Partie 2 :Déploiement des applications

Le scheduling

Définition : Le scheduling

- Contrôlé par le composant du même nom, la notion de scheduling est la notion qui permet à notre cluster de choisir à quel endroit envoyer un pod sur un cluster.
- Il s'appui pour cela sur les notions d'espace disponible bien sûr mais également sur des **teintes** et des **tolérations**.

Le problème des pods

- En l'état que se passe t-il si le nœud sur lequel le pod est mis est indisponible ?
- Pour le voir on va éteindre un nœud sur lequel nous avons un pod et on va regarder ce qui se passe.

Le problème des pods

- Le pod reste en status Terminating => Pourquoi ?
- Le composant controller va attendre le retour de la node pour déterminer le status du pod pendant 5 minutes, après il est supprimé
- Si on redémarre le nœud il se passe quoi maintenant ?

Le problème des pods

- Le pod n’existe plus !
- En effet le node à signalé à l’API que celui-ci n’existait plus pour lui.
- Pourquoi ? (indice c’est à voir avec docker)
- Quel est le problème de ce comportement ?

La solution à ce problème

- Les Deployment

- Les StatefulSet

Un deployment ?

- Il s’agit d’une ressource qui permet de générer un ou plusieurs pods
- En cas de perte d’un pod celui-ci est reconstruit dans la foulée
- Il génère une autre ressource : Le Replica Set
- Les pods créés seront identiques, mais le nom généré est aléatoire

Un statefulset ?

- Il s’agit d’une ressource qui permet de générer un ou plusieurs pods
- En cas de perte d’un pod celui-ci est reconstruit dans la foulée
- Les pods créés seront identiques, le nom généré sera le nom du pod suivi d’un numéro => Ex : nginx-0

Exercice 2 :

- En vous aidant de la documentation de Kubernetes, créez un deployment qui va créer 3 pod ibmcom/curl:3.6 avec le label app: curl (attention au command)
- De même, faites via un statefulset un pod qui contiendra l'image nginx avec le label app: nginx
- Mettez à jour l'image du deployment avec le tag 4.0.0
- Regardez ce qui se passe
- Mettez à jour l'image avec le tag inexistant 7.4.2
- Regardez ce qui se passe

Les deployment, un outil de développement parfait !

- Les déploiements sont vérifiés par le contrôler à chaque mise à jour
- Il est facile de voir les anciennes version grâce à la commande :
- `kubectl rollout history deployment <nomdeployment>`

- On peut revenir en arrière via la commande
- `kubectl rollout history deployment <nomdeployment> --revision=2`
- `kubectl rollout undo deployment <nomdeployment>`

- On peut aussi associer un HPA au besoin pour augmenter ou diminuer la quantité (on test après)
- https://blog.stephane-robert.info/post/kubernetes-metrics-server/
- `kubectl autoscale deployment <deployment> --cpu-percent=50 --min=1 --max=10`

Le StatefulSet, parfait pour les dev feignants ou les cas spéciaux

- Les pods possèdent des nom spécifiques : Parfait pour tester le status des pods !
- Adapté pour des configuration avec plusieurs hôte précis
- Adapté pour les bases de données qui n’ont pas de stockage en commun

Enfin un peu de scheduling !

- Dans la suite nous allons voir les différentes façon de faire du scheduling, celui via les ressources, celui via les teintes et les tolérations, celui des selector et enfin on va voir comment marche les affinity.

- Supprimez les ressources déployés et reprenez la base saine du deployment nginx mis en exemple dans la documentation des deployment de la documentation Kubernetes :

  wget https://k8s.io/examples/controllers/nginx-deployment.yaml

Les Requests et les Limits

- La gestion de l'allocation des ressources se fait via les options Requests et Limits
- Attention, il se peut que sur le cluster, il y ai des Quotas mis en place sur le cluster
- Par défault, sur les namespace on peut mettre le nombre de cpu et ram par pod par défaut via un LimitRange
- Les limits limitent le nombre de ressource utilisés, le request demande un minimum de ressource pour le démarrage du pod.
- SI la limit est plus grande que le nombre de ressources dans le nœud le pod ne se lance pas

Exercice 3

- Mettez en place un système de quota qui sera à la taille de votre worker sur le namespace
- Tentez de déployer un pod sans mettre de limit ou request et supprimez-le
- Mettez en place un LimitRange
- Tentez à nouveau de déployer le pod
- Mettez à jour le deployment pour mettre 100m CPU (0,1 vCPU) et 50 Gb de RAM => Voyez ce qu'il se passe, dans quel état sont les pods ?
- Mettez des quotas pertinents avec votre environnement.
- Que peut-on en déduire ?

