# Notes CKA

Cheat sheet perso :

- Débugger une ressource : `kubectl describe` ou `kubectl logs` pour voir les logs du conteneur dans le pod
- Créer le squelette yaml d'une ressource (ex : deployment puis ingress)

```
kubectl create deployment mon-pod --image=mon-image --dry-run=client -o yaml > test.yaml
```

ou

```
kubectl create ingress nom-de-l-ingress --rule="/chemin=service:port" --dry-run=client -o ya
```

- Pour créer rapiddement un pod avec un nom et une image spécifique :

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

- Impératif : commandes `kubectl` pour créer, modifier, supprimer les objets
- Déclaratif : Fichier YAML avec toute la config et juste `apply` => à utiliser principalement, car plus clair, pratique et permet de versionner ces fichiers (git...)
- Créer un service :

```
kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o
```

ou

```
kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml
```

- Créer une `ConfigMap` ou un `Secret` (remplacer `configmap` par `secret`, et rajouter generic après `Secret`):

```
kubectl create configmap configmap-name --from-literal=KEY=value --dry-run -o yaml > configm
```

ou

```
kubectl create configmap <configmap_name> --from-file=<path-to-file>
```

- `kubectl apply` permet de créer et mettre à jour les objets, contrairement à `kubectl create` qui ne peux que créer et renvoie une erreur si l'objet existe. `kubectl apply` fonctionne en plusieurs étapes. D'abord, elle met à jour la configuration de l'objet live qui tourne sur le cluster, puis met à jour le fichier JSON correspondant à la "dernière configuration appliquée". Ce fichier JSON permet notamment de suivre les suppressions des champs et garder plus facilement une trace des congifurations. Ce fichier JSON est stockée dans une annotation dans le YAML de l'objet live.
- Sur k3s, les configurations du service `agent` ou `server` sont consultables ici `/etc/systemd/system/`

- Les informations concernant le/les cluster(s) auxquels j'ai accès sont contenues dans le kubeconfig icic `~/.kube/config`. On peut le consulter notamment via `kubectl config view`. ## Core Concepts

**CRI**

- Le `CRI` est le container runtime, le moteur d'exécution des conteneurs sur les noeuds. Si c'est `docker`, alors on peut utiliser les commandes `docker` sur les noeuds (via ssh par ex). Si c'est `containerd`, alors on peut utiliser `crictl` ou `nerdctl`. Elles fonctionnent comme `docker`, `sudo crictl ps` par ex.

**ETCD**

- Base de données en paire clé valeurs, qui stocke les états du cluster (nodes, pods, configmaps, Secrets, accounts...)
- Binaire `etcdctl` qui permet de communiquer enn lignes de commande avec cette dernière
- Un changement dans le cluster est considéré comme effectué une fois que la maj dans l'etcd est faite

**Kube API server**

- C'est le point d'entrée du cluster pour l'utilisateur (`kubectl`, `curl` sur l'API)
- Ex : `kubectl get pods` tape sur l'API server (en validant d'abord la requête), qui va interroger l'etcd pour connaître l'état des pods
- C'est le centre de toutes les opérations
- Ex 2 : `kubectl apply -f pod.yml` tape sur l'API server, qui crée le pod sans l'assigner à u noeud. Le `scheduler` observe en permanence l'API server, constate la création d'un pod dans noeud, et s'occupe de choisir le noeud hôte

**Kube controller manager**

- Il surveille le cluster et fait les actions nécessaires pour garder un l'état voulu
- Il existe une multitude de controllers qui peuvent composer le `kube controller manager`. Par exemple, il peut y avoir le `node controller`, qui contrôle les noeuds, le `namespace controller`, le `deployment controller`...
- Il surveille les composants du cluster via le `kube API server`, comme la surveillance des noeuds par exemple, qu'il fait toutes les 5 secondes
- Si un noeud présente un problème, il le marque en `NotReady` et relance les pods sur un noeud sain

**Kube Scheduler**

- Le rôle du `scheduler` est uniquement de décider quel noeud accueillera chaque pod
- Il ne fait pas l'action lui même, c'est le `kubelet` qui s'en charge.
- Il filtre les noeuds qui peuvent accueillir le pod selon ses exigences (CPU, mémoire, label...)

**kubelet**

- Le rôle du `kubelet` est de gérer les interactions entre le(s)`controller(s)` et son `worker` associé.
- Il envoie des rapports concernant l'état du `worker` à l'`API server`, et se charge d'exécuter les actions qui lui sont demandées sur le 'worker(création des pods...)

**kube-proxy**

- Le rôle de `kube proxy` est de gérer la mise en réseau des noeuds et pods.
- Il est présent sur chaque noeud, et est chargé de créer les règles adéquates à chaque nouveau service du cluster (service associé à un pod...) pour que la redirection d'IP soit menée à bien (iptables)
- Un service avec une IP A redirige sur le pod associé avec une IP B
- Ce système de service permet de pallier le problème d'IP dynamique des pods, par exmeple lorsque qu'un pod est détruit et recréé. L'IP du service est fixe.

**Pods**

- Le pod représente une instance d'une application, c'est la plus petite unité du cluster

**ReplicaSets**

- `ReplicaSet` désigne à la fois la ressource en elle-même (on préfère cependant utiliser la ressource `Deployment` pour gérer les répliques), mais aussi le controller qui se situe dans le `kube controller manager`. Il s'assure que le bon nombre de pods sont en fonctionnement, et permet également de rajouter des répliques si la charge augmente pour davantage la répartir entre ces dernières.

**Deployments**

- C'est la ressource principale utilisée pour déployer les pods
- Un `Deployment` crée les `ReplicaSets nécessaires`, ainsi que les pods. Il contrôle ainsi toute la chaîne de déploiement des pods.

- Les deployments sont idéaux pour gérer les répliques. Ils permettent notamment de mettre à jour les objets de manière fluide avec les `Rolling Updates`, d'annuler les changements, de les mettre en pause et de reprendre.

