sommaire :

- Présentation de l’Auto-scaling
- Gestion des ressources
- Manipulation Auto-scaling (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Présentation de l’Auto-scaling

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

L’Auto-scaling

L'autoscaling (mise à l'échelle automatique), est un concept informatique où les ressources (nombre de serveurs, ou de conteneurs), sont ajustées automatiquement en fonction de la charge de travail ou de la demande en temps réel.

L’objectif principal de l’autoscaling va être de contribuer à garantir la disponibilité et la performance des applications, tout en optimisant les coûts en ne provisionnant que les ressources nécessaires à un moment donné. C’est une notion particulièrement utilisé dans les systèmes / solutions clouds (dont Kubernetes fait partie).

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

L’Auto-scaling

Dans K8s, on va observer plusieurs solutions pour répondre à la charge / sollicitation des applications :

- Ajuster le nombre de pods
- Ajuster les ressources attribués aux pods
- Ajuster le nombre de noeuds

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Les types d'Auto-scaling

Dans K8s, nous allons aborder 3 types d'autoscaling :

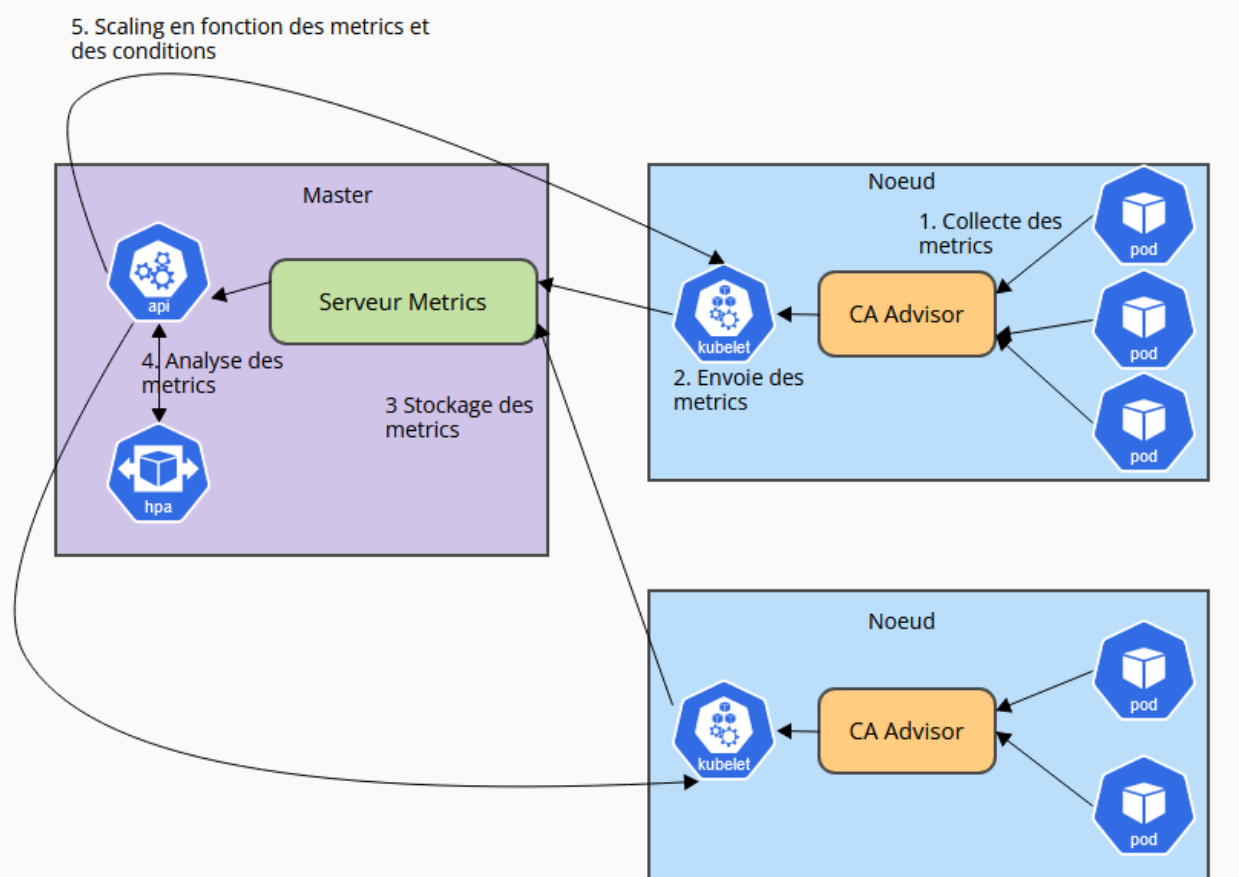
- Le **Horizontal Pod Autoscaler (HPA) :** gérer le nombre de pods
- Le **Vertical Pod Autoscaler (VPA) :** gérer l'attribution des ressources des pods
- Le **Kubernetes Nodes Autoscaler :** gérer le nombre de nœuds dans le cluster (cluster upscaling)

De manière générale, l'augmentation ou la réduction des ressources doit être basée sur une analyse de données que l'on appelle « **metrics** ». Ces données vont permettre au système en fonction de votre paramétrage de définir comment il doit réagir (augmenter ou réduire les ressources).