Les teintes et les tolérations

- On va appliquer sur les nœud une teinte qui va permettre de préciser dans quelle condition on va pouvoir déployer sur un nœuds
- Les pods vont porter une tolération : il vont supporter le déploiement sur un nœud contenant la teinte
- Appliqué par défaut sur le master

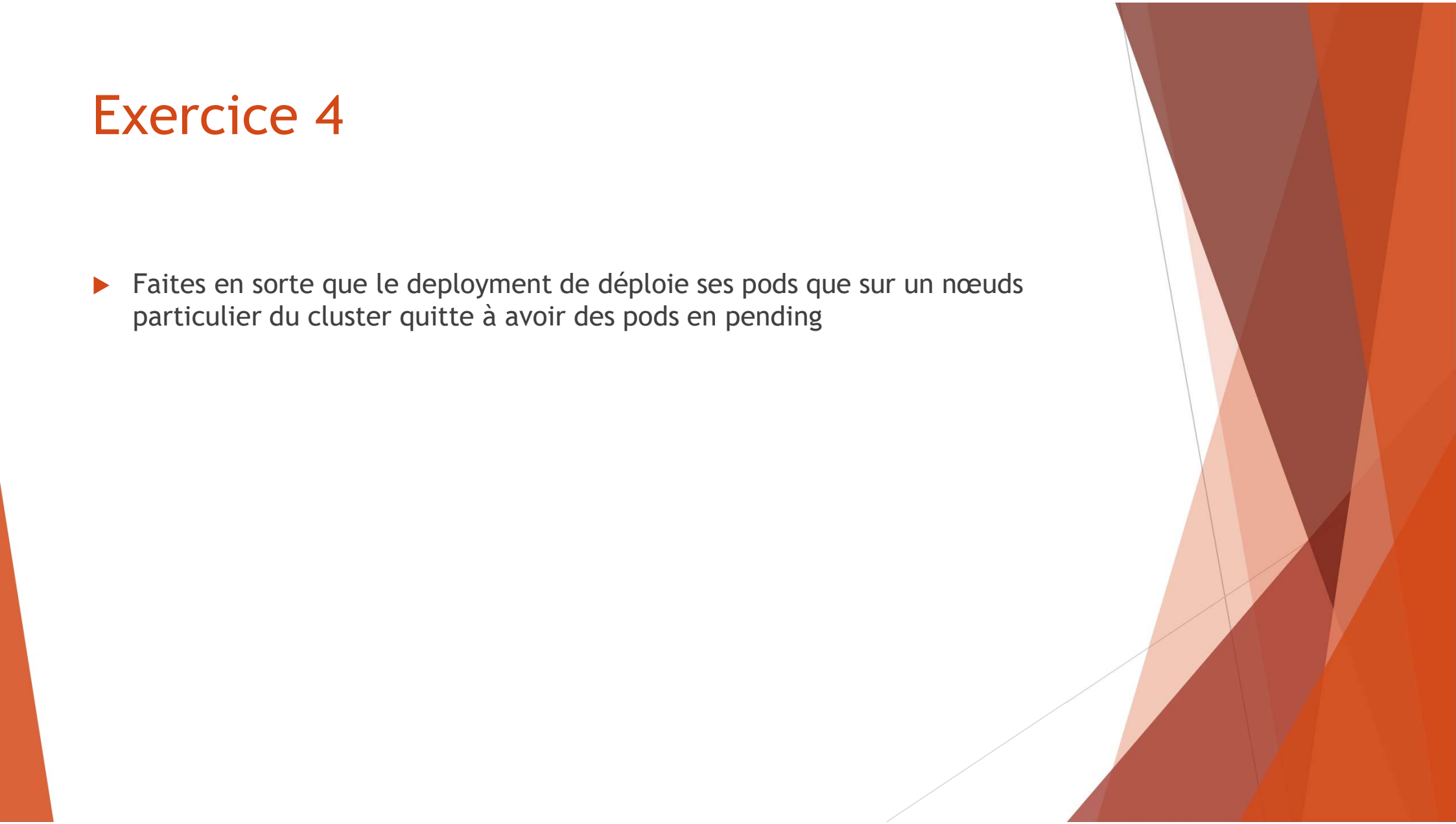
- Explications via draw.io et démonstration

