AKS : Azure Kubernetes Services

Orchestration de conteneurs dans le Cloud public

OBJECTIFS DU COURS

- Rappels sur Docker : Concepts clés
- Présentation des concepts de base de Kubernetes
- Déployer un cluster AKS avec Terraform
- Déployer des applications et gérer leur cycle de vie
- Exposer en réseau
- Assurer la persistance des données
- Intégrer Prometheus et Grafana
- Déployer en continu sur AKS

Module 1

Introduction et mise en place

Rappels sur Docker : Concepts clés

- **Nombre réduit et fini des types de ressources, qui sont :**
  - Image
  - Conteneur
  - Volume
  - Réseau
- **Périmètre d'utilisation délibérément restreint**
- **Les versions modernes de Docker reposent sur containerd**

Concepts de base de Kubernetes (alias K8s)

Kubernetes est un orchestrateur, un programme se proposant d'orchestrer des conteneurs répartis sur des ordinateurs appelés nœuds, dont le regroupement constitue un cluster, et d'en assurer tout le cycle de vie (création, fonctionnement et élimination).

Il permet d'apporter une solution pratique et efficace à deux types de problèmes que les gestionnaires de conteneurs (Docker, Podman ou autre) ne veulent pas prendre à leur charge :

- **la gestion de la « scalabilité »**
- **la haute disponibilité (HA)**

kubernetes

Origine, statut et modèle de Kubernetes

Kubernetes est né dans les laboratoires de Google sous le nom Borg pour répondre à des besoins internes. Plus tard Google a offert son produit en le confiant à la CNCF. À cette occasion, Borg a pris le nom de Kubernetes et est devenu un produit Open Source que les principales sociétés informatiques de par le monde contribuent à faire évoluer.

Des alternatives existent, peu nombreuses, Docker Swarm, Apache Mesos, Nomad, qui ne constituent qu'une concurrence marginale tant la position de Kubernetes est hégémonique dans le monde des orchestrateurs.

Origine, statut et modèle de Kubernetes (suite)

Kubernetes n'est aucunement lié à Docker.

Kubernetes sait à peine ce qu'est un conteneur et ne travaille pas au niveau du conteneur (voir diapo suivante) et sait encore moins ce qu'est une image.

Pour assurer son travail, Kubernetes se « contente » de piloter une solution de conteneurisation (choisie parmi plusieurs, cri-o, rkt, containerd, podman, par l'installateur du cluster lors de la définition de ce dernier) déployée à l'identique sur chacun des nœuds du cluster.
Pour qu'une solution de conteneurisation puisse être pilotée par Kubernetes, il faut que celle-ci se conforme à la CRI (Container Runtime Interface)*.

*** Docker n'a jamais fait en sorte d'être compatible CRI.**

Nodes, pods, services, deployments, …

Pour arriver à ses fins, Kubernetes s'appuie sur une très grande variété de types de ressources que l'on sera amené, en fonction des besoins, à compléter en en ajoutant de nouveaux.
Une installation basique offre facilement plus d'une cinquantaine de types (là où Docker n'en propose que quatre, sans possibilité d'extension).

Exécuter la commande `kubectl api-resources` **permet de se faire une bonne idée de la diversité des resources disponibles dans son cluster.**

Remarque :

Contrairement à un orchestrateur comme Docker Swarm qui ne nécessite que peu d'administration (et celle-ci se révèle en général très simple), Kubernetes peut impliquer un important travail d'administration et par conséquent une équipe de spécialistes dédiée à cette tâche.
On comprendra d'autant mieux que l'on y distingue l'activité d'administration de celle d'exploitation d'un cluster.
Il sera tout aussi facile de comprendre pourquoi de grandes sociétés (qui disposent pourtant de départements informatiques de pointe) préfèrent déléguer l'infogérance de leur(s) cluster(s) de production pour ne se consacrer qu'à leur exploitation.
Ces « hébergeurs », spécialisés en administration Kubernetes sont, pour les plus connus, Amazon avec EKS, Google avec GKE, Microsoft avec AKS, etc.

Déploiement d'un cluster AKS avec Terraform

- **Prérequis**
  - Vous devez avoir un compte Azure avec un abonnement actif.
    Si vous n'en avez pas un, *créez un compte gratuitement*.
  - Az-cli installé (*https://learn.microsoft.com/fr-fr/cli/azure/install-azure-cli*)
  - Connecté à votre abonnement Azure (`az login`)
  - Terraform installé (*https://developer.hashicorp.com/terraform/downloads*)
  - Kubectl installé (*https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/**)

ATTENTION : les explications en français peuvent être obsolètes !

Déploiement d'un cluster AKS avec Terraform

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
    }
  }
}
provider "azurerm" {
  subscription_id = "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"   # colonne "SubscriptionId" avec "az account list -o table"
}
```

```
variable "resource_group_location" {
  default     = "francecentral"   # az account list-locations | jq '[ .[] | select(.name | startswith("france")).name ] | sort'
  description = "Localisation géographique du groupe de ressources"
}
variable "node_count" {
  default     = 3   # compter environ 10 minutes pour créer le cluster
  description = "Quantité initiale de nœuds pour la réserve (pool)"
}
```

```
output "kube_config" {
  value     = azurerm_kubernetes_cluster.k8s.kube_config_raw
  sensitive = true   # obligatoire sans quoi terraform affiche une erreur
}
```

```hcl
import {                                        # https://developer.hashicorp.com/terraform/language/import
  to = azurerm_resource_group.rg_a_moi
  id = "/subscriptions/aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee/resourceGroups/rg-ycadin_cours-azure-kubernetes"
}
resource "azurerm_resource_group" "rg_a_moi" {   # terraform import azurerm_resource_group.rg_a_moi \
  location = var.resource_group_location         #    $(az group show -g rg-ycadin_cours-azure-kubernetes --query id -otsv)
  name     = "rg-ycadin_cours-azure-kubernetes"
  tags = {
    user = "ycadin"
  }
}
resource "azurerm_kubernetes_cluster" "k8s" {    # détails : az aks show -n cluster-yc -g rg-ycadin_cours-azure-kubernetes
  location            = azurerm_resource_group.rg_a_moi.location
  resource_group_name = azurerm_resource_group.rg_a_moi.name
  name                = "cluster-yc"
  dns_prefix          = "k8s-yc-noeud"   # préfixe des noms des nœuds créés
  identity {
    type = "SystemAssigned"              # ou "UserAssigned"
  }
  default_node_pool {
    name       = "agentpool"
    vm_size    = "Standard_B2S"          # Standard_A2_v2 : CPUs 4 GiO (ancienne désignation : Standard_D2_v2 : 2 CPUs 7 GiO)
    node_count = var.node_count          # facultatif (3 par défaut)
  }
  network_profile {
    network_plugin = "kubenet"           # valeurs possibles : azure, kubenet ou none
    network_policy = "calico"            # none par défaut ; requis pour que les NetworkPolicies de Kubernetes aient un effet
  }
  tags = {
    user = "ycadin"
  }
}
```

Prise en main d'AKS

- **Configuration minimale**
  - Utilisation des outputs Terraform pour mettre en place la communication entre kubectl (programme client) et le « cerveau » du cluster (l'api-server) :