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Le principe de l'autoscaling dans Kubernetes



Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Horizontal Pod Autoscaler (HPA)

Le prérequis pour pouvoir effectuer un scaling d'une application et qu'elle soit déployée avec un objet de type Deployment.

Ensuite, une commande va pouvoir préciser les conditions du scaling : **kubectl autoscale**

kubectl autoscale deployment <votreDeploiement> --cpu-percent=50 --min=1 --max=10

Par exemple, la commande ci-dessus va permettre d'adapter le nombre de pods entre 1 et 10 replicas en fonction de la charge pour maintenir une charge CPU moyenne de 50 % sur l'ensemble des pods.

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Horizontal Pod Autoscaler (HPA)

Il est également possible de définir un objet de type HorizontalPodAutoscaler via un manifeste YAML :

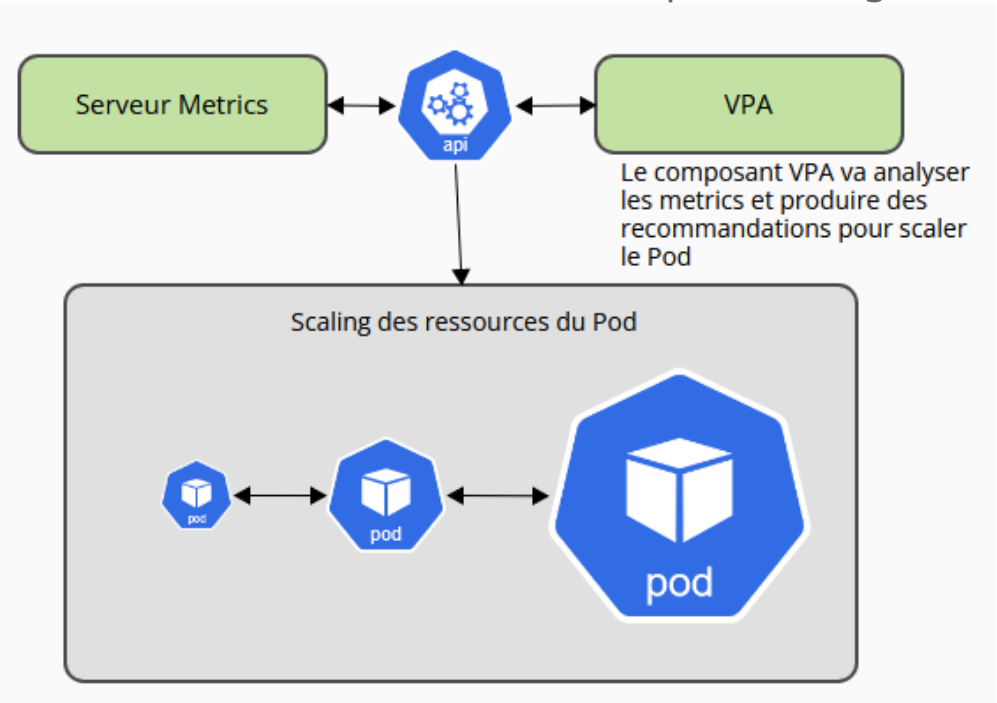
```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Vertical Pod Autoscaler (VPA)

- L'objectif du **Vertical Pod Autoscaler (VPA)** est d'adapter les ressources d'un Pod en fonction des besoins. Pour cela, il va se baser sur les metrics et le paramétrage du Pod.



Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Vertical Pod Autoscaler (VPA)

Ceci est un exemple de manifeste YAML qui va adapter les ressources du Pod :

updateMode : « **Auto** », K8s va recréer le Pod en se basant sur les recommandations.
valeurs possibles :
Off : Le composant VPA va uniquement produire des recommandations mais ne va pas mettre à jour automatiquement le Pod.
Initial : VPA assigne les ressources uniquement à la création du Pod mais plus après
Recreate : VPA assigne les ressources lors de la création / recréation du Pod

containerPolicies : la politique de paramétrage des ressources, le conteneur va avoir une allocation minimal ou maximal autorisée (minAllowed et maxAllowed)

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: nginx-vpa
  namespace: vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: mon-deploiement
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
    - containerName: "nginx"
      minAllowed:
        cpu: "250m"
        memory: "100Mi"
      maxAllowed:
        cpu: "500m"
        memory: "600Mi"
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Le Kubernetes Nodes Autoscaler

Le Kubernetes Node Autoscaler, souvent appelé Cluster Autoscaler, est un composant de K8s qui permet d'ajuster automatiquement le nombre de nœuds dans un cluster en fonction de la charge de travail.

Son objectif principal est de garantir que le cluster dispose toujours de suffisamment de capacité pour exécuter les applications sans interruption, tout en optimisant les coûts en supprimant les nœuds inutilisés lorsque la demande diminue.

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Gestion des ressources

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Gestion des ressources

Dans Kubernetes, il est possible de déclarer des limites ou des demandes dans la gestion des ressources au niveau des conteneurs qui sont situés dans les Pods.

Il est possible de définir les types de ressources suivantes :

- CPU : la puissance de calcul
- Mémoire : l'espace de stockage en mémoire vive

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Gestion des ressources – le CPU

La ressource CPU est défini en millicore. Si un CPU dispose de 2 cores, alors la valeur maximale possible est **2000m**.

Exemples :

- 250m pour 1 Pod, alors 4 Pods pourront être installés sur un nœud qui possède un CPU de 1 core.
- 500m pour 1 Pod, alors 2 Pods pourront être installés sur un nœud qui possède un CPU de 1 core.

A noter, un Pod qui ne dispose pas du pré-requis demandé sur un nœud, ne pourra pas être crée.

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Gestion des ressources – la mémoire

La mémoire est mesuré en octets, mais il est possible de la définir avec des échelles d'unités (Mo, Go, To, etc...)

A noter, tout comme pour le CPU, un Pod qui ne dispose pas du pré-requis demandé sur un nœud, ne pourra pas être crée.

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Gestion des ressources – définition

Au niveau du conteneur, une section **resources** permet de définir les demandes et les limites de CPU et mémoire.