Sélectionner notre cible : le NodeSelector

- Le NodeSelector permet à nos pod de se déployer que sur les pods portant un certain label
- Combinés, ces deux solutions offrent un ciblage précis pour les déploiement
- Peut être pratique en cas de configuration précise du nœuds : ssd, fips….
- Explications via draw.io et démonstration

Exercice 4

- Faites en sorte que le deployment de déploie ses pods que sur un nœuds particulier du cluster quitte à avoir des pods en pending



La deuxième méthode plus complète: le nodeAffinity

- Permet en plusieurs étapes de :
  - Choisir un endroit ou déployer de manière obligatoire ou non
  - Préférer un endroit pour déployer par ordre d'importance
  - Ignorer ou non le changement de label du node

https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

Exercice 5

- Forcez un pod à se déployer sur un certain type de node avec un label que vous aurez positionné. De même, obligez à ce que ce label soit présent pendant l'exécution
- Vérifiez en enlevant ce label

Dans la même veine : Le PodAffinity

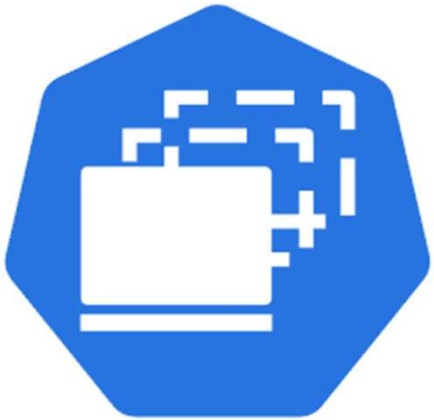
- Se comporte de la même manière que pour les node mais se sépare en deux parties : le PodAffinity et le podAntiAffinity
- Le PodAntiAffinity est pratique pour éviter que deux pods se retrouvent au mêmes endroits

Exercice 6

- Via le pod antiaffinity faites en sorte que deux pods ne puissent pas se déployer ensemble

Un système de déploiement particulier le daemonset

- Un deamonset est une ressource qui permet de créer un pod sur chaque nœud
- Cela peut être pratique pour du monitoring par exemple



Exercice 7

- Déployez un pod sur chaque nœud du cluster

Améliorer notre surveillance des pods : Les sondes

- Il existe deux types de sondes : la readiness et la liveness probe
- La readiness va tenter de savoir si le pod est à mettre à l'état Ready
- Le liveness va voir si le pod est à l'état vivant et le tuer au besoin si la condition n'est pas rempli après un certain laps de temps

- Exemple en cours : https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Exercice 8:

- Sur le pod nginx, vérifiez que le pod 80 est bien actif
- On va donc demander à ce que le readinessProbe soit sur un initial delay de 5 second et une periodSeconds de 3
- Pour la liveness ces argument vont êtres sur 120 et 10