**Services**

- Les services permettent l'interconnexion des pods, en rendant possible la communication de ces derniers vers l'intérieur et extérieur du cluster
- Puisque chaque pod a sa propre IP et change souvent à cause des destructions et créations, un service permet d'avoir une seule IP en tant que point d'entrée qui redirige ensuite vers les pods associés
- Un service permet de mapper le port d'une IP vers le port d'une autre IP.
- Il existe plusieurs types de services.
  - `NodePort` : Dans le cas d'un accès externe, le service mappe un port sur l'adresse IP des noeuds vers un port sur l'IP du pod en question.
  - `ClusterIP` : Crée une IP virtuelle au sein du cluster, qui permet aux ressources internes au cluster d'accéder aux pods à travers ce service. Il permet notamment de mettre un nom sur le service, et donc d'avoir un point de contact fixe (sans être soumsi aux changements d'IP) vers les pods associés.
  - `LoadBalancer` : Crée une répartisseur de charge entre les pods associés à ce service. Il permet l'accès à ces pods depuis l'extérieur du cluster sur les cloud providers qui intègrent cette ressource (gcp, azure, aws... mais pas présent par défaut dans Kubernetes)
- Les services peuvent être comparés à des serveurs virtuels, ils ont donc leur propre adresse IP interne au cluster.
- A partie du moment où un service (peu importe son type) a plusieurs pods associés, il agit par défaut comme un loadbalancer, avec un algorithme de répartition aléatoire
- Le port du noeud sur lequel le service (`NodePort`) "écoute" peut être compris dans un intervalle de [30 000, 32 767]
- Il est important d'utiliser les labels et selector dans ce cas, pour que le service sache quels sont ses pods associés

**Namespace**

- Ils permettent de cloisonner les ressources
- Les pods de différents namespaces peuvent toujours communiquer à travers leurs services, en rajoutant `.<namespace>`.
- Il est possible qu'il faille même adapter l'URL de connexion des apps. ex : un frontend qui se connecte à un backend, tous deux situés dans 2 namespaces différents, doit adapter son url `connect("backend.<namespace>")`.
- Depuis un autre namespace, pour accéder au service `db-service` situé dans le namespace `dev`, il faut accéder à l'hôte `db-service.dev.svc.cluster.local`

### Resource Quota

- Ressource qui permet de limiter les ressources utilisées par un namespace

## Scheduling

### Scheduler

- En pratique, le `scheduler` applique son algorithme d'ordonnancement aux pods qui n'ont pas de champ `nodeName` défini dans leurs manifests. Cet algorithme détermine le noeud le plus approprié et ajoute ensuite ce champ dans le manifest. Le pod est ensuite déployé selon le champ en question. Il est possible d'hardcoder le noeud hôte en specifiant directement ce champ dans le mannifest.

### Labels & selector

- `kubectl get pod --selector env=dev` => renvoie tous les pods qui ont le label env évalué à dev
- Les `selector` permettent aux ressources (services, deployments, replicaset...) de cibler les bons pods sur lesquels elles doivent s'appliquer. Dans un `Deployment`, le `selector` doit matcher avec les `labels` des pods

### Taints & Tolerance

- `kubectl taint nodes <node-name> key=value:<taint-effect>` => taint-effect peut prendre les valeurs :
  - `NoSchedule` : Les pods ne seront pas planifiés sur ce noeud
  - `PreferNoSchedule` : le système essaiera d'éviter ces noeuds si possible
  - `NoExecute` : Aucun pod ne sera planifié sur ce noeud et les pods actifs seront expulsés (ceux qui n'ont pas la bonne tolerance)
- Les `taints` peuvent être attribuées aux noeuds, pour qu'ils soient marqués et puissent être reconnus par les attributs de `tolerance`
- La `tolerance` est un attribut des pods, qui permettent à ces derniers d'éviter d'être schedulés sur les noeuds qui ont une `taint` non tolérée.
- Si un noeud a une `taint`, sans spécifier la tolérance des pods, alors aucun pod ne peut être schedulé sur ce dernier. Il faut explicitement configurer la `tolerance` à cette `taint` pour qu'un pod puisse y être hébergé
- Cependant, un pod avec une `tolerance` sur le noeud 1 (qui a une `taint` correspondante) ne sera pas forcémenet schedulé sur ce dernier. En effet, le couple `taint`/`tolerance` permet uniquement d'exclure des pods de certains noeuds. Un pod avec une certaine `tolerance` peut tout à fait se retrouver sur un noeud sans `taint`. D'où l'existence d'un autre concept, l'`affinity`.
- Supprimer une taint : `kubectl taint nodes <nodename> <key>-`

### Node Selectors

- Permet de s'assurer que les pods tournent sur les noeuds appropriés
- Labéliser un noeud : `kubectl label nodes <nodename> <key>=<value>`
- On utilise ensuite le champ `nodeSelector` dans le manifest du pod pour qu'il soit scheduler sur le noeud en question

### Node Affinity

- Permet aussi de s'assurer que les pods tournent sur les noeuds appropriés, mais propose un filtrage plus fin
- Elle permet notamment d'utiliser des opérateurs `In`, `Not in`, `Exists`... sur les labels des noeuds. Ainsi, un pod dont l'`affinity` est :

  ```
  - key: <label_key>
  operator: In
  values:
  - <label_value1>
  - <label_value2>
  ```

  sera schedulé sur un des noeuds qui portent un de ces labels
- `Node Affinity Types` correspond aux différentes types d'affinité :
  - `requiredDuringSchedulingIgnoredDuringExecution : required` indique qu'il faut que les règles de match de labels soient respectées, sinon le pod n'est schedulé nulle part
  - `preferredDuringSchedulingIgnoredDuringExecution : preferred` indique qu'il faut que si les règles de match de labels ne sont pas respectées, alors elles sont ignorées
  - On distingue deux parties dans les types : `Scheduling` et `Execution`. Pour le `Scheduling`, le `required` ou `preferred` s'applique au moment du scheduling du pod. Pour le `Execution`, cela s'applique même au runtime. Ainsi, si un pod tourne sur un noeud et que les labels du noeud changent, et donc l'`affinity` plus respectée (si elle, est set à `required`), alors les pods sont éjectés

### Taints/Tolerance & Affinity

- Les deux concepts peuvent être utilisés ensemble pour obtenir un scheduling très précis

### Resources requirements & limits

- Chaque pod consomme des ressources en CPU & RAM
- Il est possible de préciser les ressources à allouer au pod via le champ

  ```
  resources:
      requests:
  ```

```
            memory: "4Gi"
            cpu: 2
```

- Il est possible de mettre en place des limites de consommation de ressources

```
resources:
    limits:
        memory: "5Gi"
        cpu: 4
```

- Un pod ne peut pas consommer plus de cpu que sa limite, en revanche, il peut consommer plus de RAM que sa limite, ce qui provoquera son interruption
- La requête signifie que le pod est garanti d'avoir le nombre défini de CPU, mais peut quand même dépasser, jusqu'à la limite si elle est définie.
- Si une limite est définie mais pas la requête, alors la requête prend la même valeur que la limite

### Daemon Sets

- Un `Daemon Set` s'assure qu'une copie du pod est toujours présente dans le noeuds du cluster (si un noeud s'ajoute, alors il crée une instance automatiquement)
- Cette ressource est parfaitement adaptée pour des services qui tournent en arrière plan pour du montoring ou de la collecte de logs par exemple. Peu importe comment évolue notre cluster, on est assurés que les noeuds possèdent les services de surveillance. (ex : `kube-proxy`)
- Le `Daemon Set` est autonome, le `kube scheduler` n'a pas d'influence sur les `pods` des `Daemon Sets`