requests : dans cette section, on définit ce que demande le conteneur au minimum.
- **Memory** : la quantité en mémoire en Mo
- **Cpu** : la quantité de calcul en millicore

Limits : dans cette section, on définit les limites que le conteneur ne dépassera pas.
- **Memory** : la quantité en mémoire en Mo
- **Cpu** : la quantité de calcul en millicore

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysqlctn
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "512Mi"
        cpu: "1000m"
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.3 – Auto-scaling et gestion des ressources

Chapitre 6 : Approfondissement Kubernetes

Module 6.4
Sécurisation des clusters avec les politiques de réseau et RBAC

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Sécurisation des clusters avec les politiques de réseau et RBAC

Sommaire :

- Les politiques de réseau (NetworkPolicies)
- Role-Based Access Control (RBAC)
- Manipulation RBAC (TP)

Les politiques de réseau (NetworkPolicies)

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau

La sécurité est un élément important dans un cluster Kubernetes, par défaut, tout est ouvert au niveau des Pods dans un cluster K8s.

En terme de sécurité, il s'agit évidemment d'un problème.

Les politiques de réseau (ou **network policies**) vont permettre de **contrôler le trafic réseau des Pods** et donc de pouvoir les **isoler** au niveau réseau à travers de règles (ou **rules**).

La définition de ces **network policies** est réalisée à l'aide d'un type d'objet **NetworkPolicy** dans Kubernetes.

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau - prérequis

La mise en place de network policies nécessite que le cluster K8s dispose d'une implémentation de solution CNI (ou plugin CNI).

Voici une liste (non exhaustive) de plugins CNI qui supporte les network policies :

- Weave
- Calico
- Cilium
- Romana

Remarques : Docker Desktop ne supporte pas les network policies.

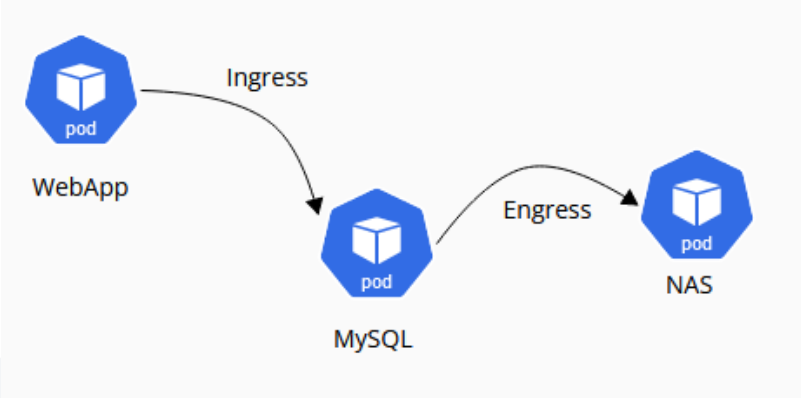
Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – les types de politiques

Il existe 2 types : **Ingress** pour les flux entrants du pod et **Engress** pour les flux sortants du pod.

L'exemple ci-dessous montre par exemple le flux d'une application Web qui nécessite un flux vers la base de données MySQL. La base de données, elle doit pouvoir sortir pour réaliser des backups sur un NAS par exemple.



Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – les règles

Les règles vont s'appuyer sur des namespaces, des labels et des PodSelector pour pouvoir définir sur quel Pod ces règles vont être actives. Il n'y a pas de règle de type **DENY,** il faut écrire dans le sens que ce qui n'est pas autorisé explicitement est donc interdit.

Pour rappel, par défaut, il n'y a aucune règle donc tous les Pods peuvent communiquer avec tout le monde.

Il serait possible de créer une règle par défaut pour isoler tous les Pods d'un namespace. L'exemple ci-dessous illustre cet exemple :

kind : le type d'objet NetworkPolicy
- **podSelector** : {} signifie que nous sélectionnons tous les Pods
- **PolicyTypes** : définit quel type de politiques nous déclarons
  - **Ingress** : les politiques entrantes du Pod
  - **Engress** : les politiques sortantes du Pod

Comme nous n'avons pas autorisé explicitement un flux, tout est donc interdit.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: monnamespace
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – quelques exemples

Ouvrir tous les flux de sortie sur tous les Pods

kind : le type d'objet NetworkPolicy

podSelector : {} signifie que nous sélectionnons tous les Pods du même namespace que la NetworkPolicy

Egress :

- {} signifie tous les flux entrants

PolicyTypes : définit quel type de politiques nous déclarons

- **Engress** : les politiques sortantes du Pod

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

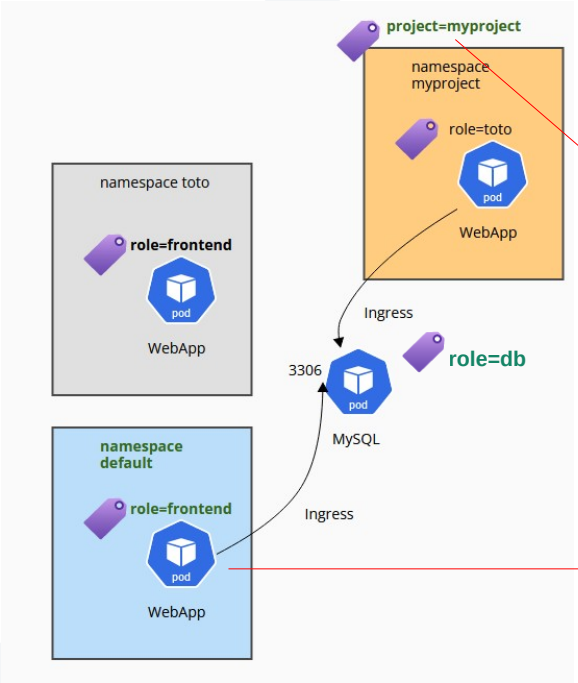
Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – quelques exemples

Ouvrir des flux entrants pour un pod MySQL

Exemple qui ouvre des flux d'un namespace **OU** d'une sélection de pods sur des labels du même namespace que la NetworkPolicy




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: myproject
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 3306
```

