Formation k8s

dépôt github

- https://github.com/lamamram/k8s-init.git
- machine phys >= 8Go
- 2+ CPU
- install vb 7 + vagrant récent

configuration de l'install Kind

1. adresses physiques des noeuds : `docker network inspect kind --format '{{range .Containers}} {{ .IPv4Address }} {{ end }}'`
2. configuration par défaut sans CNI mais inutilisée `k get nodes -o jsonpath='{.items[*].spec.podCIDR}'`
3. configuration du pool d'ip virtuelles iée au CNI calico avec encapsulation IPIP `k get ippools.crd.projectcalico.org -o jsonpath='{.items[*].spec.cidr}'`

vérification de l'install

- `k cluster-info`: voir l'adresse et le port où requêter sur le serveur d'API de k8s (même si on utilise en fait un proxy)
- `k cluster-info dump`: configurations et logs de tous les `pods`, `kubelet`, `kube-proxy`: exhaustif mais difficile à requêter

ressource élémentaire dans un cluster k8s: POD

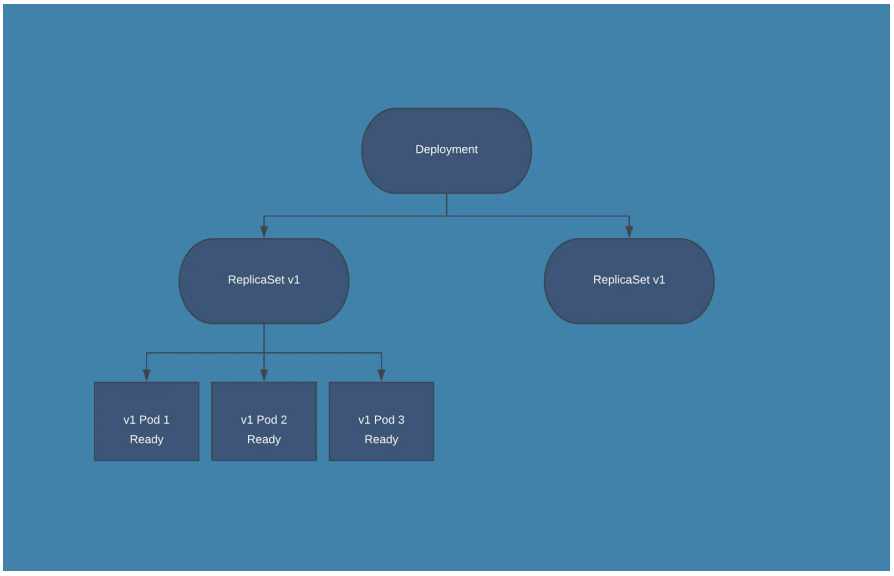
- image de test bob2606/hello-http:0.8.9
- test d'un pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: busy2
    truc: machin
  name: busy2
  namespace: test
spec:
  containers:
  - image: busybox:stable
    name: busy2
    command:
      - sleep
      - infinity
  - image: bob2606/hello-http:0.8.9
    name: hello
  dnsPolicy: ClusterFirst
  restartPolicy: Always
```

la ressource centrale dans un cluster k8s: le Deployment

- génère le squelette dans un fichier
  - `k create deployment hello --image bob2606/hello-http:0.8.9 --dry-run=client -o yaml > hello-deploy.yml`
  - dry-run=client : simule & retourne un manifeste yaml côté client i.e requête HTTP REST
  - dry-run=server: simule & retourne un manifeste yaml côté serveur i.e la réponse HTTP REST
  - dry-run=none: ne simule pas !!!! mais retourne le manifeste server

RollingUpdate: illustration

- 

problématique des Services

1. les pods sont transitoires (IP et hostname)

- utilisation du FQDN au lieu de l'IP 10.32.73.145. `k exec busy -- wget -O - http://10-32-73-145.default.pod.cluster.local:8080` (hostname aussi transitoire que l'IP)

2. de stabiliser l'ip et du hostname de notre appliquée déployée sur k8s: notion de service

- pas d'exposition à la docker (Directive EXPOSE du Dockerfile)
    - pour faire remonter cette information dans k8s on utilise la clé **containerPort** dans le déploiement