### Static Pods

- Il est possible de déployer des pods sans `control plane`, uniquement sur les noeuds `workers` via le `kubelet`.
- Pour ce faire, il existe un dossier que le `kubelet` consulte régulièrement, et crée les pods s'il trouve des manifests correspondant.
- Il est possible de configurer le chemin de ce dossier en obtenant le nom du fichier que le service du kubelet consulte. Voir le fichier dans question ici `/etc/systemd/system/`. Un argument `--config=kubeconfig.yaml` avec le nom d'un fichier doit s'y trouver (ici `kubeconfig.yaml`). Dans ce fichier, on peut renseigner le champ `staticPodPath: /etc/kubernetes/manifests`, qui correspond au chemin vers le dossier qui contient le manifests des pods statiques.
- Les pods statiques sont principalement utilisés pour déployer les pods sur des noeuds qui ne sont pas sous le joug des control planes. On retrouve ce cas-là majoritairement pour les composants du control plane lui-même (`controller manager`, `api server`, `etcd`...). Cela permet aux services

d'être davantage robustes aux pannes (cf avantages `pods`) par rapport à un binaire classique déployé sous forme de processus classique.
- Les pods statiques sont repérables facilement car ils ont le nom du noeud sur lequel ils sont déployés ajouté à la fin de leur nom

### Multi schedulers

- Plusieurs schedulers peuvent coexister dans un cluster
- Il est possible de définir ses propres schedulers custom
- Pour le mettre en place, il suffit de faire la configuration du scheduler dans un fichier yaml, puis de faire pointer le pod du scheduler vers le fichier de config yaml, via le flag `--config=/path/to/scheduler-config.yaml` dans le champ `command` du pod scheduler.
- Si plusieurs schedulers peuvent coexister, un seul d'entre eux peut être actif à la fois. On peut ainsi configurer un leader et des règles qui vont avec pour
- Pour faire en sorte qu'un pod soit déployé avec le bon `scheduler`, on peut ajouter le champ `schedulerName` avec le nom du `scheduler`
- Il est possible de voir quel scheduler a été utilisé pour déployer un pod avec la commande `kubectl get events -o wide`

## Logging & Monitoring

### Monitoring

- Le monitoring a pour but de relever des métriques concernant les pods et noeuds
- Pas de solution native de monitoring, mais beaucoup de solutions open source (`prometheus`, `metrics server`...)
- `metrics server` est souvent utilisé (par défaut sur `k3s` par exemple)
- `kubectl top nodes` permet de suivre la consommation en ressources des noeuds
- `kubectl top pod` permet de suivre la consommation en ressources des pods

### Logs

- Il est possible de suivre les logs d'un pod via la commande `kubectl logs <pod_name>`
- Rajouter l'option `-f` permet de suivre les logs en direct
- Si un pod contient plusieurs conteneurs, on précise le nom du conteneur `kubectl logs <pod_name> <container_name>`

## Application Lifecycle Management

### Rollling updates & Rollback

- Quand un `Deployment` est mis en place, il tirgger un `Rollout`, qui déclenche une `Revision`. Une `Revision` correspond au marquage du déploiement dans une certaine version. Le `Rollout` est le processus qui déclenche la `Revision`. Ainsi, en modifiant la version de l'image dans le `Deployment`, et qu'on le réapplique, il relance un `Rollout` qui marque une nouvelle `Revision`. Ainsi, on a un suivi de l'évolution des versions du `Deployment`, ce qui permet de revenir à une précédente version du cluster.
- La commande `kubectl rollout status deployment/<deployment_name>` permet d'observer l'état du `rollout`
- On peut en observer son historique avec `kubectl rollout history deployment/<deployment_name>`
- Pour mettre à jour un `Deployment`, il existe plusieurs stratégies :
  - La première est le `Recreate`, qui consiste à détruire le `Deployment`, puis à le réappliquer avec la nouvelle version. Le problème est que pendant un instant, le service est indisponible.
  - La deuxième est le `Rolling Update`, qui remplace un par un les pods d'ancienne version par un pod de nouvelle version, ce qui a pour conséquence de ne pas rendre indisponible l'application
  - Sans préciser explicitement la stratégie de mise à jour, celle par défaut est le `Rolling Update`.
- En pratique, pour mettre à jour un `Deployment`, il suffit de modifier le manifest puis d'exécuter `kubectl apply -f <deployment_name.yaml>`. Cela déclenche le `Rollout` et crée une nouvelle `Revision`.
- On peut observer le la stratégie en utilisant `kubectl describe deployment <deployment_name>`. On onbserve bien que chaque replica est mis à jour l'un après l'autre
- Si la mise à jour du Deployment met en place un ensemble de replicas (`Replica Set`) non fonctionnel, il ets possible de `Rollback` au `Replica Set` précédent fonctionnel. Pour ce faire, on utilise la commande `kubectl rollout undo deployment/<deployment_name>`
- On peut observer les `replicasets` et le nombre de pod de ce replica set avec `kubectl get replicasets`

### Pod commands & arguments

- Dans le `YAML` d'un pod, le champ :
  - `command` du `container` override l'instruction `ENTRYPOINT` de l'image du conteneur. Dans un `Dockerfile`, cette instruction précise quel est l'exécutable appelé au début de la commande pour lancer le conteneur (ex : `bash`, `python3 example.py`, `npm`...).
  - `args` du `container` override l'instruction `CMD` de l'image du conteneur. Dans un `Dockerfile`, cette instruction précise quelles sont les commandes qui suivent l'exécutable precisé dans l'instruction `ENTRYPOINT`

(ex: `ls`, `--arg1 test`, `run dev`)
  - `env` du `container` override l'instruction `ENV` de l'image du conteneur. Chaque variable d'environnement est constituée d'une clé et d'une valeur. Il est possibe d'importer la valeur de la variable d'environnement depuis des ressources extérieures, comme une `ConfigMap` ou un `Secret`, grâce au champ `valueFrom`.

### ConfigMaps

- Une `ConfigMap` est une ressource Kubernetes qui contient des paires clé valeur
- Ces paires peuvent être appelées par d'autres ressources pour la configuration :

```
env:
    - name: EXAMPLE
    valueFrom:
        configMapKeyRef:
            name: configmap-name
            key: KEY
```

ou

```
envFrom:
    - configMapRef:
        name: configmap-name
```

pour récupérer toutes les entrées

### Secrets

- Les `Secrets` sont comme des `ConfigMaps`, mais servent à stocker des informations sensibles
- Les informations des `Secrets` ne sont pas chiffrées, juste encodées, donc les secrets ne doivent pas être rendus publics
- Il est préférable que la valeur des clés soit encodée (base64). Pour ce faire, utiliser la commande

```
echo -n '<value-to-encode>' | base64
```

- Pour les décoder, on fait

```
kubectl get secret <secret_name> -o yaml
```

qui permet de récupérer les valeurs des clés, que l'on peut ensuite décoder avec

```
echo -n '<value-to-decode>' | base64 --decode
```