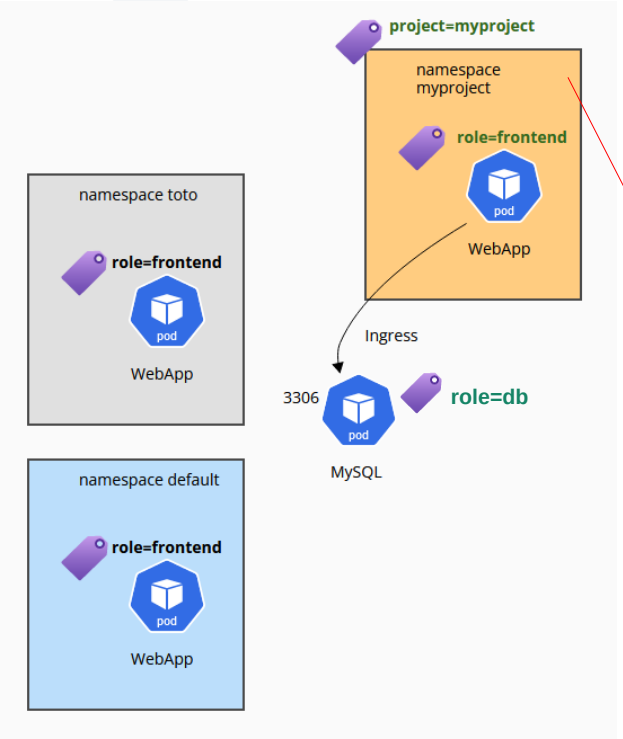
Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – quelques exemples

Ouvrir des flux entrants pour un pod MySQL

Exemple qui ouvre des flux d'un namespace **ET** d'une sélection de pods sur des labels sur le namespace qui a été sélectionné




```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: myproject
          podSelector:
            matchLabels:
              role: frontend
      ports:
        - protocol: TCP
          port: 3306
```

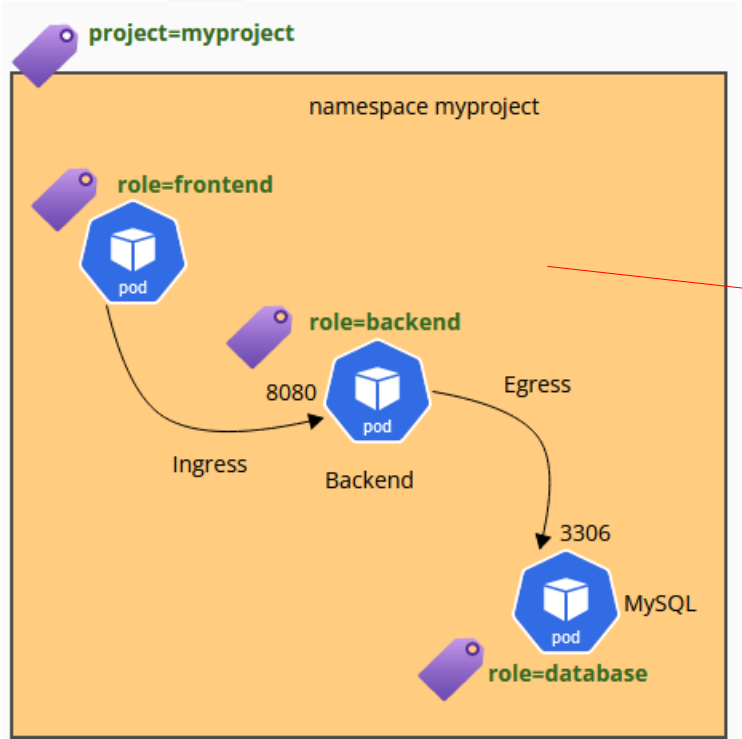
Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – quelques exemples