```
containers:
    - name: ...
      ...
      ports:
          - name: http
            containerPort: 8080
            protocol: TCP
```

- squelette: `k expose deployment hello --port 80 --target-port 8080 --dry-run=client -o yaml > hello-svc.yml`
    - REMARQUE: mauvais choix de nom commande car **expose** est dans k8s est analogue de la **publication** de docker i.e l'option -p EXTERN_PORT:INTERN_PORT de docker et pas **exposition** de docker

    - la clé **.spec.port** du service k8s désigne le port d'entrée du service (clusterIP)
    - la clé **.spec.targetPort** du servce k8s sésigne le port effectifs du pod sous jacent

3. vision globale du système déploiement + service:
    - `k get deployments.apps,pods,svc`
    - voir tous les services indépendamment du namespace: `k get -A svc`
    - voir toutes les ressources d'un namespace: `k get all [ -n <namespace_name> ]`

test du clusterIP:

```
* `k exec busy -- wget -O - http://<CLUSTER-IP>`
   - ou `k exec busy -- wget -O - http://hello` avec le hostn
  - ou `k exec busy -- wget -O - http://hello.default.svc.clu
* le réseau configuré pour les services est dispo avec cette
   - `k cluster-info dump | grep -m 1 service-cluster-ip-rang
```

NodePort

- ajouter ou remplacer la clé `type: NodePort` dans le manifeste du service
- test du nodeport: attaquer l'ip d'un noeud sur le port spécifier par nodeport
    - `wget -O - http://172.18.0.2:32400`

LoadBalancer

- le service LoadBalancer n'est pas un LoadBalancer !
- c'est pourquoi un service de type LoadBalancer exécuté sans avoir une ressource LoadBalancer reste en statut

installation du loadBalancer Metal LB dans le cluster

- `kubectl apply -f https://raw.githubusercontent.com/metallb/metallb/v0.13.7/config/manifests/metallb-native.yaml`
- https://github.com/metallb/metallb

- on va regarder les pods dans le namespace metallb-system
    - `k get -n metallb-system pods`
- nous allons exécuter la configuration metal LB en couche 2 (LAN)
    - ajout d'un "pool d'addresses externes" sur le loadBalancer pour répartir le flux entrant sur les noeuds

```yaml
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: example
  namespace: metallb-system
spec:
  addresses:
  - 172.18.255.200-172.18.255.250
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: empty
  namespace: metallb-system
```

- on peut attaquer notre déploiement depuis le port standard 80 du loadBalancer et l'adresse ip externe associée au service LoadBalancer associé à notre déploiement
- le flux va taper sur un des ports 30k + du nodePort
- il va communiquer sur l'IP du clusterIP
- va répartir la charge sur un des pods répliqu"s du déploiement
- avec un tunnel ssh

```
ssh -L 8888:172.18.255.200:80 vagrant@127.0.0.1 -p 2222 -Nf -i "C:\Users\Admin stagiaire.DESKTOP-8967908\.vagrant.d\insecure_private_keys\vagrant.key.rsa"
```

- curl http://localhost:88888

TP déploiement

1. créer un déploiement d'un pod contenant une image nginx:1.24 et un php:8.2-fpm
2. on va faire sortir le flux en entrée depuis le loadbalancer metallb
3. vérifier que le nginx soit accessible depuis l'extérieur du cluster

4. nommage

- nom du pod wwwphp
- nom des conteneurs
- nginx -> web
- php-fpm ->php

5. correction dans le github

volumes k8s

manip avec un volume hostPath

- on doit gérer le même dossier dans tous les nœuds
- on a un dossier .data déjà préconfiguer dans les noeuds
    - `docker cp <path/to/default.conf> kube-worker[2|3]:/data/default.conf`
    - docker cp /vagrant/170624/nginx_conf/default.conf kube-worker:/data/default.conf
- dans le déploiement, configurer le volume hostpath dans **.spec.volumes**
- configurer le point de montage dans **.spec.containers[0].volumeMounts[0]**

manip avec un serveur nfs

- installation