- Ces paires peuvent être appelées par d'autres ressources pour la configuration :

```yaml
env:
    - name: EXAMPLE
      valueFrom:
        secretKeyRef:
            name: secret-name
            key: KEY
```

ou

```yaml
envFrom:
    - secretRef:
        name: secret-name
```

pour récupérer toutes les entrées

- Il est possible de mettre en place un chiffrement des secrets stockés sur `etcd` via une ressource particulière, `EncryptionConfiguration`. Ce type de chiffrement s'appelle `Encryption at Rest`.
- Il est recommandé de bien gérer les `RBAC` (Role-Based Access Control) pour gérer les privilèges d'accès aux `Secrets`
- Il est recommandé d'éviter de passer par `etcd` et d'utiliser un gestionnaire tier des secrets, comme `Vault`, ou cloud `AWS`, `GCP`, `Azure`

#### initContainers

- Ce sont des conteneurs qui sont exécutés avant tous les autres, ils sont souvent utilisés pour télécharger un binaire, setup des configs...
- Ils sont déclarés via le champ `initContainers` dans le `YAML`, et prennent la même structure que celles pour le champ `containers` classique

## Cluster Maintenance

### OS Upgrades

- `kubectl drain <node_name>` termine tous les pods sur le noeud en question, et les relance sur un autre noeud disponible. Il rend également le noeud `Unschedulable`. Le `drain` permet de redémarrer un noeud sans interrompre définitivement le service porté par les pods qui tournaient dessus (pour les pods sans replicas, le down time correspond au temps nécessaire à l'arrêt du pod en cours et à son redémarrage sur l'autre noeud).
- Rajouter l'option `--ignore-daemonsets` si des ds tournent sur le noeud
- `kubectl uncordon <node_name>` rend le noeud à nouveau `Schedulable`
- `kubectl cordon <node_name>` rend le noeud `Unschedulable`, mais ne termine pas les pods qui tournet dessus, contrairement à `drain`

### Cluster Upgrade Process

- Aucun composant de Kube ne peut être à une version supérieure à celle de l'`API server`. Le `controller manager` et `kube scheduler` peuvent

être à une version de moins que l'`API server`. Le `kubelet` et `kube-proxy` peuvent être à 2 versions de moins que l'`API server`

- Pour upgrade en version un cluster (composants kube), il existe plusieurs approches :
  - D'abord, peu importe le cas, les noeuds controllers sont upgrades en premier. Pendant le temps de la mise à jour, les controllers sont inaccessibles (donc pas de `kubectl`...). En revanche, les workers continuent à servir l'application avec l'ancienne version
  - 3 approches existent pour mettre à jour les workers une fois que les controllers sont à jour :
    * Tous les workers sont mis à jour en même temps. Cela a pour conséquence de rendre le service indisponible le temps de l'upgrade
    * Les workers sont mis à jour un par un. Cela permet de continuer à servir l'application même pendant la maj.
    * On ajoute des nouveaux noeuds workers avec la bonne version, et on leur réattribue au fur et à mesure la charge des anciens noeuds qui sont delete (méthode privilégiée sur les clouds)
- Si `kubeadm` est disponible (cluster sans distrib), il existe des outils d'assistance pour upgrade un cluster. `kubeadm upgrade plan` permet de visualiser l'état des mises à jour disponibles pour tous les composants, et donne les commandes à faire pour réaliser la mise à jour. `kubeadm` est le seul composant à mettre à jour manuellement (`apt` si `debian`...)
- La maj de `kubelet` est aussi manuelle si Kube sans distrib
- Il est important de faire les màj d'une version à sa version+1 à la fois
- Workflow d'upgrade manuelle :
  - `drain` node01
  - `ssh` node01, `upgrade kubeadm`, `upgrade kubelet`, `upgrade node config`, `systemctl restart kubelet`
  - `uncordon` node01
- Voir doc pour étapes

### Backup & Restore methods

- `kube api server` : Il est possible de réaliser un backup du cluster via l'`api server`, avec

  ```
  kubectl get all --all-namespaces -o yaml > all-services.yaml
  ```

  qui va aggréger tous les yaml dans un même template.

- `etcd` : Pour rappel, tous les états du cluster sont sauvegardés dans la base de données `etcd`. On peut voir le dossier dans lequel les données sont sauvegardées en inspectant le `etcd.service`, avec le flag `--data-dir`. On peut créer un snapshot de l'`etcd` avec l'utilitaire `etcdctl` avec la commande

  ```
  ETCDCTL_API=3 etcdctl snapshot save snapshot.db
  ```

Ensuite, on stoppe le `kube-apiserver` avec `service kube-apiserver stop`. On peut ainsi restaurer le cluster depuis ce snapshot via

```
ETCDCTL_API=3 etcdctl snapshot restore snapshot.db --data-dir /var/lib/etcd-from-backup
```

On ajoute ensuite ce path au flag `--data-dir` du fichier de config etcd pour mettre à jour le `etcd.service`. On modifie également le chemin du `hostPath` du volume `etcd-data` pour mettre celui du backup. Puis on fait `systemctl daemon-reload` et `service etcd restart`. Enfin, on démarre l'`api server` avec `service kube-apiserver start`.

- Quand on effectue le `save`, il est important de préciser à nouveau les certtificats avec les flags `--endpoints`, `--cacert`,`--cert`, `--key`.
- Quand on effectue le `restore`, il est important de préciser le flag `--data-dir` pour le faire pointer sur le dossier contenant le snapshot

On préfère privilégier la première méthode.

**informations de démarrage d'`etcd`** : La meilleure manière de voir comment est lancé l'`etcd` (pod en local ou externe) est de describe le pod de l'`API server`, et d'observer les flags.

**Observer les serveurs qui composent le cluster `etcd`** : utiliser la commande `etcdctl member list`. Il faut préciser la vrsion de l'`API` (cf plus haut) avec les flags `--endpoints`, `--cacert`,`--cert`, `--key`.

Dans le cas d'un etcd externe, on peut ssh sur le serveur qui l'héberge, et observer les paramètres avec lesquels il a été lancé via la commande `ps -aux`

## Security

Toutes les communications entre le `kube api server` central et les autres composants (`controller manager`, `etcd cluster`, `kube proxy`...) sont chiffrés avec TLS (via certificats).

Le besoin de sécurité dans `Kubernetes` repose sur le fait que l'`api server` permet d'effectuer tout ce que l'on veut sur le cluster. Ainsi, il est nécessaire de restreindre les utilisateurs qui y ont accès, et limiter leurs droits.

### Authentification

Plusieurs types d'utilisateurs peuvent être amenés à intéragir avec le cluster :

- les admins
- les développeurs
- les utilisateurs finaux
- les bots

Il existe plusieurs mécanismes d'authentification :

- **Fichier statique de mots de passe** (non recommandé) : Fichier simple avec 3 colonnes, mot de passe, nom d'utilisateur, ID de l'utilisateur. Le fichier est passé en paramètre à l'`API server` via le flag `--basic-auth-file`. Il est aussi possible de modifier le manifest `yaml` de l'`API server` si l'install est faite via `kubeadm`.
- **Fichier statique de tokens** (non recommandé): A l'instar du fichier statique de mots de passe, le `user` et son `token` sont stockés en clair
- **Certificats** (TLS) : Voir section suivante
- **IDP** (Identity provider) externe comme `LDAP`, `Keycloak`...