```
echo "$(terraform output kube_config)" > azurek8s
export KUBECONFIG="$PWD/azurek8s"

kubectl version
```

Pour les autres modes d'accès (option --kubeconfig, variable KUBECONFIG ou fichier ~/.kube/config), consulter :
https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/#merging-kubeconfig-files

Utilisation de kubectl pour interagir avec le cluster

- **Un « petit tour du propriétaire » :**
  - Liste des nœuds qui constituent le cluster et détail des moindres aspects du cluster :

```
kubectl get nodes
kubectl get nodes --output wide   # en abrégé : kubectl get no -o wide

kubectl cluster-info
kubectl cluster-info dump   # ATTENTION : produit BEAUCOUP de lignes !
```

- **Se simplifier l'utilisation de kubectl :**

```
cat >> ~/.bashrc <<FIN_AJOUT
source <(kubectl completion bash)
alias k=kubectl
source <(k completion bash | sed s/kubectl/k/g)
FIN_AJOUT
. ~/.bashrc   # source et . sont une seule et même commande
k ver<TAB>
```

Namespaces : cloisonnement logique au sein du cluster

- **Création**

```
kubectl get namespaces
kubectl create namespace preprod
kubectl get ns
```

- **Utilisation**

```
kubectl run un-pod --image nginx:alpine --namespace preprod
k create -n preprod un-deploiement --image=https:alpine --replicas 3
k get all -n preprod
k get all --all-namespaces   # ou -A pour sa version courte
```

- **Suppression**

```
k delete ns preprod   # ATTENTION : PAS de demande de confirmation !
```

Module

Déploiement et gestion des applications

Manifestes (Deployment, Service, ConfigMap, Secret, …)

Travailler en mode interactif avec kubectl correspond à un fonctionnement impératif.
C'est pratique pour découvrir Kubernetes et expérimenter mais, dans le cadre d'une utilisation professionnelle, ce mode présente deux défauts majeurs :

1) il n'est que difficilement possible d'accéder aux innombrables détails des différentes ressources que l'on est amené à manipuler.
2) Surtout il est compliqué de conserver une trace des différentes opérations effectuées. Il est risqué de se baser sur l'historique de l'interpréteur de commandes employé (cela pose de nombreux problèmes).

Pour ces raisons et pour d'autres encore, Kubernetes recommande fortement d'opter pour un fonctionnement déclaratif.
Le principe consiste à décrire les ressources que l'on veut mettre en œuvre dans des fichiers au format YAML (privilégié par Kubernetes) que l'on appelle des manifestes et d'utiliser les actions `create` / `apply` / `delete` de kubectl en leur communiquant en paramètre ces fichiers (ou des dossiers entiers).

Exemples de manifestes :

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: vitrine
 namespace: preprod
 labels:
  app: vitrine
spec:
 replicas: 3
 selector:
  matchLabels:
   app: vitrine
 template:
  metadata:
   labels:
    app: vitrine
  spec:
   containers:
   - name: web
    image: nginx:1.26
```

```
apiVersion: v1
kind: Service
metadata:
 name: vitrine
 namespace: preprod
 labels:
  app: vitrine
spec:
 selector:
  app: vitrine
 ports:
 - port: 80
  protocol: TCP
  targetPort: 80
 type: LoadBalancer
```

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: nginx-discret
data:
 discret.conf: |
  server_tokens on;
```

```
apiVersion: v1
kind: Secret
metadata:
 name: mongodb-confidentiel
type: Opaque
data:
 mongo-root-username: dXNlcm5hbWU=
 mongo-root-password: cGFzc3dvcmQ=
```

Utilisation des manifestes :

- **Création**

```
kubectl create -f un_manifeste.yaml
k create --recursive -f dossier   # lit les fichiers .yaml, .yml et .json
```

- **Mise à jour (ou création)**

```
kubectl apply -f un_manifeste.yaml
```

- **Suppression**

```
kubectl delete -f un_manifeste.yaml
```

- **Édition (à la volée, au format YAML)**

```
kubectl edit pod/un-pod
```

- **Documentation (« explication ») / génération de modèles**

```
kubectl explain [ --recursive ] pod ; kubectl explain pod.metadata
k create deploy/modele --image nginx -oyaml --dry-run=client > mod_deploy.yml
k get -n kube-system ds/cloud-node-manager -o yaml > exemple_daemonset.yaml
```

Les labels pour catégoriser les ressources

Les labels représentent un type de donnée essentiel pour le fonctionnement de Kubernetes.
Ils servent tour à tour pour discriminer les résultats de l'action get, pour l'établissement de relations entre ressources (un replicaset et les pods qui lui sont associés par exemple ou entre un service et la ressource qu'il permet d'exposer).
Enfin ils sont un des principaux moyens utilisés pour déterminer le placement des pods sur les nœuds du cluster.

- **Visualisation / exploitation**

```
kubectl get all --show-labels
kubectl get all --selector app==vitrine   # --selector et -l sont équivalentes
```

- **Création**

```
kubectl label namespace/preprod responsable=Thomas
```

- **Modification**

```
kubectl label namespace preprod responsable=Alice --overwrite
```

- **Suppression**

```
kubectl label namespace preprod responsable-
```

Annotations

Les labels présentent plusieurs limitations, quant aux noms qu'ils peuvent avoir mais aussi à la taille et au jeu de caractères qu'ils autorisent dans leur valeur.
Il peut, parfois, s'avérer utile, d'enregistrer dans les ressources des éléments d'informations textuels dont la taille et/ou le contenu outrepassent les possibilités autorisées par les labels.
Pour répondre à ce besoin, il existe les annotations, leur mise en œuvre est similaire à celle des labels.
Contrairement aux labels, il ne faut pas utiliser les annotations pour répondre à des exigences techniques (certains contrôleurs, et non des moindres, ne respectent pas cette préconisation).

- **Visualisation / exploitation**

```
kubectl get ns preprod -o jsonpath='{.metadata.annotations}{"\n"}'
```

- **Création**

```
kubectl annotate namespace/preprod statut="Dernière révision le 2 août"
```

- **Modification**

```
kubectl annotate namespace preprod statut="Last release 08/02" --overwrite
```

- **Suppression**