Ouvrir des flux entrants et sortants sur un Pod

Exemple d'un flux entrant et sortant sur un pod (dans ce cas le backend)




```yaml
apiVersion: networking.k8s.io/v
kind: NetworkPolicy
metadata:
  name: backend-network-policy
  namespace: myproject
spec:
  podSelector:
    matchLabels:
      role: backend
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
      ports:
        - port: 8080
          protocol: TCP
  egress:
    - to:
        - podSelector:
            matchLabels:
              role: database
      ports:
        - port: 3306
          protocol: TCP
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Les politiques de réseau (network policies)

Les politiques de réseau – quelques exemples

Ouvrir des flux avec un ipBlock

Exemple avec un **ipBlock,** un IpBlock déclare un range d'adresse IP (via un **cidr**) où le flux est autorisé.
la directive **except** signifie un range d'IP qui ne sera pas autorisé.

Dans la section ports, on peut préciser également un range de ports accessible avec **port** et **endPort**.

Cet exemple permet de sortir du Pod avec comme label « **role=db** » vers les pods : qui ont une adresse IP dans 172.17.0.0–172.17.0.255 et 172.17.2.0–172.17.255.255 et vers les ports **32000-32768**.

Les pods 172.17.1.0 – 172.17.1.255 sont exclues de l'autorisation.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: multi-port-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Egress
  egress:
    - to:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
      ports:
        - protocol: TCP
          port: 32000
          endPort: 32768
```

Role-Based Access Control (RBAC)

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC)

En terme de sécurité, nous avons vu comment contrôler le trafic du cluster avec les network policies.

Il reste à aborder la question des accès qui est un concept primordial au niveau de la sécurité. L'implémentation des droits d'accès est défini avec le concept de Role-Based Access Control (RBAC) dans Kubernetes, où les droits sont définis à travers des rôles et affecter aux utilisateurs.

Quelques définitions dans K8s

- **Les rôles :** les rôles vont définir les autorisations possibles (**verbs** : get, list, describe, etc...) sur des types de ressources (pods, deployments, services,etc...) dans Kubernetes. On trouvera deux types de rôles : le clusterRole (qui peut intéragir avec l'ensemble du cluster) et le rôle (qui est lié à un namespace)

- **Les utilisateurs** : Un utilisateur va être un consommateur des APIs de Kubernetes.
  On peut trouver deux types d'utilisateur : les **UserAccount** (utilisateur réel) et les **ServiceAccount** (utilisateur applicatif) qui va pouvoir être utilisé par un Pod.

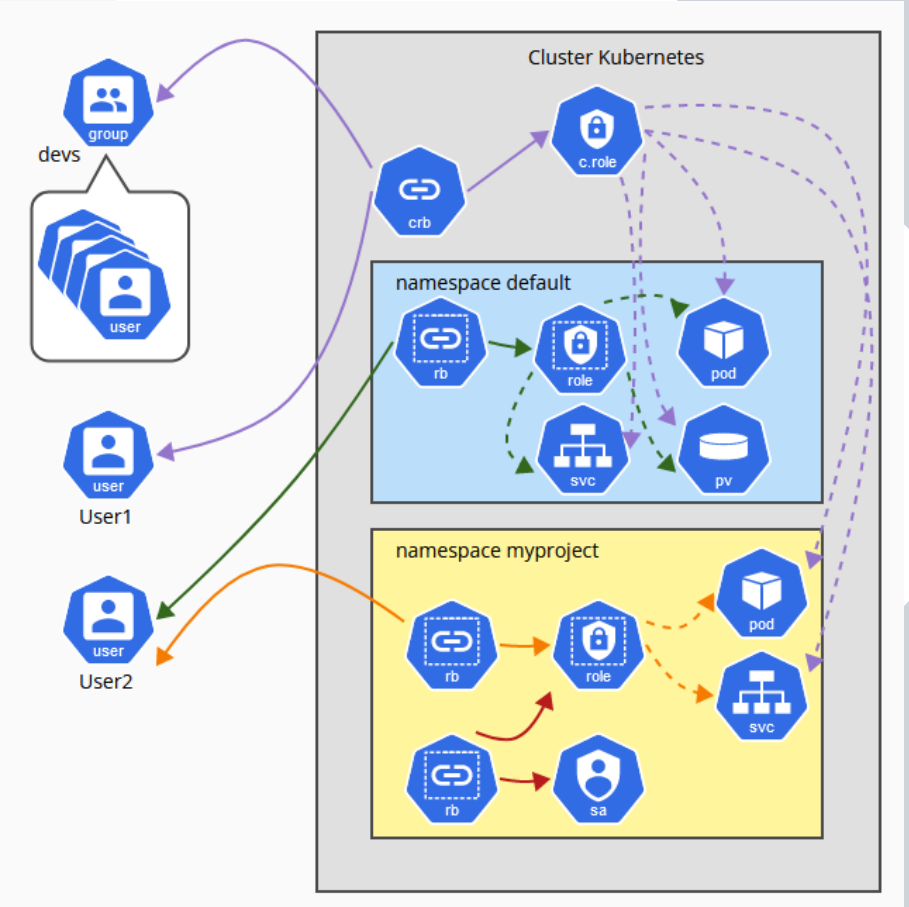
- **Les rôles bindings** : Il s'agit de l'affectation d'un rôle à un utilisateur. A noter :
  - on peut affecter un rôle à plusieurs utilisateurs différents dans le même roleBinding.
  - on peut également affecter un rôle à un groupe d'utilisateur.