**TLS**

- Un certificat permet d'établir la confiance lors de l'échange entre deux entités
- Le serveur génère sa paire de clé publique et privée. En pratique, cela peut être fait avec `openssl`. La clé privée est contenue dans le fichier `.key` ou `.pem`, et la clé publique dans le fichier `.pem` ou `.crt` (ce fichier est le certificat).
- Avec `HTTPS/TLS`, la clé publique est en fait contenue dans un certificat. Le serveur cherche à signer ce certificat pour attester de l'authenticité de ce dernier, et donc de la clé publique qu'il transmettra aux utilisateurs pour ensuite permettre le chiffrement asymétrique.

  Il crée donc une `CSR` (Certificate Signing Request) et l'envoie à une `CA` (Certificate Authority). Cette `CA` est la référence de sureté, une entité tierce (publique comme `Digicert`, `Symantec` ou privée en mettant soi même sa `CA` en place) qui **vérifie les informations du certificat** (nom de domaine, numéro de série, DNS, date de validité...) avant de signer la `CSR` avec sa clé privée (celle de la `CA`, voir paragraphe suivant), pour donner un certificat valide au serveur. Cette vérification permet d'empêcher qu'un serveur malhonnête puisse envoyer des certificats valides.

  Le mécanisme de communication avec les `CA` passe aussi via un chiffrement asymétrique. Les navigateurs intègrent les clés publiques de ces `CA` pour pouvoir communiquer de manière sécuriser avec ces `CA`, notamment pour envoyer la `CSR`.

  On peut générer une `CSR` avec `openssl` à partir de la clé privée `.key` (qui en réalité sert à en extraire la clé publique qui sera dans la `CSR` et donc dans le certificat).

  Le certificat signé est ensuite envoyé à l'utilisateur, qui l'utilisra (donc la clé publique contenue dedans) pour chiffrer ses messages avec, et communiquer en sécurité avec le serveur.

  Il existe également ce mécanisme du côté utilisateur, pour garantir au serveur que l'utilisateur est authentique (et pas un hacker par ex). Le

client génère un certificat avec une `CSR` qu'il fait signer par une `CA` avant de parler au serveur. Cependant, ces opérations côté client sont gérées automatiquement et ne nécessitent pas d'actions utilisateurs.

Tout ce processus qui vient d'être décrit s'appelle une `PKI` **(Public Key Infrastructure)**.

- Puisque ce processus existe côté serveur et client, on distingue souvent leurs noms avec :
  - Pour les certificats (clé publique) :
    - `server.crt`
    - `server.pem`
    - `client.crt`
    - `client.pem`
  - Pour les clés privées :
    - `server.key`
    - `server-key.pem`
    - `client.key`
    - `client-key.pem`
- Conventionnellement, s'il n'y a pas le mot clé `key` dans le nom, alors c'est un certificat (clé publique), si c'est le cas, alors c'est une clé privée
- Il est possible de signer soi même un certificat, et n'apporte donc aucune garantie de sécurité

**TLS dans Kubernetes**

- On différencie 3 types de certificats :
  - **Client certificates** : Clé privée et certificat (clé publique) générés par le client pour assurer son identité auprès du serveur
  - **Serving certificates** : Clé privée et certificat (clé publique) générés par le serveur, dont le certificat est celui qui sera signé par la `CA`
  - **Root certificates** : Clé privée et certificat (clé publique) générés par la `CA` pour garantir son authenticité notamment au moment de signer les `CSR`
- Chaque composant de Kubernetes qui prend la forme d'un serveur doit avoir sa paire clé privée/certificat. Parmi ces composants, on peut notamment citer :
  - `kube-api server`
  - `etcd server`
  - `kubelet server`
- On cite aussi les clients d'un cluster qui communiquent avec ce dernier, qui ont également leur paire:
  - l'administrateur du cluster (utilisateur de `kubectl`..)

  - `kube-scheduler` qui communique à l'`api server`
  - `kube-controller-manager` qui communique à l'`api server`
  - `kube-proxy` qui communique à l'`api server`

- Tous ces certificats doivent être signés, il faut donc une `CA`. Cependant, Kubernetes exige une `CA` spécifique uniquement à l'`etcd` et une autre pour le reste des services. Pour le moment, on ne considère qu'une seule et unique `CA`, qui a sa paire clé privée/certificat (clé publique)
- Générons la paire de clés de la `CA`:

  ```
  openssl genrsa -out ca.key 2048
  ```

  puis la `CSR`

  ```
  openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr
  ```

  puis signons la `CSR`

  ```
  openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt
  ```

- De manière analogue, on peut générer toutes les autres paires. La seule différence est la signature. Admettons que l'on veuille établir la paire `admin.key`/`admin.crt` pour l'administrateur du cluster. Les deux premières commandes sont les mêmes, en changeant juste le nom. La commande de signature sera :

  ```
  openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt
  ```

- En pratique, on ne génère pas toutes ces paires soi même, les outis d'aide comme `kubeadm`, ou les distrib comme `k3s` ou `rke2` le font automatiquement
- La commande

  ```
  openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
  ```

  permet de visualiser ce que contient le certificat
- En describant les pods, on peut visualiser le chemin des certificats qui sont utilisés

### Certificates API

- Les logiques de signatures par la `CA` sont faites dans le `controller manager`. On y trouve notamment deux composants `CSR-APPROVING` et `CSR-SIGNING`
- Il existe une `Certificates API` dans le control plane, qui permet de gérer les actions de `CA` automatiquement (approbation de la `CSR`...).
- Pour utiliser cette API, il faut utiliser la ressource `CertificateSigningRequest`. L'utilisateur crée sa clé privée, puis génère sa `CSR` à partir de cette dernière. On encode en base 64 cette `CSR` et on met le résultat dans le champ `request` de la ressource. L'utilisateur apply ensuite sa ressource.