```
kubectl annotate namespace preprod statut-
```

Exposition (réseau) des applications

L'impressionnante diversité de types de ressources offerte par Kubernetes (plusieurs dizaines contre quatre pour Docker et moins d'une dizaine pour Docker Swarm) s'explique, entre autres, par un souci de forte spécialisation de chaque type de ressource. De fait, ni les pods, ni les déploiements, pas plus que les daemonsets ou les statefulsets ne sont, de base, accessibles en réseau. Pour leur fournir de la « connectivité » il convient de les associer à une ressource complémentaire, un service.
Les services se déclinent en quatre types : ClusterIP (par défaut), NodePort, LoadBalancer et ExternalName.
ClusterIP est limité à un usage interne au cluster, pour la communication entre pods.
NodePort permet d'associer un port de communication (choisi par l'api-server, par défaut dans l'intervalle 30000-32767) permettant l'accès au pods (qualifiés de « endpoints ») associés à ce service depuis n'importe quel nœud du cluster sur le port en question. Les paquets sont aiguillés alternativement (principe de répartition de charge) vers les différents endpoints du service.
ExternalName n'est qu'un mécanisme d'alias (comme l'enregistrement CNAME d'un service DNS).
LoadBalancer est, contrairement à ce que peut laisser supposer sa désignation, un mécanisme qui ignore le répartiteur de charge présent dans Kubernetes au profit d'un LoadBalancer externe. (Solution en général proposée chez les « gros » hébergeurs de clusters Kubernetes infogérés, tels que Amazon, Google ou Microsoft.)

Exposition réseau, exemples :

- **NodePort**

```
kubectl create deploy boutique --image nginx --replicas 3
kubectl expose deploy boutique --type NodePort --port 80
* K8s ne peut deviner sur quels ports « écoutent » les conteneurs d'un pod
kubectl get svc/boutique -o wide
```

- **LoadBalancer**

```
kubectl create deploy compta --image nginx --replicas 3 --port 80
k expose deploy/compta --type=LoadBalancer   # port renseigné ci-dessus
k get svc compta --watch
* l'obtention d'une « EXTERNAL-IP » peut prendre plusieurs secondes
curl $(k get svc compta -o jsonpath='{.status.loadBalancer.ingress[].ip}')
```

- **Suppression**

```
k delete svc boutique     # seuls les services sont supprimés, les
k delete service/compta   # déploiements correspondants ne sont pas affectés
```

Stockage persistant

Les pods (et leurs conteneurs) n'étant pas voués à exister éternellement, il peut être nécessaire de déporter à l'extérieur de ces pods les données traitées par les programmes qui s'exécutent au sein des conteneurs afin dans assurer la conservation durable, indépendamment du cycle de vie des pods.
À cette fin Kubernetes fournit un type de volume appelé Persistent Volume, comparable aux volumes de Docker.
Les PV (Persistent Volume) de Kubernetes se distinguent de leurs homologues Docker sur au moins trois aspects :

- un découplage entre l'offre de stockage à proprement parler (ressources de type PersistentVolume) et les demandes de stockage (ressources de type PersistentVolumeClaim).
  L'idée étant que l'offre de stockage est définie par les administrateurs du stockage de l'infrastructure qui accueille le cluster Kubernetes (car eux seuls connaissent les capacités disponibles et surtout les protocoles de partage de fichiers employés en interne).
  Les demandes de stockage (PVC) sont, elles, définies par les exploitants (utilisateurs) du cluster.
  Kubernetes se chargeant de mettre en correspondance, de façon optimale, les premiers avec les secondes.
  Cette décorrélation facilite le portage d'applications entre clusters (développement -> production, par exemple) situés dans des infrastructures ne disposant pas des mêmes solutions de stockage (elles sont en général « propriétaires » dans les clouds Amazon, Google ou Microsoft).
- Tandis que les volumes Docker se déclinent en seulement deux variantes, nommé et « bind », les PV se déclinent quant à eux en un grand nombre de variantes : hostPath, emptyDir, local, NFS, CephFS, iSCSI, etc.
- Des classes de stockage (StorageClass) permettent de gérer dynamiquement le provisionnement des PV.

Mise en œuvre, manifestes :

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: entrepot-azurefile-csi
provisioner: file.csi.azure.com
parameters:
  skuName: Standard_LRS  # par défaut
  storageAccount: entrepot4aks
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=0
  - gid=0
  - mfsymlinks
  - cache=strict
  - nosharesock
  - actimeo=30
  - nobrl
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: demande-pv-boutique
spec:
  storageClassName: entrepot-azurefile-csi
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Mi
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: boutique
  labels:
    app: boutique
spec:
  replicas: 2
  selector:
    matchLabels:
      app: boutique
  template:
    metadata:
      labels:
        app: boutique
    spec:
      containers:
        - name: serveurweb
          image: nginx:alpine
          volumeMounts:
          - name: fichierswebstatiques
            mountPath: /usr/share/nginx/html
      volumes:
        - name: fichierswebstatiques
          persistentVolumeClaim:
            claimName: demande-pv-boutique
```

Stockage persistant, mise en œuvre

- **Création d'un compte de stockage (spécifique Azure / AKS)**

```
GROUPE_AZURE_AKS=rg-ycadin_cours-azure-kubernetes
NRG=$(az aks show -g $GROUPE_AZURE_AKS -n cluster-yc --query nodeResourceGroup -otsv)
az storage account create -g $NRG -n entrepot4aks -l francecentral --sku Standard_LRS
```

- **Création de la classe de stockage (Kubernetes) personnalisée (premier manifeste, à gauche)**

```
kubectl k create -f classe-de-stockage-entrepot.yaml
kubectl get sc entrepot-azurefile-csi -o wide   # juste pour contrôler
```

- **Création de la demande de volume propre à l'application à déployer (deuxième manifeste, au centre)**

```
kubectl create -f demande-pv-boutique.yaml
k get pvc,pv -o wide | grep demande-pv-boutique   # pas de "propagation" de label
```

- **Création du déploiement, des ressources associées (dernier manifeste, à droite) et vérifications**

```
k create -f deploiement-boutique.yaml
k get po -l app==boutique -o wide   # pour avoir les noms des pods (utilisés ci-après)
k exec boutique-xxxxxxxxx-prems -- curl -s localhost
k exec boutique-xxxxxxxxx-prems -- sh -c "echo COOL > /usr/share/nginx/html/index.html"
k exec boutique-xxxxxxxxx-prems -- curl -s localhost   # doit afficher « COOL »
k exec boutique-xxxxxxxxx-deuze -- curl -s localhost   # doit afficher « COOL »
```

Module

Scalabilité et mise à jour

« Scalabilité » : gestion de la mise à l'échelle

La gestion de la mise à l'échelle est l'un des enjeux majeurs que se proposent de résoudre les orchestrateurs de conteneurs. Mais que recouvre ce terme ?
Il existe deux principaux types de mécanismes de mise à l'échelle, le vertical* et l'horizontal. Ce dernier est fourni en standard dans Kubernetes et consiste à augmenter ou réduire de le nombre de pods (nommés replicas) d'un déploiement ou d'un statefulset.
Ce principe se décline en deux variantes, manuelle et automatique appelée Horizontal Pod Autoscaler (HPA).