```
sudo apt-get update && sudo apt-get install -y nfs-kernel-ser
sudo mkdir /usr/local/partage
sudo vim /etc/exports
/usr/local/partage *(rw,no_root_squash,no_subtree_check,fsid=
## créer deux sous dossiers
sudo mkdir /usr/local/partage/conf_nginx
sudo mkdir /usr/local/partage/siteweb
## mettre le default.conf dans le dossier nginx et le index.p
sudo chown -R vagrant:vagrant /usr/local/partage
## rechargement du fichier
sudo exportfs -a
## redémarrer le service
sudo systemctl restart nfs-kernel-server
```

- configuration du déploiement avec un volume nfs : cf github
- test de la communication entre nginx <-> php dans le pod
    - `k exec wwwphp-xxxxx-xxxx -c www -- curl http://localhost/index.php`
- test depuis l'extérieur: j'ai besoin d'une requête avec un ServerName localhost
    - tunnel SSH
    - `ssh -L 8888:172.18.255.200:80 vagrant@127.0.0.1 -p 2222 -Nf -i "C:\Users\Admin stagiaire.DESKTOP-8967908\.vagrant.d\insecure_private_keys\vagrant.key.rsa"`
    - dans son navigateur `http://localhost:8888/index.php`

Env, ConfigMap, Secrets

Envs

clé **.spec.template.spec.sontainers[*].env[]**

```
name: USER
value: bob
```

- test `k exec wwwphp-574c4469b8-n9wrd -c www -- bash -c 'echo $USER'`

ConfigMap

`k create configmap wwwphp-nginx-cm --from-file nginx_conf/default.conf --dry-run=client -o yaml > wwwphp-nginx-cm.yml`

- REM: attention aux fins de ligne: k8s n'est pas le format windows \r\n (en tout cas dans un cluster unix)
- conversion d'un fichier CRLF en LF avant de générer
- REM: on peut créer des configMap multi file si les fichiers doivent être monté au même endroit
- intérêt du configMap vs le volume nfs direct

1. le nfs est une ressource extérieure et unique au cluster k8s => SPOF Single Point Of a Failure
2. le cm est ubiquitaire du fait que k8s peut le servir depuis n'importe quoi dpl/pod
3. est sécurisé par le cluster

- opérateurs YAM |- et |+ |-: Préserve les nouvelles lignes internes, mais supprime la nouvelle ligne finale. |+ : Préserve les nouvelles lignes internes et les nouvelles lignes finales.

utilisation d'u configMap de type clé / valeur comme chargeur de variables d'environnement

1. créer un fichier de type .env