Optimiser ses pods via le stockage persistent

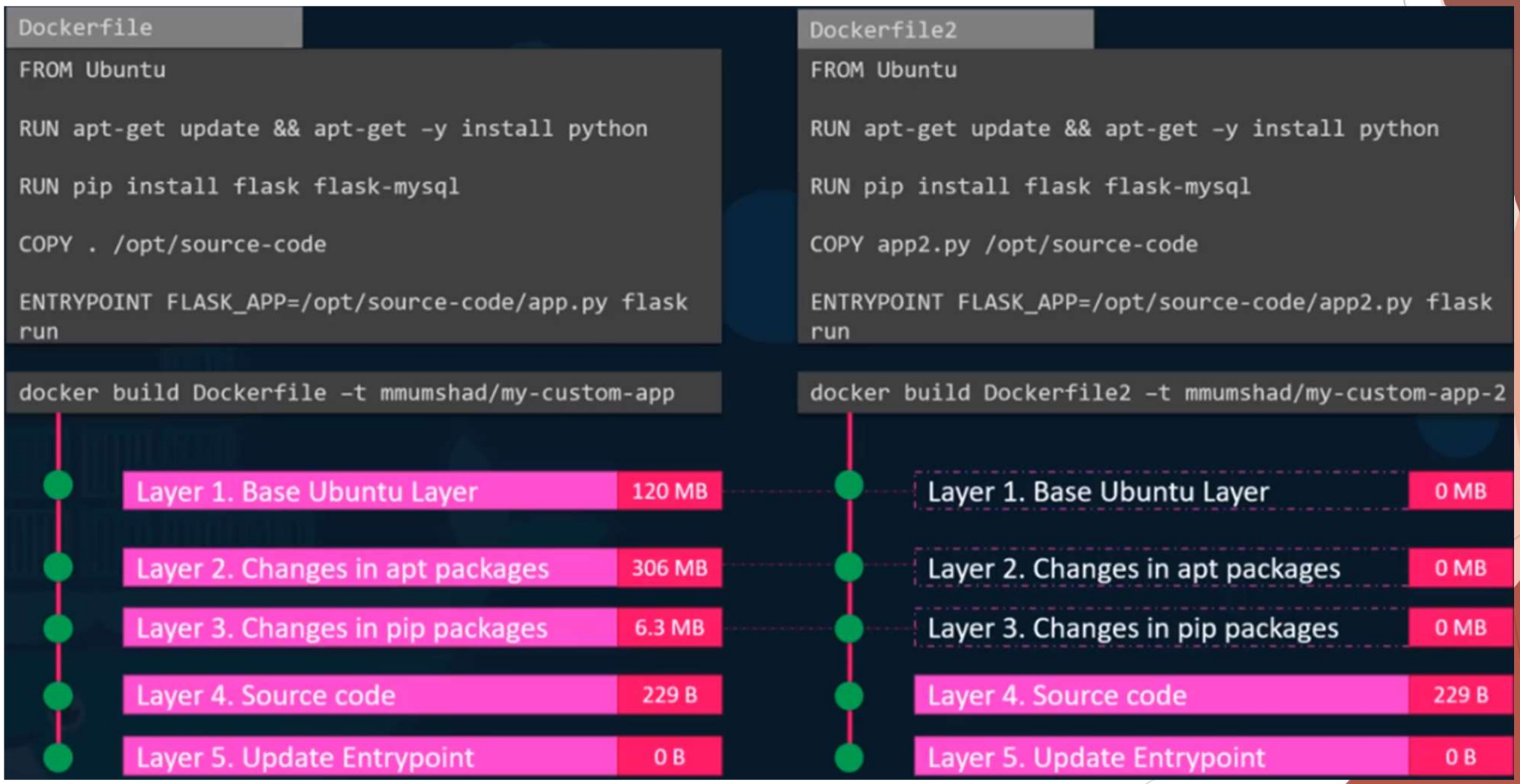
La « problématique » des conteneurs

- Comment fonctionne le stockage sur Docker ?

La « problématique » des conteneurs

- Le stockage fonctionne par couche, ainsi, dans un DockerFile, chaque instruction correspond à une nouvelle couche dans le stockage.
- Ce fonctionnement existe pour économiser de la place car les données des pods sont stockées au même endroit.