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC)

Tous ces concepts peuvent être illustrés à travers ce schéma :



Chapitre 6 : Approfondissement Kubernetes

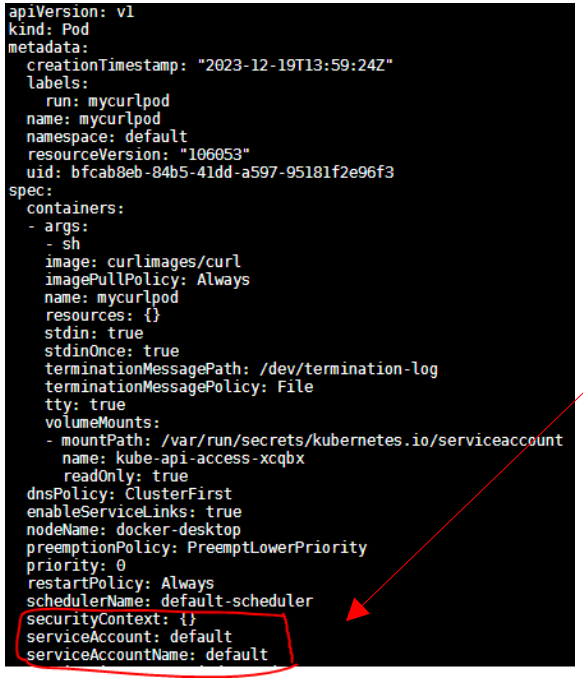
Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC)

Par défaut, un serviceAccount est utilisé par l'ensemble des Pods (mais avec des droits très limités).

Ceci peut être vu via une commande sur un Pod quelconque :

```
kubectl get pod mycurlpod -o yaml
```



Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) - exemple

Illustration de ces concepts par un exemple

1/ Création d'un namespace « rbac » et tous les objets suivants seront crées dans ce namespace

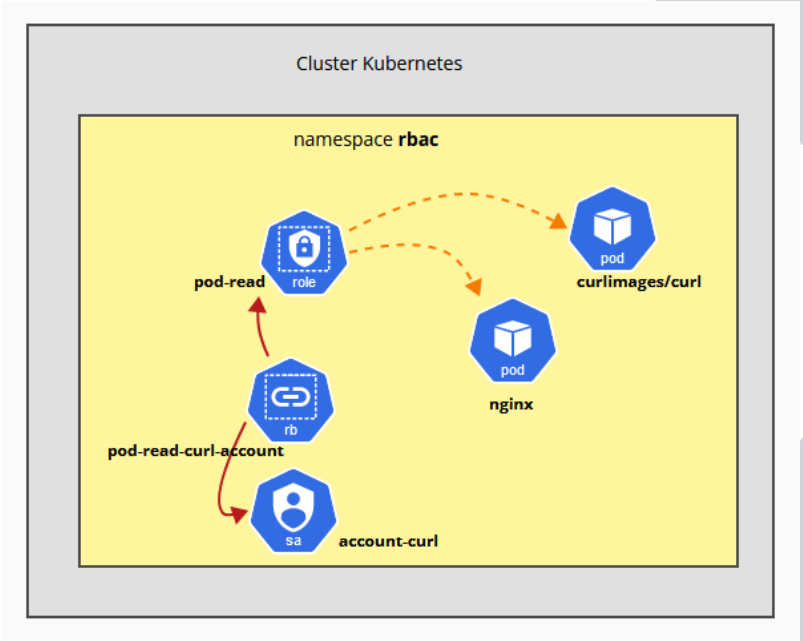
2/ Création d'un rôle « pod-read » qui dispose d'un droit de get et list sur les objets pods

3/ Création d'un serviceAccount « account-curl »

4/ Création d'un roleBinding qui va lier le sa « account-curl » au rôle « pod-read »

5/ Création d'un Pod « curlpod » qui va nous aider à interagir avec l'API de Kubernetes via des requêtes Curl. Ce Pod devra utiliser le serviceAccount « account-curl »