* La mise à l'échelle verticale consiste à fournir plus de moins de ressources CPU et RAM aux pods.
** Il existe également un concept de mise à l'échelle au niveau du cluster qui consiste à ajouter automatiquement des nœuds quand cela s'avère nécessaire. Ce mécanisme n'est pas disponible en standard dans Kubernetes.

- **Exemples de changements d'échelle en mode manuel**

```
kubectl scale deploy boutique --replicas 5
k apply -f deploiement-boutique.yaml  # après édition pour changer replicas
kubectl edit deploy boutique       # revient à éditer un manifeste éphémère
```

Horizontal Pod Autoscaler (HPA)

Déterminer automatiquement à quel moment il est nécessaire d'augmenter le nombre de pods d'un déploiement et à quel moment il est possible de le diminuer impose de pouvoir mesurer la consommation CPU des pods. Pour réaliser cela, il convient d'installer préalablement à toute mise en œuvre de HPA un serveur de métriques (avec son API). Un tel service est de base présent dans les clusters AKS. Pour le vérifier il suffit d'exécuter l'instruction suivante :

```
k top no  # des statistiques sont affichées, prouvant la présence du service
```

Il convient également de définir les prérequis en terme de consommation CPU pour les différents conteneurs qui constituent les pods dont il faudra assurer la mise à l'échelle automatique afin que le contrôleur HPA sache quels sont les seuils à partir desquels déclencher une augmentation ou une réduction.

Remarque : Kubernetes, et cela peut se comprendre, est beaucoup plus prompt à augmenter le nombre de replicas qu'à le diminuer.

Horizontal Pod Autoscaler, exemple :

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: boutique
spec:
 selector:
  matchLabels:
   app: boutique
 template:
  metadata:
   labels:
    app: boutique
  spec:
   containers:
    - name: serveur
      image: nginx:alpine
      ports:
       - containerPort: 80
      resources:
       requests:
        cpu: 10m   # 10 milli-cpu => 0.01 CPU requis
```

- **Mise en œuvre d'une politique de changement d'échelle automatique**

```
kubectl create -f deploiement-boutique.yaml   # 1 seul replica (par défaut)
kubectl expose deploy boutique
kubectl create -f hpa-boutique.yaml
```

- **Dans un premier terminal :**

```
watch "kubectl get hpa hpa-boutique; echo; kubectl get po -l app==boutique"
```

- **Dans un second terminal :**

```
IP=$(k get svc boutique -ojsonpath={.spec.clusterIP})
for N in {1..2500}; do k exec POD_BOUTIQUE -- curl -s $IP > /dev/null; done
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: hpa-boutique
spec:
 maxReplicas: 10  # minReplicas = 1 par défaut
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: boutique
 targetCPUUtilizationPercentage: 50
```

*Formule pour le nombre de replicas : desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]*

Définition de ressources : requests/limits CPU/mémoire

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: frontend:1
    resources:
      requests:
        memory: "64Mi"   # Mebioctets
        cpu: "1"   # 1 cœur de CPU
      limits:
        memory: "96Mi"   # 2^20 octets
        cpu: "3"   # 3 cœurs de CPU
  - name: log-aggregator
    image: logaggre:2
    resources:
      requests:
        memory: "30M"   # Megaoctets
        cpu: "150m"   # 15% de CPU
      limits:
        memory: "50M"   # 10^6 octets
        cpu: "250m"   # 25% de CPU
```

La section resources que l'on peut trouver dans les spécifications de conteneurs ne sert pas seulement pour le fonctionnement de l'Horizontal Pod Autoscaler. Elle permet avant tout d'informer Kubernetes quant aux quantité minimale de mémoire et/ou de puissance minimale de processeur requises pour garantir le bon fonctionnement des processus qu'accueilleront les pods et également les limites (garde-fous) en terme de maxima autorisés quant à ces mêmes ressources.

Pour les premières (prérequis), cela peut influencer le scheduler dans son choix des nœuds sur lesquels placer les nouveaux pods. En effet, si au moment du choix du placement certain nœuds ne disposent pas ou plus de suffisamment de ressources pour satisfaire les « exigences » (requests) définies alors les nœuds en question seront écartés de la liste des destinataires potentiels.

En ce qui concerne les limites (garde-fous comme j'aime à les appeler), il n'y en a pas par défaut.

Si l'on en définit alors cela préviendra le risque d'accaparement des ressources consommées par un conteneur au détriment d'autres conteneurs ou, le cas échéant, des programmes qui tournent nativement sur le nœud.

Mises à jour : rolling updates et rollbacks

Tôt ou tard, tout programme, tout au moins la plupart, connaît des modifications, améliorations, corrections.
Il en va de même pour ceux qui s'exécutent dans des conteneurs.
Dans ce cas, c'est une image plus moderne, incluant le programme, qui est générée pour succéder à une image de version plus ancienne, une image étant immuable par nature.
Un orchestrateur ayant pour vocation de garantir un taux maximal de disponibilité, le basculement d'une ancienne version à une nouvelle ne doit pas entraîner de discontinuité dans le service rendu (par les déploiements ou les statefulsets notamment).
Pour éviter tout risque d'interruption de service, différentes techniques de mises à jour ont été imaginées.
Kubernetes adopte, par défaut, la mise à jour par roulement.
Le principe est de remplacer une portion des replicas d'un déploiement (ou d'un statefulset), 25% par défaut, par des pods basés sur la nouvelle image, puis, quand ces derniers sont à l'état « Running », de passer à la portion suivante et ainsi de suite jusqu'à remplacement complet de l'ensemble des pods.
Kubernetes accorde un « droit à l'erreur ». Si une mise à jour ne donne pas le résultat escompté, il est facile de l'annuler pour revenir à l'état précédent.