- L'administrateur peut voir les csr créées avec `kubectl get csr`, et les approuver avec `kubectl certificate approve <nom-du-certificat>`. L'utilisateur peut alors récupérer ce certificat (`kubectl get csr <nom-du-certif> -o yaml` puis decoder le certificat depuis base64) et l'utiliser pour communiquer avec l'`API server`
- Il est possible de requêter l'`API server` via HTTP avec `curl` par exemple, en précisant les certificats avec lesquels sont faits la requête. ex : `bash curl https://my-kube:6443/api/v1/pods \ --key admin.key --cert admin.crt --cacert ca.crt`

**Kubeconfig**

- Ces certificats et clés pour discuter avec l'`API server` sont présents par défaut dans le `~/.kube/config`
- `kubectl` va d'abord chercher dans le path `~/.kube` pour essayer d'y trouver un fichier `config`. Il est possible de préciser un chemin vers un autre `kubeconfig` en paramètre
- Exemple de fichier kubeconfig :

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://192.168.121.215:6443
name: default
contexts:
- context:
    cluster: default
    user: default
    namespace: default
name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

- Dans un fichier `kubeconfig`, on distingue 3 champs particuliers :
  - `clusters` : répertorie les différents clusters qui existent
  - `contexts` : définit les règles d'accès aux clusters par les utilisateurs. Chaque contexte indique quel user a accès à quel cluster. Le sous-champ `namespace` permet de définir le namespace par défaut du

contexte.
  - `users` : répertorie les différents users des clusters

Il est possible de changer de contexte en utilisant

`kubectl config use-context <context>`

- Comme dit dans la section précédente, on peut requêter l'`API Rest` de l'`api server` via :

  ```
  curl https://my-kube:6443/api/v1/pods \
  --key admin.key
  --cert admin.crt
  --cacert ca.crt
  ```

  Dans le cas de l'utilisateur par défaut, on peut notamment trouver les clés et certficats nécessaires das le `kubeconfig`.

- Sinon, il existe la commande `kubectl proxy` qui met en place un serveur proxy qui utilise les certficats et clés par défaut du `kubeconfig` lors du `curl` sur ce serveur proxy. Ainsi, il n'est plus nécessaire de devoir préciser les certificats à chaque requête `curl`.

### Authorization

- Différentes manières de gérer les authorisations :
  - **Accès interne au cluster** `Node authorizer` : Comme vu précédemment, les certificats peuvent avoir des groupes (ex : `system:...`), et le `Node authorizer` sait alors quel droit donner à l'utilisateur selon son certificat
  - **Accès externe au cluster** `ABAC` (Attribute Base Access Control) : utilise un fichier `policies.json` dans lequel sont définies les politiques d'accès (`user`, `namespace`, `resource`...). Compliqué à gérer car il faut éditer manuellement le fichier à chaque modification.
  - **Accès externe au cluster** `RBAC` (Role Base Access Control) : On définit des `Roles`, par exemple le rôle `developer`, qui peut créer des pods dans un namespace particulier. Alors, on a juste à attribuer ce rôle à l'utilisateur pour restreindre ses droits. Cela permet aussi de juste avoir à modifier le rôle pour faire évoluer les droits.

- Le mode d'autorisation se set au lancement de l'`api server` via le flag

  `--authorization-mode=None,RBAC,Webhook`

  Les modes d'authorisation sont traités sequentiellement, si un mode refuse l'authorisation, alors le suivant est utilisé

### RBAC

- Un `Role` est une ressource à part entière de Kubernetes et ressemble à

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

Cependant, un `Role` n'est associé qu'à un seul namespace. Si on souhaite définir un rôle qui touche à l'**ensemble** du **cluster**, alors on préfèrera la ressource `ClusterRole`. Autre cas dans lequel le `ClusterRole` est nécessaire, si on souhaite donner un droit d'action sur les `nodes` ou les `pvs`, qui sont des ressources non cloisonnées dans un `namespace`.

- Ensuite, on crée un `RoleBinding` ou `ClusterRoleBinding`, qui associe le `Role` au `User`

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: pod-reader-binding
    namespace: mynamespace
subjects:
    - kind: User
      name: "user1"
      apiGroup: rbac.authorization.k8s.io
roleRef:
      kind: Role
      name: pod-reader
      apiGroup: rbac.authorization.k8s.io
```

- `kubectl get roles` permet de voir les `Roles` et `kubectl get rolebindings` permet de voir les `RoleBindings`. On peut `describe` ces ressources pour avoir plus de détails.
- Il est possible de tester les actions sur des ressources via la commande, par exemple,

```
kubectl auth can-i create deployments --as user1 --namespace test
```

### Service Accounts

- On distingue les `User Accounts` et les `Service Accounts`
  - Les `User Accounts` sont les comptes utilisateurs, qui correspondent aux personnes humaines qui intéragissent avec le cluster (admin,

développeur...). C'est ce qu'on a utilsié pour les `RBAC` dans la section précédente
  - Les `Services Accounts` s'adressent aux programmes externes qui ont besoin d'intéragir avec le cluster

- Il est possible de créer un token d'accès pour un `Service Account` avec la commande

  `kubectl create token <sa-name>`

- Les token pour les `Service Accounts` peuvent être montés en volume via `Secrets`

### Image Security

- On peut utiliser un regsitry privé pour utiliser des images pour les pods
- On peut créer un secret qui contient les credentials de la registry pour permettre aux pods d'y accéder

```
kubectl create secret docker-registry private-reg-cred \
  --docker-username=dock_user \
  --docker-password=dock_password \
  --docker-email=dock_user@myprivateregistry.com \
  --docker-server=myprivateregistry.com:5000
```

- On ajoute ensuite ces champs dans le manifests du pod/deployment

```
spec:
    containers:
      - name: my-container
        image: myprivateregistry.com:5000/your-image:latest
    imagePullSecrets:
      - name: private-reg-cred
```

### Kubernetes Security

- On peut deféinir des règles de sécurité dans le `CRI` (`docker`, `containerd`...), comme le `user ID` qui exécutera le conteneur. On préfère éviter que les conteneurs soient exécutés en `root` pour améliorer la sécurité.

- De plus, on peut ajouter ou retirer des privilèges sur les actions qu'un conteneur peut faire sur l'hôte via le flag `--cap-add` ou `--cap-drop`

- On peut également effectuer ces actions dans le manifest `yaml` du pod, dans le champ `containers`

- ex :

  ```
  apiVersion: v1
  kind: Pod
  ```

```
metadata:
name: mypod
spec:
containers:
- name: mycontainer
    image: myprivateregistry.com:5000/your-image:latest
    securityContext:
        capabilities:
            add: ["MAC_ADMIN"]
```

### Network Policy

- On peut mettre en place une `Network Policy` pour restreindre les traffis entrants dans un pod. Par exemple, dans une app web, le front n'a pas besoin de communiquer avec la base de données, et le fait que ce traffic soit autorisé (configuration par défaut de Kube, tous les pods peuvent contacter tous les pods) représente une faiblesse de sécurité. On peut alors définir quels ports doivent être ouverts et quels traffics sont autorisés
- On différencie un `Ingress` d'un `Egress` par leur sens de communication. `Ingress` désigne le flux entrant, `Egress` le flux sortant.
- On associe une `NetworkPolicy` à des `pods` via leur label. ex :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
    name: mysql-network-policy
spec:
    podSelector:
        matchLabels:
            app: example
    policyTypes:
    - Ingress
    - Egress
    ingress:
    - ports:
        - port: 3306
        from:
        - podSelector:
            matchLabels:
                role: frontend
    egress:
    - ports:
        - port: 3306
        to:
        - podSelector:
```

```
        matchLabels:
            role: backend
```

- Au lieu de `podSelector`, on peut aussi utiliser le champ `namespaceSelector` pour autoriser le traffic dans tout le namespace, ou alors `ipSelector` pour autoriser le traffic depuis/vers cette `IP`.