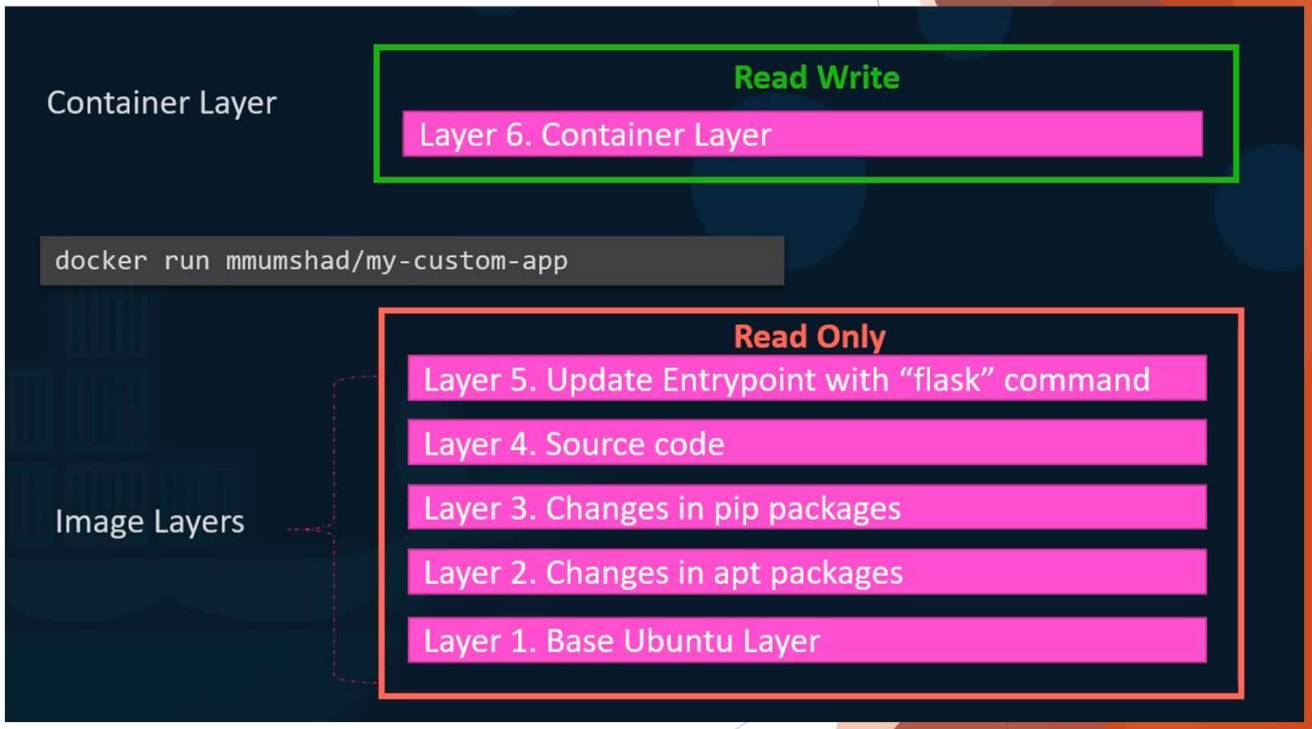
La « problématique » des conteneurs



La « problématique » des conteneurs

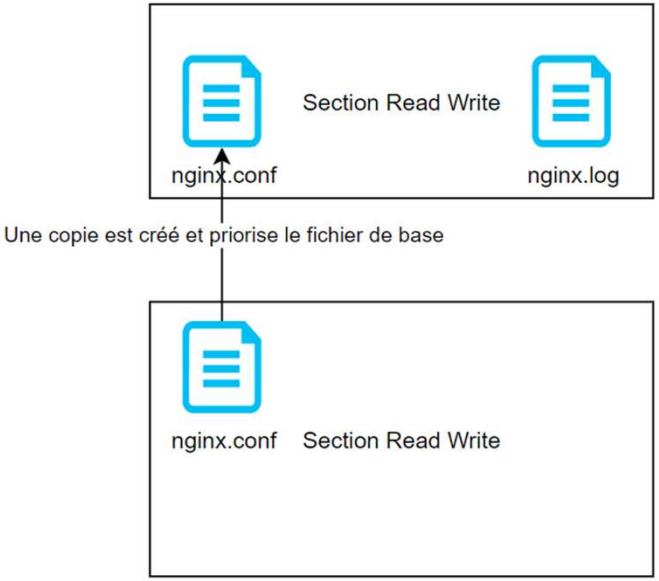
Les couches de l'image sont en Read Only, le seul moyen de modifier la configuration est de reconstruire le DockerFile

Au dessus nous avons donc la couche « Conteurneur » qui est en Read Write mais qui va mourir dès que le Pod va mourir.



Si on veut modifier le code ou les fichiers de l'image de base, comment ça marche?