Rolling updates et rollbacks, exemples

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: boutique
 labels:
  app: boutique
spec:
 replicas: 12
 selector:
  matchLabels:
   app: boutique
 template:
  metadata:
   labels:
    app: boutique
  spec:
   containers:
    - name: serveurweb
      image: nginx:1.18
```

- **Mise en place d'une version initiale du déploiement :**
  `kubectl create -f deploiement-boutique.yaml`
- **Deux façons de déclencher la mise à jour par roulement :**
  - `kubectl set image deploy boutique serveurweb=nginx:1.27`
  - Éditer le manifeste pour y modifier la valeur de image puis exécuter `kubectl apply -f deploiement-boutique.yaml`
- **Pour suivre la progression de la mise à jour :**
  `kubectl rollout status deploy boutique`
- **Pour annuler une mise à jour et revenir à l'état précédent :**
  `kubectl rollout undo deploy boutique`
- **Et pour avoir l'historique des mises à jour :**
  `kubectl rollout history deploy boutique`

Ingress, contrôleur Ingress Nginx

Les types NodePort et LoadBalancer des services n'offrent que des possibilités limitées. L'intervalle de ports utilisable pour les NodePort est restreint. Chaque LoadBalancer « consomme » une adresse IP publique (pas de « mutualisation »). Enfin, aucun de ces types ne permet de définir du « routage » pour aiguiller des requêtes HTTP.
La nécessité d'exposer un grand nombre de services peut aboutir à une pénurie des adresses IP disponibles.
Pour répondre à ces problématiques, la solution la plus évidente consiste alors à employer un reverse-proxy.
Il en existe dont le fonctionnement a été adapté pour s'intégrer à Kubernetes.
Mais en général on préférera se tourner vers un contrôleur Ingress. Contrôleur Ingress et reverse-proxy sont très similaires sur la forme. Dans les grandes lignes, ils offrent les mêmes possibilités.
Parmi les caractéristiques générales on notera l'écoute par défaut en HTTP (port 80) et en HTTPS (port 443 avec un certificat SSL auto-signé). La possibilité de définir du routage sous la forme de règles qui peuvent être simples ou complexes dans le but d'aiguiller chaque requête reçue vers le pod approprié.

Remarque : Kubernetes intègre de base les ressources Ingress (qui permettent la définition des règles de routage) mais n'est pas capable de les mettre en œuvre tant qu'un contrôleur Ingress (optionnel) n'a pas été installé.
En l'absence de ce dernier, toute ressource Ingress qui aura été créée restera sans effet.

Ingress, mise en œuvre et exemple :

- **Préalable, installation d'un contrôleur Ingress :**

```
  k apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-
v1.12.3/deploy/static/provider/cloud/deploy.yaml
  k get svc -n ingress-nginx  # ingress-nginx-controller doit avoir une EXTERNAL-IP
  IP_PUBLIQUE=$(k get -n ingress-nginx svc ingress-nginx-controller \
                    -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

- **Création et exposition des applications :**

```
  k create deploy zorglub --image nginx:alpine --replicas 2 --port 80
  k create deploy trucmuche --image nginx:alpine --replicas 2 --port 80
  k expose deploy zorglub
  k expose deploy trucmuche
```

- **Personnalisation de la page d'accueil des pods :**

```
  for POD in $(k get po -l "app in (trucmuche, zorglub)" --no-headers \
                       -o custom-columns=:.metadata.name)
  do
    k exec $POD -- sh -c "echo je suis le pod $POD > /usr/share/nginx/html/index.html"
  done
```

Suite et fin à la diapo suivante…

Ingress, mise en œuvre (suite et fin) :

- **Premières règles et vérification :**

```
k create ing sites-web --class=nginx --rule "*.zorglub.aks/*=zorglub:80" \
                                     --rule "*.trucmuche.aks/*=trucmuche:80"
k get ing -w   # attendre l'obtention d'une IP (dans la colonne ADDRESS)
curl --header "Host: web.trucmuche.aks" $IP_PUBLIQUE
```

- **Définir la classe Ingress par défaut :**

```
k annotate ingressclass nginx ingressclass.kubernetes.io/is-default-class=true
```

- **Second jeu de règles et vérification :**

```
k create ingress domaines --rule "zorglub.aks/*=zorglub:80" \
                          --rule "trucmuche.aks/*=trucmuche:80"
k get ingresses -w   # attendre l'obtention d'une IP (dans la colonne ADDRESS)
curl -H "Host: trucmuche.aks" -k https://$IP_PUBLIQUE
```

Certificats SSL/TLS avec Let's Encrypt

Ajouter un contrôleur Ingress au cluster représente une étape importante, mais elle n'est qu'un jalon vers une solution complète.
En effet, le contrôleur Ingress a beau écouter en standard sur les ports 80 (HTTP) et 443 (HTTPS), c'est un certificat SSL/TLS « factice » (auto-signé) qui est présenté par défaut au client (typiquement un navigateur web).
Il convient donc, avant de publier un site pour de la production, de se doter d'un certificat en bonne et due forme.
Le contrôleur Ingress a ceci de très intéressant qu'il va permettre de gérer les différents certificats acquis par le biais de ressources Kubernetes et ce de manière automatique et uniforme (nous épargnant ainsi des configurations, potentiellement disparates, des différents serveurs Web déployés dans les conteneurs).
Il existe nombre d'organisations, appelées autorités de certification (CA), habilitées à émettre des certificats.
Ces autorités étaient toutes payantes dans le passé mais, dans le but de favoriser l'adoption des mécanismes de chiffrement, d'identification et de sécurisation d'une façon générale des échanges réseau (à commencer par Internet), de nouvelles ont vu le jour, à but non lucratif. L'une des plus connues et des plus plébiscitées se nomme Let's Encrypt.

Certificat Let's Encrypt, mise en œuvre :

- **Configuration d'un nom de machine (enregistrement DNS) :**

```
L'obtention d'un certificat requiert un nom de machine. Ce dernier devra
être associé à l'adresse stockée dans $IP_PUBLIQUE récupérée préalablement.
Cette opération peut être automatisée avec Terraform, DNSControl ou octoDNS.
FQDN=ing.formateur-devops.ovh   # dans le cas présent (ing pour « ingress »).
```

- **Installation optionnelle (sur le poste courant) de l'utilitaire associé au gestionnaire ci-après :**

```
  VERSION=$(lastversion cmctl)   # retourne 2.2.0
  curl -L https://github.com/cert-manager/cmctl/releases/download/v$VERSION/
cmctl_linux_amd64.tar.gz | tar xz -C /usr/local/sbin/
```

- **Installation du gestionnaire de certificats :**

```
  VERSION=$(lastversion cert-manager)   # retourne 1.18.1
  k apply -f https://github.com/cert-manager/cert-manager/releases/download/
v$VERSION/cert-manager.yaml
  cmctl check api --wait 2m  # moins de 10s pour "The cert-manager API is ready"
  k get all -n cert-manager
```

Suite à la diapo suivante…

Certificat Let's Encrypt (suite) :

- **Création d'un émetteur ("issuer") de certificats :**

```
k create -f letsencrypt-prod-ciss.yaml   # le manifeste ci-dessous
k get ciss -o wide
k get secrets -n cert-manager clef-letsencrypt-prod -o yaml
```