### Volumes

- Les volumes sont des entités qui permettent de partager des données entre un pod et le noeud hôte. En pratique, on fait correspondre un dossier local à un dossier dans le conteneur/pod. Tout ce qui est écrit dans l'un des deux dossiers, est répliqué dans l'autre. Les données sont alors persistentes et permettent à un pod détruit, de se créer en conservant ses données.
- `Persistent Volume` : Les `Persistent Volumes` sont des volumes qui sont disponibles dans tout le cluster (pas liés à un namespace), dans lesquels ont peut définir la taille du stockage alloué, le mode d'accès etc...

  ```
  apiVersion: v1
  kind: PersistentVolume
  metadata:
      name: my-pv
  spec:
      capacity:
          storage: 1Gi
      accessModes:
          - ReadWriteOnce
      hostPath:
          path: /path/on/host
      persistentVolumeReclaimPolicy: Retain
  ```

  Le champ `hostPath` n'est pas à utiliser en production, puisqu'on préfère un stockage distribué.
- `Persistent Volume Claim` : Les `PVC` sont la ressource qui permet d'associer un pod à un `Persistent Volume`. Le `PVC` est bound au `PV` le plus adapté au besoin renseigné dans le `PVC` et respecte ses contraintes (`label`...)

  ```
  apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
      name: my-pvc
  spec:
      accessModes:
          - ReadWriteOnce
  resources:
  ```

```
    requests:
        storage: 1Gi
```

- On utilise le `PVC` dans le manifest du `pod/deployment` comme suit:

```
spec:
    containers:
        - name: myfrontend
          image: nginx
          volumeMounts:
          - mountPath: "/var/www/html"
            name: mypd
    volumes:
        - name: mypd
          persistentVolumeClaim:
            claimName: myclaim
```

### Storage Classes

- `StorageClass` est une ressource qui va permettre de gérer automatiquement les `PV` du cluster. Dans le manifest du `PVC`, on précise le champ `storageClassName` qui va sélectionner la `StorageClass` précedemment créée. Cette `StorageClass` est liée à un provisionner (cloud ou local) qui va créer les `PV` nécessaires demandés par les `PVC`. Ex de `StorageClass` :

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: my-storage-class
provisioner: kubernetes.io/hostpath
volumeBindingMode: Immediate
```

et le `PVC` qui la sélectionne :

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: my-pvc
spec:
    storageClassName: my-storage-class
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
        storage: 1Gi
```

## Networking

### Pre-requesites

`Switch, routeur & gateway`

- Un `switch` permet de relier des machines **au sein d'un même réseau**. Un `switch` a une adresse IP avec un masque, et permet la communication entre des machines dont l'IP est comprise dans ce réseau
- Chaque machine doit avoir une interface réseau pour se connecter à ce réseau (pour obtenir l'IP dans le réseau)
- Un `routeur` permet de relier **deux réseaux entre eux**. Il possède deux IPs, qui appartiennent à chacun des deux réseaux qu'il interconnecte.
- Cependant, juste avec un `routeur` qui relie les IP, le réseau A ne peut pas encore communiquer avec le réseau B. Il faut préciser au réseau A qu'il peut atteindre le réseau B via l'IP du réseau A associée au routeur. Cela s'appelle une `Gateway`
- Exemple `Gateway` :

  `ip route add 192.168.2.0/24 via 192.168.1.1`

  On fait cette commande dans le réseau 1, qui saura qu'il faut passer par la `Gateway 192.168.1.1` (IP du réseau 1 associée au routeur côté réseau 1) pour atteindre `192.168.2.0/24` (réseau 2). De manière analogue, on fait la même chose dans le réseau 2.

  `ip route add 192.168.1.0/24 via 192.168.2.1`

  `192.168.2.1` étant l'IP associée au routeur côté réseau 2 (la `Gateway` du réseau 2).
- On peut donc maîtriser tout ce qui sort de chacun des réseaux. Par exemple, si on veut pouvoir ping `dns.google.com` depuis le réseau 2, il faudrait faire

  `ip route add 8.8.8.8 via 192.168.2.1`
- Au lieu de préciser toutes les routes vers internet, on préfère faire

  `ip route add default via 192.168.2.1`

  Ce qui a pour effet de faire passer par la `Gateway 192.168.2.1` toutes les requêtes dont l'IP de destination n'est pas dans les autres routes du routeur.
- On peut visualiser la table de routage via la commande

  `route`
- La `Gateway 0.0.0.0` indique au réseau qu'il n'a pas besoin de `Gateway` pour atteindre le réseau correspondant.

- On préfère souvent scinder les rôles des routeurs. Par exemple, un routeur qui permet l'accès à internet, et un autre routeur qui permet l'interconnexion du réseau 1 au réseau 2. Pour ce faire, puisqu'on rajoute un routeur, on rajoute une `Gateway` pour chaque réseau qui accèdera à ce nouveau routeur.

**Machine Linux as routeur**

- Il est possible d'attribuer plusieurs interfaces réseau à une machine. Elle peut ainsi être connectée à deux réseau différents.
- Soit une machine A (`192.168.1.5`), une machine B (`192.168.1.6` et `192.168.2.6`) et une machine C (`192.168.2.5`). Soit un réseau A (`192.168.1.0/24`) et un réseau B (`192.168.2.0/24`). Si on veut que A puisse parler à C, et ce sans routeur, alors on peut ajouter une route qui passe par la machine B :

  ```
  ip route add 192.168.2.0/24 via 192.168.1.6
  ```

  Si on veut que la communication soit possible dans l'autre sens, alors :

  ```
  ip route add 192.168.1.0/24 via 192.168.2.6
  ```
- Pour obtenir la réponse d'un `ping` en utilisant la machine B comme routeur, il faut autoriser la redirection entre interfaces réseau (puisque la machine B est reliée au réseau A et B par deux interfaces différentes). Pour ce faire, il faut set à 1 le fichier `/proc/sys/net/ipv4/ip_forward`.

**DNS**

- Un `DNS` est un serveur qui permet la résolution de nom de domaine. Il a un format paire clé-valeur et associe une adresse IP à un nom de domaine.
- Sur une machine individuelle, on peut répliquer ce fonctionnement dans le fichier `/etc/hosts`. La résolution ne se fait que sur la machine en local.
- Avec un serveur `DNS` dédié, il faut indiquer à chaque machine de s'y référer pour résoudre les noms de domaine. Pour ce faire, on renseigne l'entrée suivante dans le fichier `/etc/resolv.conf` :

  ```
  nameserver IP_DNS
  ```

**Network Namespaces** (vidéo à revoir si nécessaire car complexe)

- On peut créer un namespace de réseau avec la commande

  ```
  ip netns add example
  ```
- Si ce namespace de réseau marche, alors en lisant les interfaces réseau, on devrait voir uniquement celle(s) associée(s) au namespace, et pas celle(s) de l'hôte.

