Cours DevOps - Partie 5 : Orchestration avec Kubernetes

Vue d'ensemble - Points abordés dans cette Partie 5

Aperçu rapide de tout ce qui est vu aujourd'hui

Théories

- **Introduction à Kubernetes**
  - Pourquoi Kubernetes est devenu le standard
  - Différences avec Docker Swarm
  - Cas d'usage en production
- **Architecture Kubernetes**
  - Control Plane (API Server, Scheduler, Controller Manager, etcd)
  - Nodes (Kubelet, Kube-proxy, Container Runtime)
  - Modèle déclaratif et boucle de réconciliation
- **Pods**
  - Unité de base dans Kubernetes
  - Différence avec les conteneurs Docker
  - Multi-container pods
  - Cycle de vie des pods
- **Deployments et ReplicaSets**
  - Gestion déclarative des pods
  - Scaling horizontal
  - Rolling updates et rollbacks
  - Stratégies de déploiement
- **Services**
  - ClusterIP, NodePort, LoadBalancer
  - Service Discovery automatique
  - Sélecteurs et labels
  - DNS interne
- **ConfigMaps et Secrets**
  - Séparation configuration et code
  - Gestion des variables d'environnement
  - Secrets encodés base64
  - Montage en volumes ou variables
- **Persistent Volumes**

    - PersistentVolume (PV) et PersistentVolumeClaim (PVC)
    - StorageClass
    - Différence avec volumes Docker
- **Namespaces**
    - Isolation logique des ressources
    - Multi-tenancy
    - Quotas et limites
- **Introduction à Helm**
    - Package manager pour Kubernetes
    - Charts et templates
    - Releases et versioning

Mises en pratique

- **Installation de Minikube**
    - Setup d'un cluster local
    - kubectl configuration
    - Premier cluster fonctionnel
- **Premiers Pods**
    - Créer un pod nginx
    - Inspecter et débugger
    - Logs et exec
- **Deployments**
    - Créer un deployment
    - Scaler manuellement
    - Rolling update
    - Rollback
- **Services et exposition**
    - ClusterIP pour communication interne
    - NodePort pour accès externe
    - Test de service discovery
- **ConfigMaps et Secrets**
    - Créer des ConfigMaps
    - Créer des Secrets
    - Utiliser dans les pods
- **Persistent Volumes**
    - Créer un PV et PVC
    - Monter dans un pod

    - Test de persistance
- **Projet d'après-midi : API Flask sur Kubernetes**
    - Backend Flask simple
    - PostgreSQL avec PersistentVolume
    - ConfigMaps pour configuration
    - Secrets pour credentials
    - Services pour networking
    - Déploiement complet
    - Scaling et updates
- **Introduction Helm**
    - Installer Helm
    - Déployer un chart public
    - Créer un chart simple
    - Package l'application Flask

---

1 - Introduction : Kubernetes, le Standard de l'Orchestration

L'essentiel

Pourquoi Kubernetes ?

Kubernetes (K8s) est devenu **le standard de facto** pour l'orchestration de conteneurs en production.

Utilisé par :

- Google (créateur original)
- Netflix, Spotify, Airbnb, Uber
- La majorité des entreprises tech
- Tous les cloud providers (AWS EKS, Google GKE, Azure AKS)

Pourquoi ce succès ?

```
✅ Open source et vendor-neutral
✅ Écosystème gigantesque (Helm, Istio, Prometheus, etc.)
✅ Auto-scaling avancé (HPA, VPA, Cluster Autoscaler)
✅ Support multi-cloud natif
✅ Déclaratif et GitOps-friendly
✅ Communauté énorme et documentation riche
✅ Standard CNCF (Cloud Native Computing Foundation)
```

Kubernetes vs Docker Swarm

Vous avez appris Swarm hier. Comparons avec Kubernetes :

Aspect	Docker Swarm	Kubernetes
Complexité	Simple	Complexe
Courbe d'apprentissage	1-2 jours	Plusieurs semaines
Fichiers config	1 fichier YAML	Multiples fichiers YAML
Auto-scaling	Manuel	Automatique (HPA)
Écosystème	Limité	Énorme (Helm, Operators, etc)
Multi-cloud	Possible mais basique	Natif et optimisé
Cas d'usage	PME, simplicité	Production grande échelle
Installation	Inclus avec Docker	Installation séparée
API	Docker API	Kubernetes API

Quand choisir quoi ?

```
Docker Swarm :
└─> Équipe familière avec Docker
└─> Besoin de simplicité
└─> Production PME (< 100 nodes)
└─> Pas besoin de features avancées

Kubernetes :
└─> Production grande échelle
└─> Multi-cloud ou cloud provider
└─> Auto-scaling nécessaire
└─> Équipe DevOps dédiée
└─> Écosystème riche requis
```

Ce que Kubernetes apporte de plus

1. Auto-scaling automatique

```yaml
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: api-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
```

```
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

K8s scale automatiquement de 2 à 10 pods selon la charge CPU.

2. Rolling updates avancés

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1          # Combien de pods en plus pendant l'update
    maxUnavailable: 0    # Combien de pods peuvent être down
```

Contrôle fin sur les déploiements.