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer            # ou "Issuer" si confiné dans un namespace
metadata:
 name: letsencrypt-production   # ou "letsencrypt-pour-jouer" (si l'on opte pour staging ci-après)
spec:
 acme:
  server: https://acme-v02.api.letsencrypt.org/directory   # ou https://acme-staging-v02.api.letsencrypt.org/directory
  email: yannick@diablotin.fr
#   profile: tlsserver    # "classic" par défaut (https://cert-manager.io/docs/configuration/acme/#acme-certificate-profiles)
  privateKeySecretRef:
   name: clef-letsencrypt-prod   # ou clef-letsencrypt-jeu (si staging ci-avant)
  solvers:
   - http01:
      ingress:
       ingressClassName: nginx   # anciennement "class: nginx" (https://cert-manager.io/docs/releases/upgrading/ingress-class-compatibility)
```

Suite et fin à la diapo suivante…

Certificat Let's Encrypt (suite et fin) :

- **Création d'un Ingress et vérification de l'obtention d'un certificat :**

```
k create ing boutique --rule="$FQDN/*=boutique:80,tls=clef-le-boutique" \
     --annotation cert-manager.io/cluster-issuer=letsencrypt-production
k get Ingress,CertificateRequest,Challenge,Secret,Certificate -o wide
curl https://$FQDN   # NOTER absence de l'option -k
```

- **En cas de problème (si cURL évoque un certificat auto-signé)…**

```
k describe challenge | tail -3
```

Error presenting challenge: admission webhook "validate.nginx.ingress.kubernetes.io" denied the request: ingress contains invalid paths: path /.well-known/acme-challenge/… cannot be used with pathType Exact

L'erreur est connue dans la version 1.18 de cert-manager.

- **Résolution (pis-aller) :**

```
k -n ingress-nginx patch cm ingress-nginx-controller \
  -p '{"data":{"strict-validate-path-type":"false"}}'  # où l'on découvre patch
```

***** Détruire puis créer à nouveau l'Ingress pour valider la correction ci-dessus.

```
k get ing,cert,cr,secret | egrep 'boutique|NAME|^$'  # Tout doit être à "True"
curl -Ivs https://$FQDN 2>&1 | grep ^*   # pour voir le détail du certificat
```

Module

Sécurité et monitoring

Role-Based Access Control (RBAC)

Même s'il est peu probable qu'une commande
kubectl delete all --all --all-namespaces
soit exécutée par inadvertance. Il est en revanche plus plausible d'écrire un jour
kubectl delete **n**o --all 😱
en voulant juste supprimer tous les **p**ods de l'espace de nommage courant. Et c'est le drame !

Une des forces, un des avantages de Kubernetes, est d'offrir un puissant mécanisme des gestions des identités et des droits afférents appelé RBAC (Role-Based Access Control).
Les actions dans un cluster Kubernetes ne se font jamais de façon anonyme. Un doute ?
kubectl auth whoami ; kubectl config view --minify -o jsonpath='{.contexts[].context.user}{"\n"}'

RBAC n'est pas obligatoirement activé. Les installations « modestes » s'en passent très bien. Aussi parce que ce composant est apparu tardivement, en 2017, dans Kubernetes.
À noter que RBAC ne se limite à gérer les droits des usagers mais aussi ceux des applications (appelés « comptes de service »).

RBAC, mise en œuvre :

Définition et configuration (dans le cluster actuel) d'un nouvel utilisateur : Ramirez

- **Récupération d'éléments contextuels :**

```
CONTEXT=$(k config current-context)
CLUSTER=$(k config view -o jsonpath="{.contexts[?(@.name == '$CONTEXT')].name}")
SERVEUR=$(k config view -o jsonpath="{.clusters[?(@.name == '$CLUSTER')].cluster.server}")
k config view --raw -o jsonpath="{.clusters[?(@.name=='$CLUSTER')].cluster.certificate-authority-data}" |
base64 -d > cluster.crt
```

- **Génération des documents validant l'identité du nouvel usager :**

```
openssl genrsa -out ramirez.key 4096  # 2048 bits par défaut
openssl req -new -key ramirez.key -out ramirez.csr -subj /CN=ramirez/O=immortels  # /O=groupe facultatif
cat > ramirez-csr.yaml << EOF
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: ramirez
spec:
  request: $(cat ramirez.csr | base64 | tr -d '\n')
  usages:
    - client auth
  signerName: kubernetes.io/kube-apiserver-client
EOF
k create -f ramirez-csr.yaml
k get csr ramirez
k certificate approve ramirez
k get csr ramirez    # utiliser sans tarder car disparaît au bout de 110 minutes environ (si "Approved")
```

Suite à la diapo suivante…

RBAC, mise en œuvre (suite) :

- **Intégration du nouveau venu au cluster :**

```
k get csr ramirez -o jsonpath='{.status.certificate}' | base64 -d > ramirez.crt

k --kubeconfig kubeconfig_ramirez config set-cluster $CLUSTER --server $SERVEUR \
    --embed-certs --certificate-authority cluster.crt
k --kubeconfig kubeconfig_ramirez config set-credentials ramirez \
    --embed-certs --client-certificate ramirez.crt --client-key ramirez.key
k --kubeconfig kubeconfig_ramirez config set-context rami@immortels.fr \
    --cluster $CLUSTER --user ramirez
k --kubeconfig kubeconfig_ramirez config use-context rami@immortels.fr
```

- **Contrôles et tentatives diverses :**

```
k --kubeconfig kubeconfig_ramirez auth whoami
k --kubeconfig kubeconfig_ramirez config view --minify

k --kubeconfig kubeconfig_ramirez get po   # Error from server (Forbidden): pods…
k --kubeconfig kubeconfig_ramirez auth can-i list po  # approche plus « délicate »
```


Suite et fin à la diapo suivante…

RBAC, mise en œuvre (suite et fin) :

- **Définition et affectation d'un rôle (aussi restrictif que possible) :**

```
k create role eternel --resource=pods,deployments,services \
                --verb=create,get,list,watch  # les astérisques sont autorisées
k create rolebinding eternel-rami --role=eternel --user=ramirez
```

- **Vérifications :**

```
k --kubeconfig kubeconfig_ramirez get po
k --kubeconfig kubeconfig_ramirez auth can-i list pods
KUBECONFIG=kubeconfig_ramirez k get all  # probablement partiellement en erreur
KUBECONFIG=kubeconfig_ramirez k run pod-de-rami --image nginx:alpine
KUBECONFIG=kubeconfig_ramirez k get po pod-de-rami
KUBECONFIG=kubeconfig_ramirez k describe po pod-de-rami
KUBECONFIG=kubeconfig_ramirez k delete po pod-de-rami -n toto
KUBECONFIG=kubeconfig_ramirez k create deploy app-de-rami --image nginx --replicas 2
KUBECONFIG=kubeconfig_ramirez k get deploy,pod,rs -l app==app-de-rami
```

- **À regarder si des erreurs imprévues apparaissent :**