6/ Création d'un Pod nginx standard



Chapitre 6 : Approfondissement Kubernetes

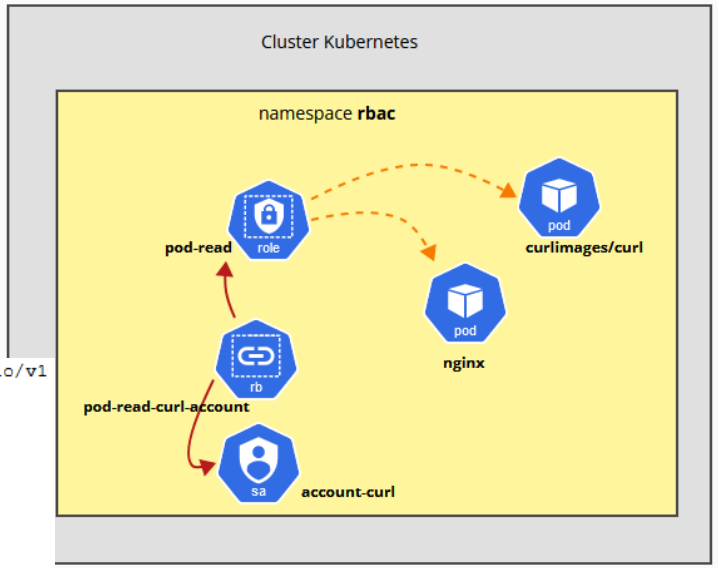
Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) - exemple

1/ Création du namespace :

```yaml
apiVersion: v1
kind: Namespace
metadata:
  creationTimestamp: null
  name: rbac
spec: {}
status: {}
```

2/ Création du rôle pod-read :




```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  creationTimestamp: null
  name: pod-read
  namespace: rbac
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
```

- **kind** : Role, le type de resource
  - **rules** : les règles du rôle
    - **apiGroups** : le groupe d'api sur lequel porte les ressources que nous allons gérer. valeur à vide dans ce cas, car « v1 » sans label pour les pods au niveau de l'apiVersion.
    - **resources** : nous souhaitons autoriser la resource de type pods
    - **verbs** : les actions qui sont autorisées pour le rôle

Remarques : La commande **kubectl api-ressources -o wide**, permet de retrouver la liste des ressources et les apiGroups

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) - exemple

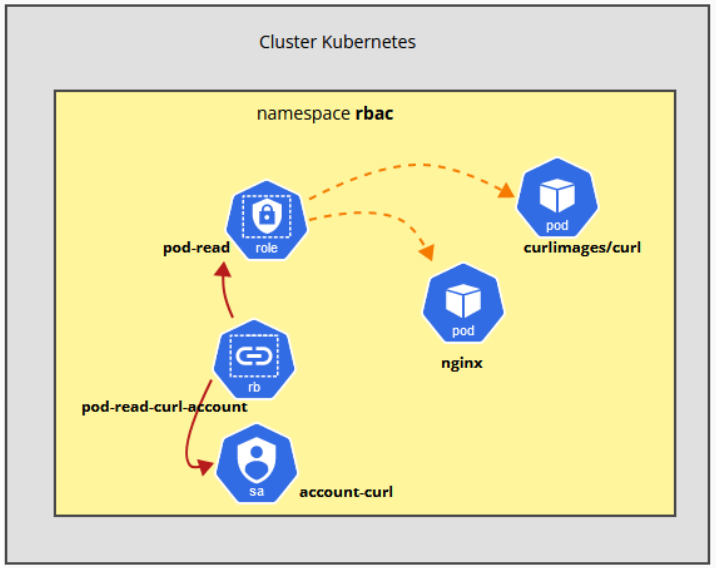
3/ Création d'un serviceAccount « account-curl »

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: null
  name: account-curl
  namespace: rbac
```

4/ Création d'un roleBinding qui va lier le sa « account-curl » au rôle « pod-read »

- **kind** : RoleBinding, le type de resource
  - **roleRef** : reference vers le rôle à lier
  - **subjects** : liste des éléments où va être apposer ce rôle.
    Dans notre cas, le serviceAccount « account-curl »

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  creationTimestamp: null
  name: pod-read-curl-account
  namespace: rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-read
subjects:
- kind: ServiceAccount
  name: account-curl
  namespace: rbac
```



Chapitre 6 : Approfondissement Kubernetes

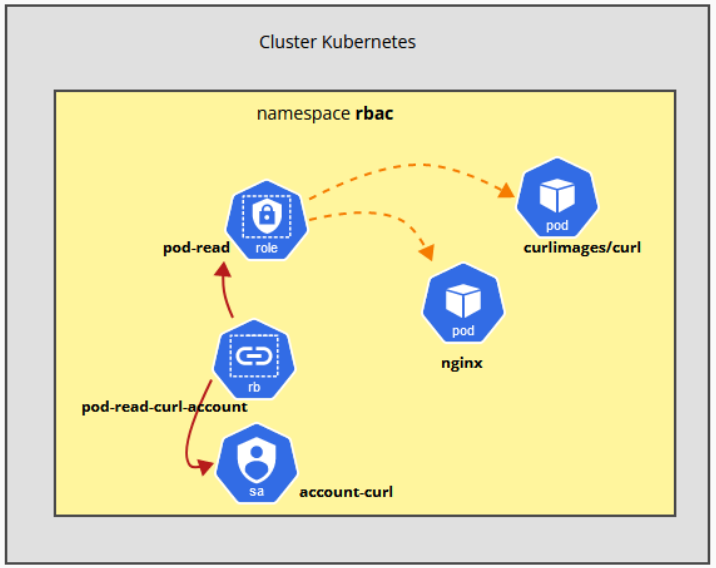
Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) - exemple