`ip -n example ip link`

(ça marche)

- Les conteneurs ne peuvent donc pas voir le réseau exterieur au namespace
- On ajoute une interface réseau virtuelle au namespace avec l'adresse IP

  `ip -n example addr add 192.168.1.10/24 dev veth-example`

- Le `NAT` (Network Address Translation) mappe des addresses IP privées vers une seule adresse IP publique. Cela permet à plusieurs machines au sein d'un réseau privé de partager une adresse IP publique commune pour accéder à internet.

**Docker Networking**

- Le type de réseau le plus intéressant de docker est `bridge`.
- Le `bridge` est un sous réseau sur l'hôte que les pods rejoignent avec leur propre IP.
- L'hôte a une interface réseau interne `docker0` créée par défaut. Elle agit comme un pont entre les interfaces réseau des conteneurs et celle de l'hôte. Elle est associée au réseau `bridge`.
- Quand un conteneur est créé, `Docker` crée un namespace réseau pour ce conteneur seul
- Pour que ce conteneur, et donc ce namespace, rejoigne le réseau `bridge`, `Docker` crée une interface réseau virtuelle côté `bridge` (`vethxxxxxx`) et une autre côté `namespace` (`eth0@xxxx`), qu'il relie entre eux.
- Pour que le conteneur/namespace soit accessible depuis l'extérieur, on utilise le `port-forwarding`. Au moment du lancement du conteneur, on précise `8080:80` par exemple, qui mappe le port `8080` de l'hôte au port `80` du conteneur. Ainsi, on peut accéder au service du conteneur en faisant `IP_HOTE:8080`.
- En pratique, cette opération est faite via `NAT` grâce à `iptables` qui ajoute l'assocation `80/8080`. En somme, docker effectue :

  `iptables -t nat -A PREROUTING -j DNAT --dport 8080 --to-destination BRIDGE_CONTAINER_IP`

**Cluster Networking**

- La commande `arp` permet d'avoir l'adresse `MAC` associée aux adresses `IP` connues de la machine
- `netstat -npl | grep -i <process_name>` permet de voir sur quel port écoute le processus

### CNI

- Une `CNI` (Container Network Interface) est l'interface qui définit la manière dont les conteneurs sont connectés, isolés et configurés dans le cluster. Il existe différents plugins `CNI` : `Weave`, `Flannel` `Calico`, `Cilium`...
- Il est possible de voir comment déployer chaque plugin `CNI` dans la doc de Kubernetes. Les plugin prenennt la forme de `DaemonSet` pour qu'ils soient créés/détruits avec chaque noeud.
- On peut trouver les informations du cni utilisé dans `/etc/cni/net.d` sur les noeuds

### Ingress

- Les services (même `NodePort`) ne sont pas adaptés pour un accès extérieur. On souhaite éviter d'avoir à préciser l'IP et le port du service lorsque l'on accède à notre application.
- Le moteur de la solution `Ingress`, qui permet de rediriger et `Loadbalancer` le traffic vers les différents service à l'intérieur du cluster, est l'`Ingress Controller`. Il n'y en a pas par défaut, il faut l'installer.
- Il existe différents `Ingress Controller`, comme `nginx`, `haproxy`, `traefik`...
- Plusieurs ressources accompagnent l'`Ingress Controller` : `Service Account`, `Roles`, `RoleBindings`
- La ressource qui permet de configurer cette solution est la ressource `Ingress`.
- Un `Ingress` joue le rôle d'un reverse proxy.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: example-ingress
    namespace: service1-service2-same-namespace
spec:
    # Permet de réécrire l'url une fois redirigée sur le service => ne conserve pas le
    nginx.ingress.kubernetes.io/rewrite-target: /
    rules:
        - host: example.com
            http:
                paths:
                    - path: /route1
                    pathType: Prefix
                    backend:
                        service:
                            name: service1
```

```
            port:
            number: 80
    - path: /route2
    pathType: Prefix
    backend:
        service:
            name: service2
            port:
            number: 80
```

- Les ressources `Ingress` sont namespacées. Ainsi, si on veut rediriger vers un service dans un autre namespace que les autres, il faut créer un `Ingress` dans ce namespace

### Design & Install a Kubernetes Cluster

### Design a Kubernetes Cluster

- `Write Quorum` correspond au nombre minimum de noeuds qui doivent être disponibles pour assurer la `HA` en mode écriture. Ce `Quorum` correspond à `Write Quorum = (N/2)+1` Ainsi, avec un cluster de 5 noeuds en `HA`, l'écriture d'un objet est assurée tant qu'il y a `|_(5/2) + 1_| = |_3.5_| = 3`. On a ainsi une `Fault tolerance` de 2 noeuds.
- Pour un cluster `etcd` par exemple, il faut avoir un nombre satisfaisant de noeuds master (puisque ce sont eux qui portent les instances de `etcd`) pour assurer un `Quorum` satisfaisant .

### Troubleshooting cluster

### Application Failure

- `curl` sur le service de l'application
- `kubectl describe` le servcice, vérifier que le `Selector` correspond au label du `pod`
- `kubectl get pod` permet de voir les restarts d'un pod et son statut
- `kubectl describe pod` pour voir els `logs` du pod
- `kubectl logs <pod_name> -f` pour voir les logs du conteneur du pod
- On répète ces étapes pour les applications dont dépend celle qu'on veut débugger. Ex : si c'ets une app web, après l'avoir debug en elle-même, on debug le pod et service de la base de données associées...

### Controlplane Failure

- Sur les noeuds `master`, on peut troubleshoot les différents composants selon la manière par laquelle ils sont utilisés.
- S'ils sont déployés en tant que `pod`, on peut les débugger comme des pods usuels (`describe`, `get`, `logs`..)

- S'ils sont déploéys en tant que services, on peut inspecter leur statut avec `systemctl status kubelet` ou plus simplement `service kube-controller-manager status`
- On peut faire de même pour les composants des noeuds `workers`, pour `kubelet` et `kube-proxy`
- Ne pas oublier que les pods des composants du `controlplane` sont définies comme des pods statitques dans `/etc/kubernetes/manifests`

### Worker Node Failure

- `kubectl describe node`
- `top h` pour voir conso ressources
- `df -h` pour voir stockage
- `service kubelet status`
- `journalctl -u kubelet`
- `openssl x509 -n /var/lib/kubelet/node.crt -text` pour voir les détails du certificat

### Json Path

- Il est possible d'obtenir davantage d'informations sur les ressources en ajoutant le flag `--output=json`. Par exemple, `kubectel get pod <pod_name> --output=json | jq` permet de visualiser la sortie au format json.
- il est possible de n'extraire que certaines informations et de reformater la sortie de la commande en opérant sur ce json path, comme suit : `kubectl get nodes -o jsonpath='{.items[0].status.nodeInfo.architecture}'`