```
USER=bob
```

2. créer le configMAp à partir du fichier env

```
k create configmap wwwphp-user-env --from-env-file .env --dry-run=client -o yaml > wwwphp-user-env.yml
```

3. configurer dans la clé **.spec.template.spec.containers[0].envFrom** (cf github)
4. test: `k exec wwwphp-694ff78bd8-k2tgk -c www -- bash -c 'echo $USER'` bob

REM; avec la clé prefix dans l'élément envFrom: ex: prefixe **NGINX_** `k exec wwwphp-64948974cc-k6nd8 -c www -- bash -c 'echo $NGINX_USER'`

utilisation de la valeur d'une variables dans une autre variable

1. avec **.spec.template.spec.env[*].valueFrom.configMapKeyRef**
2. configurer le déploiement (cf github)
3. tester `k exec wwwphp-ff9dcbcb9-kmpr7 -c www -- bash -c 'echo $ADMIN_FIRSTNAME'`

les Secret

- comme les configMap, mais pour les données sensibles

1. `k create secret generic wwwphp-secret-mdp --from-literal MY_PASSWORD=roottoor --dry-run=client -o yaml > wwwphp-secret-mdp.yml`

2. apply, get

REM: la valeur dans le manifeste **n'est pas chiffrée mais encodée en base64 **

- k8s n'a pas la responsabilité première de chiffrer
- donc on doit ajouter la fonction chiffrement avec des plugins
- ex Vault https://developer.hashicorp.com/vault/tutorials/kubernetes/kubernetes-raft-deployment-guide

3. configurer dans le déploiement avec enFrom[*].scretRef ...
4. on ne peut pas voir la valeur dans le describe MAIS dans le get -o yaml !!!!!!!!!

Kustomize

1ère étape

- 1. on peut rassembler nos manifestes YAML pour les lancer en un seul coup et unifier le nommage des ressources k8s

2. créer un dossier pour rassembler les ressources k8s
3. structure du yaml kustomization: (cf github)
    - utilisation de labels & annotations communs à toutes les ressources k8S
    - utilisation d'un namespace commun // (n'est pas créé !!!)
    - utilisation d'un préfixe commun //
4. test: `k kustomize`
5. lancement `k apply -k .` dans le dossier contenant le fichier kustomization.yml
6. test du kustomization:
    - k exec -n wwwphp base-wwwphp-67f5bf66f-j2tf2 -c www -- curl localhost/index.php
    - `k exec -n wwwphp base-wwwphp-588dbd9954-nb67c -c www -- bash -c 'echo "$ADMIN_FIRSTNAME"'`

2ème étape

- utilisation des configMapGenerator et secretGenerator pour générer directement les ressources k8s configMap et secret à partir des fichiers et / ou des littéraux

(cf github) & mêmes tests

micro TP

1. Déployer les ressources de l'environnement de dev

- espace de nom dev-wwwphp
- préfixe dev-
- réplicas 3
- image php:8.3-fpm
- configMap USER: mike

*SOLUCE (cf github)

- test: `k kustomize`
- regarder le `k get -k .` pour le nb de réplicas

statefulSet

démo

1. créer un statefulset
    - avec 3 réplicas
    - un volume emptyDir
    - ajoute un initContainer
    - ajouter 2 pionts de montages pour le container principal et auxiliaire
    - ajouter une commande différenciant le pod de terminaison -0 => master et les autres => worker
2. test
    - le conteneur init s'arrête après la commande donc il n'est existe plus
    - le pod hello n'a qu'un accès http
    - donc on a besoin d'un service k8s pour communiquer sur le port 8080 du pod hello
3. création d'un service ClusterIP headless (cf github)
    - donc ajoute la clé **clusterIP: None**
    - il faut ajouter la clé .**spec.serviceName** dans le statefulSet
        - WARNING: cette mise à jour n'est pas possible => suppression / création
4. test finalement

- on peut accéder au pod "-0" avec le nom d'hôte **<pod_name>-0.<headless_service_name>**
- pour voir la != de traitement effectuée au initContainer
- `k exec busy -- wget -O - http://hello-1.hello-set:8080/status`

pour aller plus loin (MYSQL repliqué)

https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

les Sondes

setup

- nginx-deploy-probe.yml
    - manifeste: composite
    - suppression collective: `k delete -f nginx-deploy-probe.yml`
- ajout des élements du nfs
    - index.html
- setup:

```
cp index.html /usr/local/partage/siteweb/
sudo exportfs -a
sudo systemctl restart nfs-kernel-server
```

- test: `k exec busy -- wget -O - http://nginx`

explications des sondes

1. startupProbe:
    - check l'état Running des conteneurs
    - donc il faut attendre la fin du démarrage
    - tant que la startupProbe n'a pas terminée en succès on checke pas les 2 autres sondes
2. livenessProbe

    - la commande lancée associée à cette sonde va impliquer un redémarrage si elle échoue

3. readiness

- la commande lancée associée à cette sonde va impliquer une désactivation du pod du service attenant
    - le pod n'est pas redémarré ou supprimé
    - mais on ne peut pas y accéder
- cas d'usage: un processus lent peut causer la dégradation du pod sans pour autant être une erreur
    - processus de maintenance / upgrade / n'importe quel processus lent

4. les sondes liveness et readiness sont indépendantes

test liveness probe

```
# au niveau de chaque conteneur
livenessProbe:
            command:
              - cat
              - /etc/nginx/nginx.conf
```

- 1er test:
    - OK mais pas d'évènements particulier dans le describe
    - par défaut la sonde est périodique (10s)
- 2ème test avec un exec faux chécké après 30s
    - on peut utiliser `k get pods -w` => rédamrrage après 30s

test startup probe

```
startupProbe:
            httpGet:
              path: /index.html
              port: 80
            initialDelaySeconds: 10
```

- 1er test:
    - ajout de la sonde en mode httpGet sur le path index.html et le port 80 (Vrai)
    - délai de 10s

    - la valeur périodique n'a pas de sens sur cette sonde => OK
- 2ème test startup + liveness
    - même startup
    - liveness faux au bout de 8s préuide 10s => la startup fonctionne à 10s => la liveness n'a pas été exécute à 8s parce qu'elle attendait un réponse de la startup => la liveness a checkée à ~18s à redémarrage => puis cycle startup / liveness => OK

test readiness probe