Problème : La configuration est perdu après le redémarrage



Régler la « problématique » des conteneurs

- Pour résoudre le problème, il faut monter la partie du conteneur qui nous intéresse dans un stockage permanent : Un « volume » sous docker :
- `docker volume create data_volume` => va stocker un volume dans le répertoire /var/lib/docker/volumes
- `docker run -v data_volume:/var/lib/mysql mysql` => Monte le dossier des données mysql du pod dans /var/lib/mysql
- Cela marche aussi pour du stockage sur l'hôte avec par exemple :
- `docker run -v /data/dossierhote:/var/lib/mysql mysql`

Et sur Kubernetes ? Comment on met en place ce mécanisme ?

- Sur Kubernetes, il existe plusieurs objets pour écrire des données permanentes sur un cluster Kubernetes
- Nous avons les configmap et les secrets, ce sont deux objets qui on le même but : accueillir des informations unitaires : identifiant, fichiers…
- Nous avons ensuite les PV, PVC et SC, tous fonctionnent de concert pour apporter des espaces de stockage sur notre conteneur avec des sources de stockage variées (NFS, stockage local, rbd, glusterfs, des Cloud Storage comme gcePersistentDisk, Azure File, awsElasticBlockStore)
- **Attention par conséquent aux coûts cachés !**

Aparté : Avant les configMap et les Secret, les ENV

- Afin de paramétrer certaines applications, des conteneurs ont besoin de variables d'environnement (ex: les bases de données)
- https://hub.docker.com/_/mysql
- Testons ce fichier !

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
  - name: env-print-demo
    image: bash
    env:
    - name: GREETING
      value: "Warm greetings to"
    - name: HONORIFIC
      value: "The Most Honorable"
    - name: NAME
      value: "Kubernetes"
    command: ["echo"]
    args: ["$(GREETING) $(HONORIFIC) $(NAME)"]
```

Les ConfigMap

- Peuvent servir pour stocker des variables d'environnement ou des fichiers
- Exemple de syntaxe :

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cm-test
data:
  env: test
  file.txt: |
    hello
    file
```