```
 * contrôler en premier si l'espace de noms est le bon :
  k --kubeconfig kubeconfig_ramirez config view \
    --minify -o jsonpath='{..namespace}'   # « rien » équivaut à l'espace default
```


k api-resources --sort-by name -o wide | less # pour connaître les verbes disponibles par type de ressources
??? Parler des options --as / --as-group / --as-uid # pour réaliser une action ponctuellement avec une identité tierce (un peu comme sudo ou runas)

Network Policies

Dans un cluster Kubernetes, tous les pods sont connectés à un seul grand réseau. Et si les pods ne peuvent se « voir » (le ping n'aboutit pas), rien ne les empêche, en standard, d'échanger.
Il peut être souhaitable de mettre en place quelques restrictions afin de prévenir les communications indésirables.
Pour cela Kubernetes propose les politiques réseau par le biais d'une ressource nommée NetworkPolicy.
Mais… les règles définies avec les NetworkPolicies n'ont d'effet que si un « moteur de stratégie réseau » a été mis en place. Dans le cas d'AKS, cela ne peut se faire, officiellement, que lors de la création du cluster (avec l'une des deux valeurs possibles : azure ou calico, la valeur par défaut étant none).

Network Policies, mise en œuvre :

- **Mise en place du scénario :**

```
for NS in prod dev troie ; do k create ns $NS ; done

k -n prod create deploy serveur --image nginx --replicas 2 --port 80
k -n prod expose deploy serveur

for P in $(k -n prod get po -l app=serveur -o custom-columns=:.metadata.name --no-headers)
  do k -n prod exec $P -- sh -c "echo pod $POD > /usr/share/nginx/html/index.html" ; done

k -n dev run client --image nginx:alpine
k -n troie run mechant --image nginx:alpine
```

- **Les protagonistes entrent en scène :**

```
k -n dev exec client -- curl -s serveur.prod.svc.cluster.local  # PAS de restriction !

k -n troie exec mechant -- curl -s serveur.prod.svc.cluster.local  # PAS de blocage ! 🤨
```

Suite et fin à la diapo suivante…

Pas de « - » devant namespaceSelector, implique une combinaison avec podSelector
https://kubernetes.io/docs/concepts/services-networking/network-policies/

Network Policies, mise en œuvre (suite) :

- **Mise en place de politique(s) de restriction réseau et vérification :**

```
k create -f police_reseau.yaml

k -n troie exec mechant -- curl -s serveur.prod.svc.cluster.local
# plus de réponse (c'est ce que l'on veut)

k -n dev exec client -- curl -s serveur.prod.svc.cluster.local
# pas de réponse non plus => politique TROP coercitive
```

- **Ajustement pour être en conformité avec la politique puis vérifications :**

```
k get -n prod po serveur --show-labels  # ça, a priori, c'est bon
k get -n dev po client --show-labels  # ça, c'est bon également
k get ns dev --show-labels  # c'est là que ça coince !
k label ns dev espace=clients  # ajustement nécessaire
# pour correspondre à la définition de la règle de filtrage

k -n troie exec mechant -- curl -s serveur.prod.svc.cluster.local
# toujours pas de réponse, c'est « bloqué » (ce qui est le but)

k -n dev exec client -- curl -s serveur.prod.svc.cluster.local
#  SUCCÈS !
```

SI ÇA NE FONCTIONNE PAS ! (Pas de blocage d'aucune sorte) : az aks show -g rg-yc -n k8s-yc --query networkProfile | jq -r .networkPolicy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restriction-service
  namespace: prod
spec:
  podSelector:
    matchLabels:
      app: serveur
  policyTypes:
  - Ingress   # par défaut
  ingress:
  - from:
    - podSelector:
        matchLabels:
          run: client
      namespaceSelector:
        matchLabels:
          espace: clients
    ports:
    - protocol: TCP
      port: 80
```


Pas de « - » devant namespaceSelector, implique une combinaison avec podSelector
https://kubernetes.io/docs/concepts/services-networking/network-policies/
*** NE doit PAS être « none »**

Intégration de Prometheus et Grafana

Arrivera un jour où il faudra avoir une vision précise du fonctionnement des ressources.
Mettre en place un système de supervision et de surveillance du cluster et présenter les informations de façon aussi pratique et intelligible que possible se révélera, sinon nécessaire, du moins souhaitable.
Le couple Prometheus / Grafana se propose de faire cela.

La volonté de déployer Prometheus et Grafana constitute une bonne excuse pour aborder l'outil Helm.

Helm s'apparente à un gestionnaire de paquets (tels qu'on les connaît sur Linux, apt, dnf, etc), spécifique à Kubernetes. Les paquets y prennent le nom de "chart" et le déploiement d'un chart est appelé "release". Contrairement aux gestionnaires de paquets traditionnels, le concepteur d'un chart peut offrir un très haut degré de personnalisation du déploiement par le biais de données de configuration qu'on appelle "values".
À l'instar de n'importe quel gestionnaire de paquets, Helm facilite le déploiement d'application, en favorise l'automatisation et leur gestion tout au long de leur exploitation.

Prometheus et Grafana, installation et accès :

- **Déploiement :**

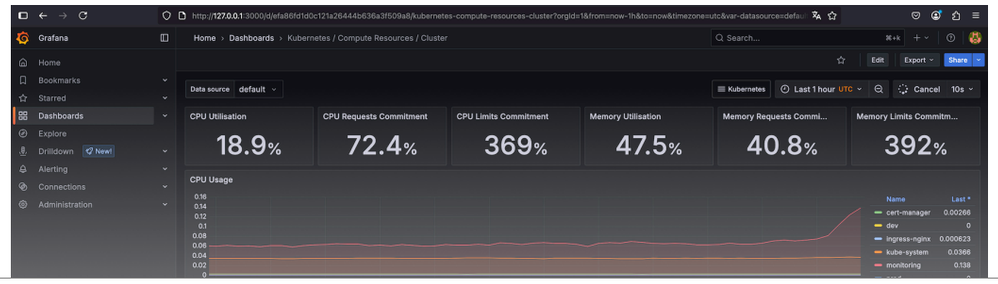
  Cela commence par l'installation de Helm : https://helm.sh/fr/docs/intro/install/