5/ Création d'un Pod « curlpod » qui utilise le serviceAccount

- **command** : sh, la commande à exécuter
- **tty** : true , indique que l'entrée standard doit être un terminal interactif
- **stdin** : true , indique de laisser l'entrée standard ouvert pour que l'on puisse s'attacher dessus
- **serviceAccount** : nom du compte (déprécié)
- **serviceAccountName** : nom du compte

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: curlpod
  namespace: rbac
spec:
  containers:
  - name: curl
    image: curlimages/curl
    command: [ "sh" ]
    tty: true
    stdin: true
  serviceAccount: account-curl
  serviceAccountName: account-curl
```



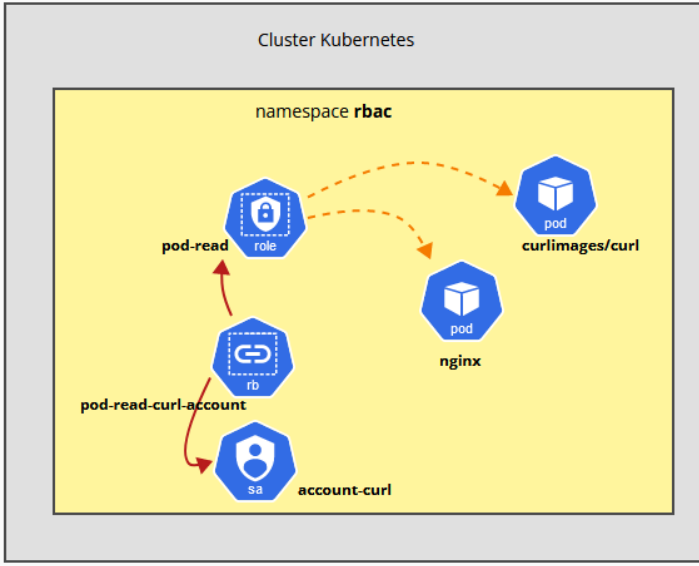
6/ Création d'un Pod nginx standard

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-nginx
  namespace: rbac
  labels:
    app: app-nginx
spec:
  containers:
  - name: ctn-nginx
    image: nginx
    ports:
    - containerPort: 80
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) - exemple



Tester les droits

1/ Se connecter au Pod de debug curlPod

kubectl attach curlpod -it -c curl

2/ Récupérer le token d'authentification du compte de service

TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)

3/ Tester l'API Server

- **curl -H "Authorization: Bearer $TOKEN" https://kubernetes.default.svc/api/v1/pods –insecure**

On observe que sur le namespace « default », le Pod n'a pas accès à la liste des pods.

```
~ $ curl -H "Authorization: Bearer $TOKEN" https://kubernetes.default.svc/api/v1/pods --insecure
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:rbac:account-curl\" cannot list re
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
  },
  "code": 403
```

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Role-Based Access Control (RBAC)

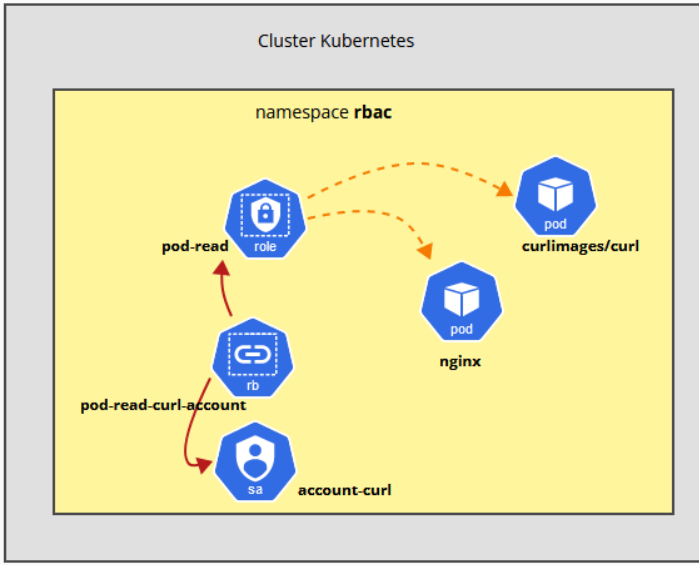
Role-Based Access Control (RBAC) - exemple

Tester les droits

curl -k -H "Authorization: Bearer $TOKEN" https://kubernetes.default.svc/api/v1/namespaces/rbac/pods

Sur le namespace « rbac », l'utilisateur dispose bien des droits.

```
~ $ curl -k -H "Authorization: Bearer $TOKEN" https://kubernetes.default.svc/api/v1/namespaces/rbac/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "128877"
  },
  "items": [
    {
      "metadata": {
        "name": "curlpod",
        "namespace": "rbac",
        "uid": "0cefa17f-28a7-4591-8f5c-1216f1d0e25a",
        "resourceVersion": "124822",
        "creationTimestamp": "2023-12-29T21:47:37Z",
        "managedFields": [
          {
            "manager": "kubectl-create",
            "operation": "Update",
```



Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Sécurisation des clusters avec les politiques de réseau et RBAC

Manipulation RBAC (TP)

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Sécurisation des clusters avec les politiques de réseau et RBAC

Manipulation RBAC (TP)

Suivre le TP fournit par le formateur.

Chapitre 6 : Approfondissement Kubernetes

Module 6.4 – Sécurisation des clusters avec les politiques de réseau et RBAC