Les ConfigMap

- Pour les monter voici un exemple :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-cm
spec:
  container:
    - name: test-cm
      image: alpine
      commande: ["sleep","3600"]
      env:
        - name: CMENV
          valueFrom:
            configMapKeyRef:
              name: cm-test
      volumeMounts:
        - name: cm-test-ref
          mountPath: /home
  volumes:
    - name: cm-test-ref
      configMap:
        name: cm-test
```

Exercice 9:

- Créez un configmap qui permettra pour un pod nginx, de modifier le fichier index.html afin d'afficher un message différent comme « Hello World »

Les secrets

- Reposent sur le même principe que les configmap
- Les données sont encodées en base64

- Pourquoi utiliser cette méthode ?

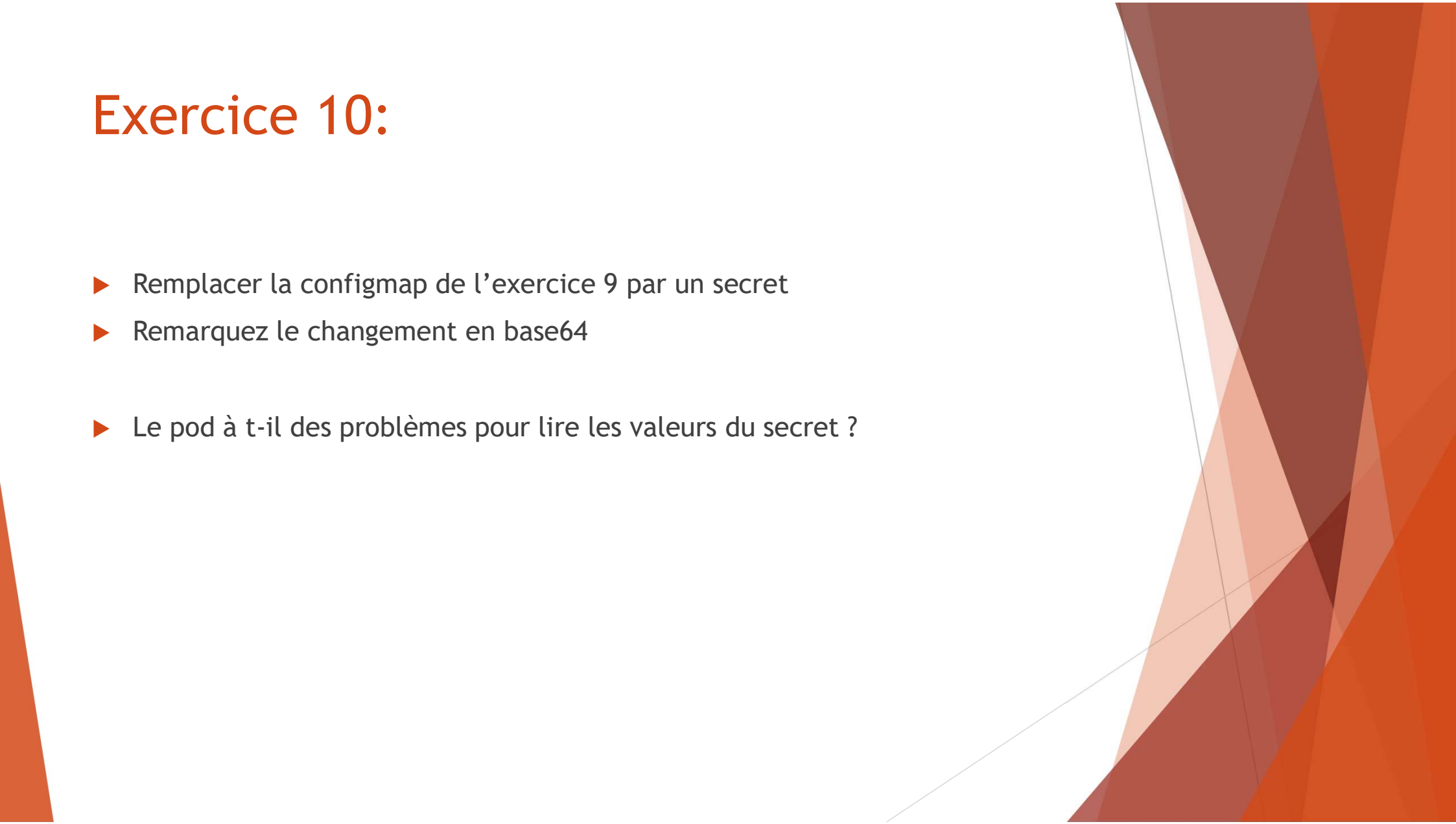
Les secrets

- Les secrets sont des ressources différentes avec des droits différents
- Des solutions tierces telles que Key Vault Manager permettent de stocker les valeurs dans un espace sécurisé.

Exercice 10:

- Remplacer la configmap de l'exercice 9 par un secret
- Remarquez le changement en base64

- Le pod à t-il des problèmes pour lire les valeurs du secret ?



Les SC - Storage Class -

- Il s'agit d'une ressource qui permet de faire le lien entre Kubernetes et un système de stockage comme les stockages Cloud.
- Liste des provisioner disponible : https://kubernetes.io/docs/concepts/storage/storage-classes/#openstack-cinder
- Exemple de configuration avec Cinder pour Openstack:

OpenStack Cinder

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  availability: nova
```

- Permet de provisionner des PV de manière dynamique
- Risque d'un peu de configuration pour les droits