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm update   # utile seulement si le repo ci-dessus était déjà installé
helm install prometheus prometheus-community/kube-prometheus-stack \
  --create-namespace -n monitoring  # installé sinon dans l'espace courant par défaut
```

* Il faut attendre une douzaine de secondes avant d'obtenir un résultat.

- **Effectuer les opérations décrites dans le compte-rendu :**

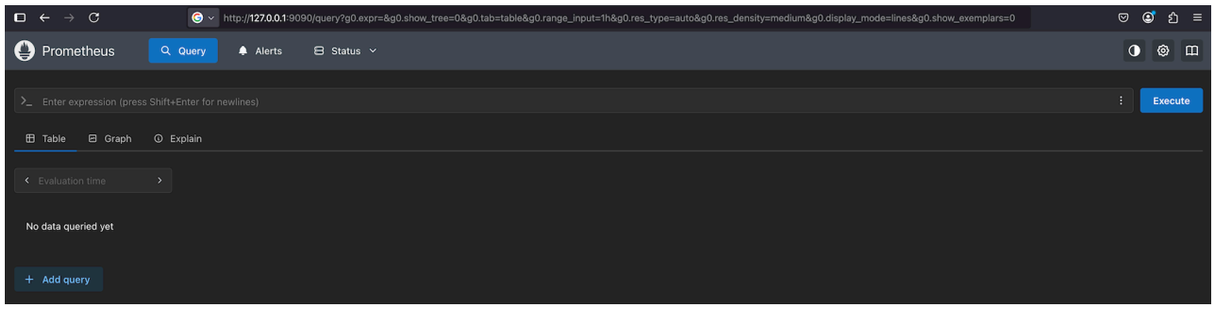
  Le mot de passe admin de Grafana est normalement « prom-operator » par défaut.



Sur Linux ou macOS il est suggéré d'enchaîner avec source <(helm completion bash)

- **Pour accéder à l'interface de Prometheus, compléter avec :**

```
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-prometheus 9090:9090
```



Module

Stratégies de placement des pods

Commander ou influencer le scheduler

```
apiVersion: v1
kind: Pod
metadata:
 name: pod-sur-noeud0
 labels:
  run: pod-sur-noeud0
spec:
 nodeName: aks-node-vm000
 containers:
  - name: web
    image: nginx:alpine
    ports:
     - name: web
       containerPort: 80
    volumeMounts:
     - name: fichiersweb
       mountPath: /usr/share/nginx/html
 volumes:
  - name: fichiersweb
    hostPath:
     path: /donnees-pour-pod
```

Il peut être nécessaire, dans certaines circonstances, de choisir le nœud sur lequel placer un pod, et parfois juste influencer le scheduler quant aux nœuds qui devraient recevoir de nouveaux pods.
Il existe au moins trois moyens pour arriver à réaliser cela.

- **Placement « simple » :**

```
k create -f placement-pod.yaml
k expose po pod-sur-noeud
k get svc pod-sur-noeud0 -o wide

k exec pod-sur-noeud0 -- sh -c "mount | grep nginx"
k exec pod-sur-noeud0 -- sh -c \
    "echo essai > /usr/share/nginx/html/index.html"
k exec pod-sur-noeud0 -- curl -s localhost

k delete -f placement-pod.yaml

k create -f placement-pod.yaml
k exec pod-sur-noeud0 -- curl -s localhost
```

Influencer le scheduler

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: minage
 labels:
  app: minage
spec:
 replicas: 3
 selector:
  matchLabels:
   app: minage
 template:
  metadata:
   labels:
    app: minage
  spec:
   nodeSelector:
    gpu: nvidia-90HX
   containers:
   - name: mineur
     image: cryptocoin
```

Plutôt que de désigner « directement » un (forcément) unique nœud avec nodeName, il est possible d'en désigner autant que l'on souhaite, indirectement, en les « étiquetant ».

- **Suggérer « simplement » le placement :**

```
  k create -f deploiement_minage.yaml
  k get po -l app==minage -o wide -w
* les pods restent indéfiniment en état « Pending »

  k label no aks-node-vm001 gpu=nvidia-90HX
* les pods trouvent leur « place » sur aks-node-vm001
```

Influencer le scheduler (suite)

Il peut arriver que plusieurs étiquettes (ou combinaisons d'étiquettes) conviennent pour choisir un nœud.

- **Placement, la voie compliquée :**

```
k label no aks-node-vm000 gpu=ASUS77V
k label no aks-node-vm001 gpu=NV745X
k create -f deploiement_minage_complexe.yaml
k get po -l app==minage -o wide --sort-by spec.nodeName
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: minage
 labels: { app: minage }
spec:
 replicas: 3
 selector:
  matchLabels:
   app: minage
 template:
  metadata:
   labels: { app: minage }
  spec:
   affinity:
    nodeAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
       - key: gpu
         operator: In
         values: [ ASUS77V, NV745X ]
   containers: [ { name: mineur, image: cryptocoin } ]
```

Module

Application avec état

StatefulSet, application avec état

Certaines applications qui fonctionnent nativement en mode « clusterisé » ne se satisfont pas de la façon dont travaille les déploiements.
Ces derniers répliquent des pods dont les noms (au moins pour ce qui est de leur suffixe) est aléatoire.
Hors, une application « clusterisée » a, en général, besoin de pouvoir désigner de façon univoque et surtout non-aléatoire les différents « membres » (qui prendront la forme de pods dans Kubernetes) de son cluster.
Les plus représentatives de ces applications sont les moteurs de bases de données. On peut citer Galera Cluster, MongoDB, rqlite, et d'une manière générale la plupart des moteurs NoSQL.
De plus, non content d'avoir besoin de nommer les pods de façon précise et stable (entre deux utilisations de l'application) ont souvent besoin d'être assuré quant à l'ordre dans lequel les pods seront créés (accessoirement supprimés) et souvent aussi voudront accéder à des volumes persistants et avoir la garantie que les pods retrouveront leurs volumes respectifs d'une fois sur l'autre.

StatefulSet, mise en œuvre :

```
---
apiVersion: v1
kind: Service
metadata:
 name: rqlite-interne
spec:
 clusterIP: None   # None => Headless
  publishNotReadyAddresses: true
  selector:
    app: rqlite
  ports:
    - port: 4001
      targetPort: 4001  # TCP par défaut
---
apiVersion: v1
kind: Service
metadata:
 name: rqlite
spec:
  selector:
    app: rqlite
  ports:
    - port: 4001
      targetPort: 4001
```

Module

CI/CD et cas pratique global

Rappels sur GitHub Actions pour la CI/CD

Déploiement continu sur AKS

Gestion des secrets dans GitHub Actions

Cas pratique global

- **Déploiement d'une application conteneurisée sur AKS avec :**
  - Terraform pour l'infrastructure.
  - Configuration des déploiements Kubernetes (manifests).
  - Intégration CI/CD pour automatiser les déploiements.
  - Exposition via Ingress avec certificats SSL.

Iconographie

Ce cours s'accompagne d'une iconographie afin de mieux se repérer :

Sécurité

Accessibilité

Eco-conception

Clean Code

Avertissement