```
readinessProbe:
            tcpSocket:
              port: 8888
            failureThreshold: 3
            periodSeconds: 6
```

- test sur un port non écouté => faux on voir la désactivation et non le redémarrage et non suppresion => stuck !!!!!! => attention aux checks précis qu'on veut tester

communications inter pod / namespace / netpolicies

tester la communication inter pod via les namespace

- setup `k create ns stage`
- setup np-whoami-deployment.yml (cd github)
- test entre le pod dans le ns default et le pod whoami dans le ns stage => KO pour `k exec busy -- wget -O - http://whoami` : pour le hostname court du service whoami => **MAIS OK** `k exec busy -- wget -O - http://whoami.stage.svc.clust er.local` : pour le FQDN et les hostnames intermédiaires
- `k exec busy -- cat /etc/resolv.conf`

test la netpol "deny all"

np-deny-all.yml (cf github)

REM; ce deny-all est lié au namespace ciblé ou "defaut"

- `k apply, k get netpol`
- `k exec busy -- wget -O - http://whoami.stage.svc.cluster.local` => KO

règle allow-all-ingress

- permet tout flux entrant mais n'isole pas l'Egress

règle : np-whoami-ingress.yml (cg github)

- REM:
    - La sélection de la cible du flux entrant est liée au namespace de la netpol
    - La sélection de l'origine du flux entrant est EGALEMENT liée // PAR DEFAUT => pour modifier ce comportement par défaut, on va ajouter un namespaceSelector explicite pour spécifié le(s) namespace(s) qui sont éligibles à la selection aux labels précédentes
- REM: ajouter / supprimer des labels
    - `` `k label pods busy target=whoami` / `k label ``

Contrôleur d'ingress Nginx

*on utilise Helm pour ajouter la ressource k8s ingress-nginx dans notre cluster

- ajouter un dépôt tiers dans les dépôts canoniques helm (~apt-get) `helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx`
- rafraichir le cache de paquets `helm repo update`
- installer le paquet ingress-nginx/ingress-nginx du dépôt ingress-nginx `helm install ingress-nginx ingress-nginx/ingress-nginx`
- ajout de deux déploiement associées à leur clusterIP resp.
- ajout du controleur d'ingress (il faut la configuration loadBlancer metallb)
- manifeste (cf github)
- apply

- test : `curl -H host:hello-1.k8s.lan 172.18.255.200`
- tunnel SSH `ssh -L 8888:172.18.255.200:80 vagrant@127.0.0.1 -p 2222 -Nf -i "C:\Users\Admin stagiaire.DESKTOP-8967908\.vagrant.d\insecure_private_keys\vagrant.key.rsa"`
- dns externe (127.0.0.1 hello-1.k8s.lan)

gérer les ressources matérielles

1. installation du metrics-server + patch (tls-insecure) (slides)
2. utilisation des commandes k top nodes ou k top pods

- en dynamique `watch -n 1 kubectl top nodes` (pas k car collision avec watch)

réguler les ressources au niveau d'un pod

- on se donne un déploiement qui génère du tempsCPU (while 1) dans le ns resources
- on applique et on constate l'utilisation du CPU alloué à ce pod
- `k top -n resources pods`
- a priori aucune régulation
- on peut ajouter les clés **.spec.template.spec.containers[*].resources**
- ^pour établir les niveaux min max en cpu et memory d'un déploiement

réguler les resources au niveau du namespace

avec les resourcesQuotas

- ajout du manifeste resources-rq.yml (cf github)
- `k describe ns resources`