https://openmetal.io/docs/manuals/kubernetes-guides/conf
cinder-with-kubernetes

Les PV – Persistent Volume – et les PVC – Persistent Volume Claim

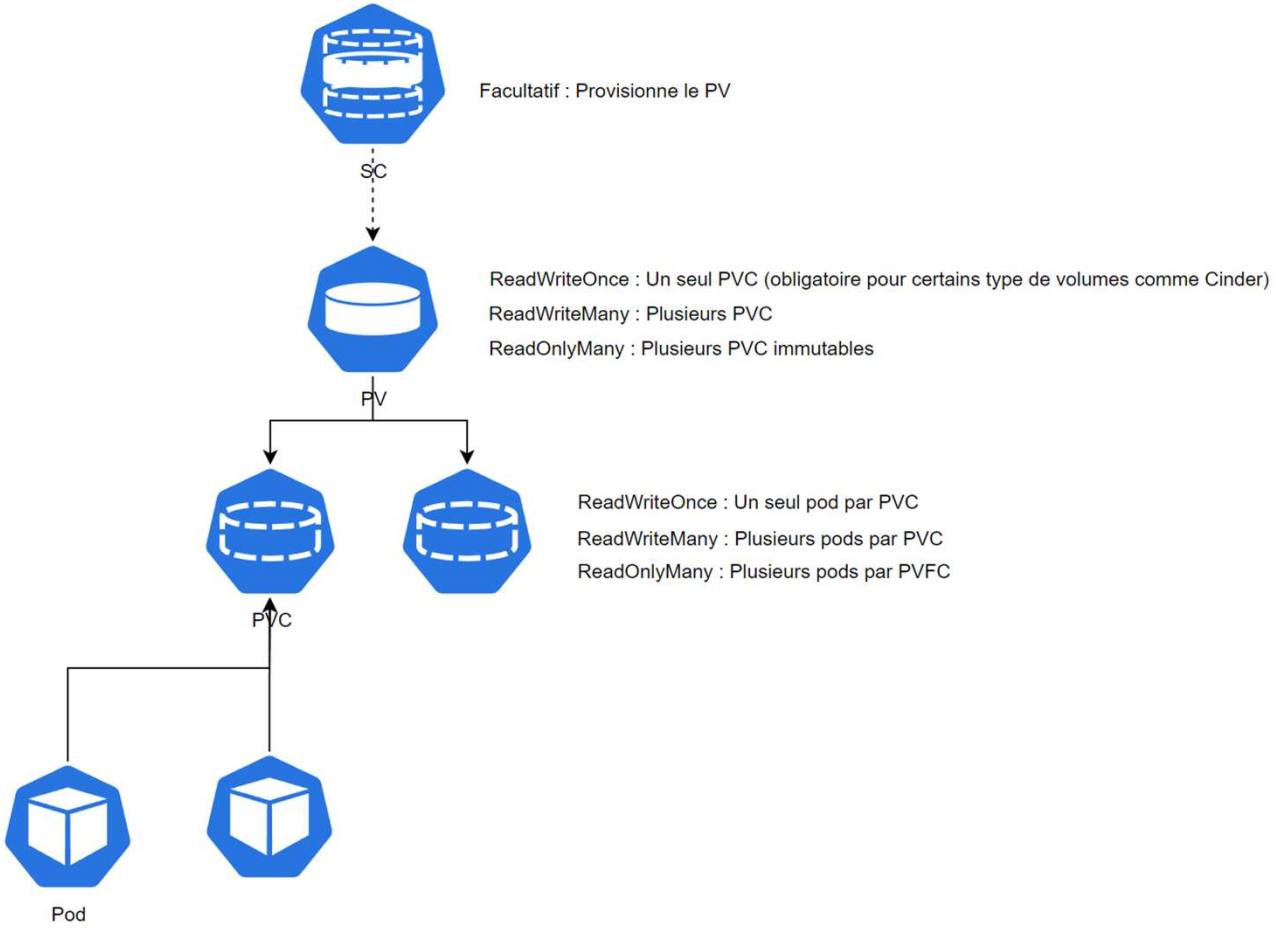
- Il s'agit d'une allocation d'un espace de stockage.
- Sur celui-ci il sera provisionné en fonction du type d'accès à un ou plusieurs PVC
- Les PVC sont les ressources montées par le pod

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Exemple :


Facultatif : Provisionne le PV
SC
ReadWriteOnce : Un seul PVC (obligatoire pour certains type de volumes comme Cinder)
ReadWriteMany : Plusieurs PVC
ReadOnlyMany : Plusieurs PVC immutables
PV
ReadWriteOnce : Un seul pod par PVC
ReadWriteMany : Plusieurs pods par PVC
ReadOnlyMany : Plusieurs pods par PVFC
PVC
Pod

Exercice 10:

- Déployez un conteneur NGINX
- Créez un PV qui va être monté dans le dossier /kube/vol de votre machine (inspirez-vous de pods statiques)
- Associer un PVC à ce PV
- Montez ce PVC à l'endroit ou NGINX met ses logs
- Vérifier la présence des logs sur la machine

FIN PARTIE 2