```
                                POD           Namespace
```

1. test limite cpu 750m 500m (KO) valeur max de l'addition des limites des pods/ctn du ns
2. test requête cpu 300m 200m (KO) valeur max de l'addition des requêtes des pods/ctns du ns

3. test sans ressources null qqch (KO) avec un RQ les ctns des pods **doivent définir** les seuils

- Attention à partir du moment où on a ajouté un RQ dans le NS => tous les pods doivent renseigner les métriques du RQ dans leur conteneurs

ajout d'une limitRange

- 1er effet: ajouter des seuils limite / requête par défaut pour les pods non pouvus de seuils dans leur définition, dans le namespace doté d'un RQ
- 2ème effet: contrôler les intervalles de seuils existants pour chaque pod du namespace

POD limitR

1. test bad request cpu 50m 100m (KO) il faut >= 100 (avec limit OK)
2. test good request cpu 100m 100m (OK) (avec limit OK)

effets différents entre limitRange et resourceQuotas

POD LR

1. r:100m l:250m 1 replica r:100m r:400m r: 200m l: 700m OK pour le LR et le RQ
2. r:100m l:250m 2 replica r:100m r:400m r: 200m l: 700m OK pour le LR (pour chaque ctn !) et le RQ (Total)
3. r:100m l:250m 3 replica r:100m r:400m r: 200m l: 700m OK pour le LR (pour chaque ctn !) mais KO pour RQ

précaution contre l'attague bomb fork

- ajouter la clé **podPidsLimit: 100**
- dans `docker exec -it kube-worker cat /var/lib/kubelet/config.yaml`
- `docker exec -it kube-worker bash -c 'echo "podPidsLimit: 100" >> /var/lib/kubelet/config.yaml'`
- `docker exec -it kube-worker bash -c 'systemctl restart kubelet'`

autoscaling

- contrôler le nb de réplicas par les ressources
- `k autoscale -n resources deployment load --min 1 --max 5 --cp --cpu-percent 66 --dry-run=client -o yaml > resources-hpa-load.yml`
- créer un déploiement avec 1 réplica
- créer le hpa (apply)
- REM: les seuils du HPA sont calculés à partir des ressources totales du cluster
- REM: les seuils acceptables du HPA peuvent ne pas être les mêmes que ceux du RQ , RL => dernières ressources qui brident le HPA

auth & RBAC

installation du client kubectl sur un machine et synchroniser sur un cluster : Sur Windows

1. Installation de kubectl => site k8s

- Création du répertoire ***appdata\k8s***
- Récupération du binaire depuis l'adresse "https://dl.k8s.io/release/v1.30.0/bin/windows/amd64/kubectl.exe"
- Copie du binaire kubectl dans le répertoire ***appdata\k8s***
- Modifier la variable d'environnement PATH pour y intégrer le répertoire appdata\k8

2. Configuration alias et complétion

- Dans une fenêtre PowerShell

```
kubectl completion powershell >> $PROFILE
echo 'Set-Alias -Name k -Value kubectl' >> $PROFILE
echo 'Register-ArgumentCompleter -CommandName k -ScriptBlock
. $PROFILE
```

3. Récupération du contexte kind-stage de la VM

- À la racine du répertoire personnel :
- `scp -r -p 2222 -i <pkey.rsa> vagrant@127.0.0.1:.kube .kube`
- Vérification

```
k config get-contexts
```

ajouter un "utilisateur" avec un certificat x.509

1. openssl

- Création de la clé privée
- openssl genrsa -out bob.key 2048
- Création d'une requête de certification
- openssl req -new -key bob.key -out bob.csr -subj "/CN=bob/O=podreader"
- Encodage base64 avec suppression des sauts de ligne
- `base64 bob.csr | tr -d "\n" > bob-csr.b64`

2. ressource k8s CertificateSigningRequest

- Après application du manifeste, on vérifie `k get csr`
- admin: `k certificate approve bob`
- l'approbation ajoute le contenu du crt dans l'objet csr
- `k get csr bob -o jsonpath='{.status.certificate}'` pour voir
- `k get csr bob -o jsonpath='{.status.certificate}' | base64 -d > bob.crt` : exporter le crt b64 décodé

3. côté client on a besoin du .crt et du .key

- création "user"
  - `k config set-credentials bob --client-certificate bob.crt --client-key bob.key`
- création context i.e: association cluster/user
  - `k config set-context bob-context --cluster kind-kube --user bob`
- changer de contexte
  - `k config use-context bob-contex`

4. test d'une commande dans le nouveau contexte

- `k get pods`: ERROR => car pas d'autorisation a priori
- `k auth can-i create pods`

procédure RBAC pour un "utilisateur"

1. changer de contexte sur l'admin **kind-kube**
2. création d'un rôle qui peut lire les pods dans un namespace ROLE
   - même chose dans la totalité du cluster: CLUSTERROLE
3. création d'un roleBinding qui va associer note rôle et un utilisateur (/CN= du csr)/groupe (/O= du csr) lié à un namespace

- même chose au niveau du cluster: CLUSTERROLEBINDING

4. appliquer , rechanger de contexte et retester

ajouter un compte de service: "serviceAccount"

1. créer un namespace

- `k create ns test`

2. constater qu'un objet serviceAccount de nom "default" est créé automatiquement

- `k get -n test sa`

3. on utilise pas le "sa" default par défaut (best practice)

```
k create sa www-sa \
--namespace test \
--dry-run=client -o yaml > test-www-sa.yml
```

4. deploiement de test : se donner

- une image exploitable (contenant curl) EX nginx
- un service clusterIP

5. exploiter le conteneur nginx

- `k exec -it <pod_name-xxx-xxxxxx> -c <ctn_name> -- bash`

6. découverte des éléments de communication vers le serveur d'API k8s

```
# mettre en cache le nom d'hôte du serveur d'API k8s en TLS
APISERVER=https://kubernetes.default.svc

# mettre en cache le chemin vers le dossier serviceAccount mo
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# mettre en cache le contenu du token JWT pour s'authentifier
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# mettre en cache le chemin vers le certificat de l'autorité
CACERT=${SERVICEACCOUNT}/ca.crt

# se connecter via curl sir l'endpoint /namespaces/test/pods
curl --cacert ${CACERT} \
     --header "Authorization: Bearer ${TOKEN}" \
     -X GET ${APISERVER}/api/v1/namespaces/test/pods
```

7. résultat: ERROR

- Parce que nous n'avons les **autorisations** de cet appel **GET pods**

procédure RBAC pour un compte de service

1. ajouter un roleBinding associant

- le role global **view** sur le cluster => ClusterRole
- et le service de compte **www-sa** du namespace **test**

```
k create rolebinding www-sa-readonly \
  --clusterrole=view \
  --serviceaccount=test:www-sa \
  --namespace=test
```

- REM: la désignation **test:www-sa**
    - permet de différencier un nom de "sa" dans un ns particulier dans le cas où l'on travaille avec des **ClusterRoles** et / ou **ClusterRoleBindings**
- REM: le **--namespace=test** est lié à l'objet RoleBinding
    - le roleBinding va brider le role global view dans un ns

2. retester la commande curl dans le ctn nginx: pour la réponse

PersistentVolume et PersistentVolumeclaim

- reprendre l'exemple du déploiement piloté par kustomization
    - ajouter une ressource PersistentVolume (PV) **pv-nfs.yml** => décrivant le montage nfs
    - ajouter une resource PersistentVolumeClaim (PVC) **pvc-nfs** => formalisant une requête de stockage
        - avec des crtières

```
accessModes:
    - ReadWriteMany
  volumeMode: Filesystem
  storageClassName: nfs
  resources:
    requests:
      storage: 10Mi
```

- Si un volume persistant validant ces critères disponible, sera utilisé
- sinon le pod ne peut pas être ordonnancé

Bonus: PVC, storageClass et provionner

- création dynamique de PV via un Provisioner (nfs ici)
    - en fonction des requêtes de stockage
    - elles même liées à une classe de stockage
- cas d'utilisation das le dossier **k8s-nfs-storage**

1. créer le ns k8s-nfs-storage
2. considérer le manifeste **deployment**

- dont l'image peut manipuler le montage du nfs et monter des PVS à la volée
- config du nfs: PROVISIONER_NAME, NFS_SERVER, NFS_PATH

2. considérer le manifeste **rbac.yml** composé d'

- un ServiceAccount : permettant au déploiement précédent
    - de manipuler les PVS, PVCS, nodes, StorageClasses pour
    - les monter et les détacher sur les noeuds
    - les monter et les détacher sur les pods
- un rôle composite pour réaliser les actions précédentes
- un roleboinding entre le SA et le Role

3. considérer le manifeste **class.yml**, utilsant le provionner
4. appliquer les manifestes précédents au moyen du manifeste **kustomization.yml**
5. considérer les manifestes **test-claim.yml** et **test-pod.yml**

- pour le test

6. enchaînement

- POD test-pod => PVC test-claim
- PVC test-claim => StorageClass nfs-client
- StorageClass nfs-client => provisionner k8s-sigs.io/v1
- provisionner k8s-sigs.io/v1 va satifaire la demande du PVC test-claim du POD test-pod
- en ajoutant un PV

cf le slide p 45