3. Health checks granulaires

```
livenessProbe:    # Redémarre le pod si échec
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10

readinessProbe:   # Retire du load balancer si échec
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 5
```

4. Écosystème riche

- **Helm** : Package manager (comme apt pour Linux)
- **Istio** : Service mesh pour microservices
- **Prometheus** : Monitoring natif
- **Cert-manager** : Gestion automatique SSL/TLS
- **Operators** : Automation avancée

Concepts clés à retenir

Déclaratif :

Vous déclarez l'état souhaité, Kubernetes s'occupe de l'atteindre et le maintenir.

```yaml
# Vous dites : "Je veux 3 réplicas de mon API"
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3
  # ...
```

Kubernetes va :

- Créer 3 pods
- Les surveiller en permanence
- Si un pod crash : en recréer un automatiquement
- Si vous changez `replicas: 5` : créer 2 pods de plus

Tout est une ressource :

Dans Kubernetes, tout est représenté par des objets YAML :

- Pods
- Deployments
- Services
- ConfigMaps
- Secrets
- Volumes
- Etc.

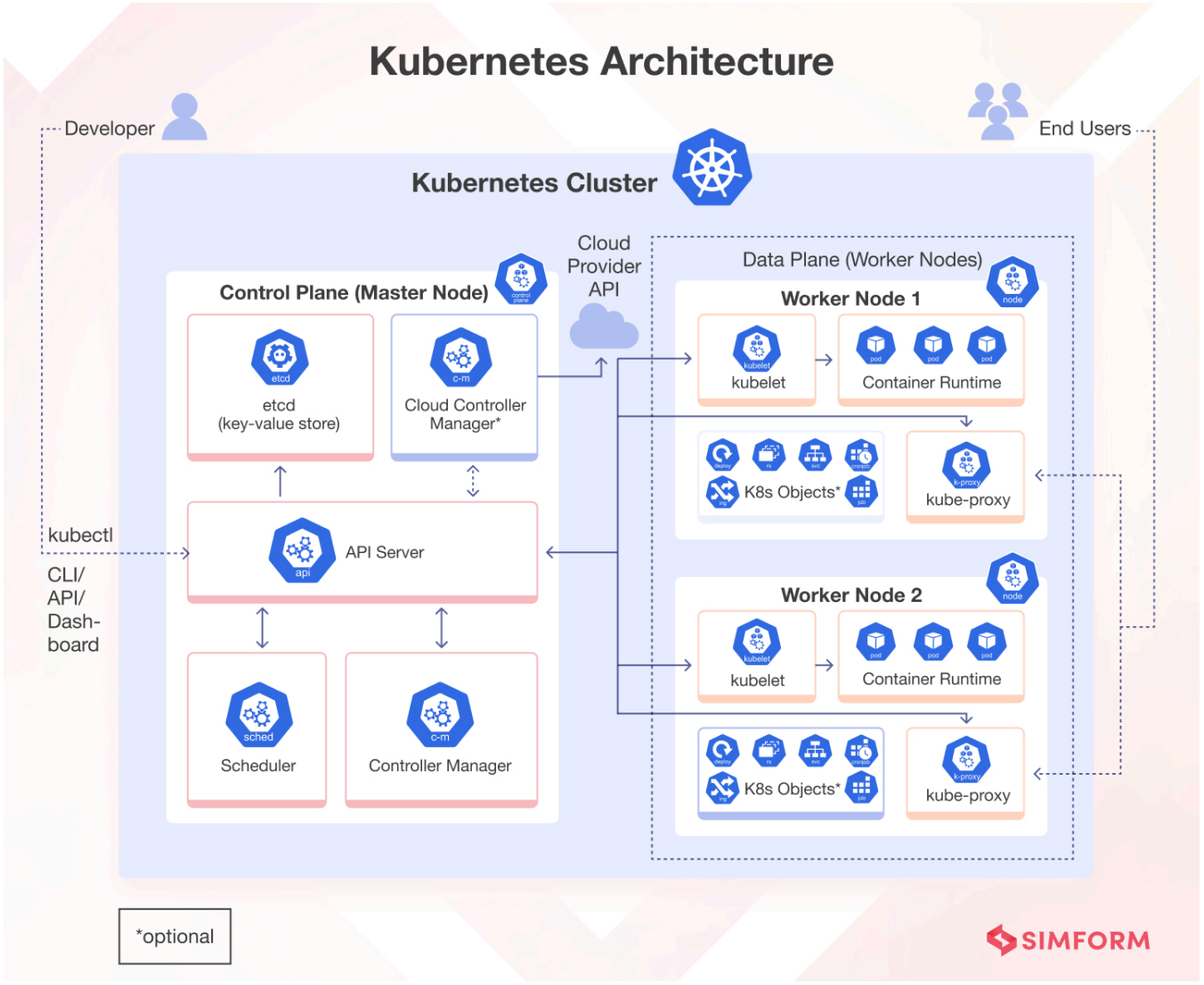
Vous gérez tout via `kubectl` ou fichiers YAML.

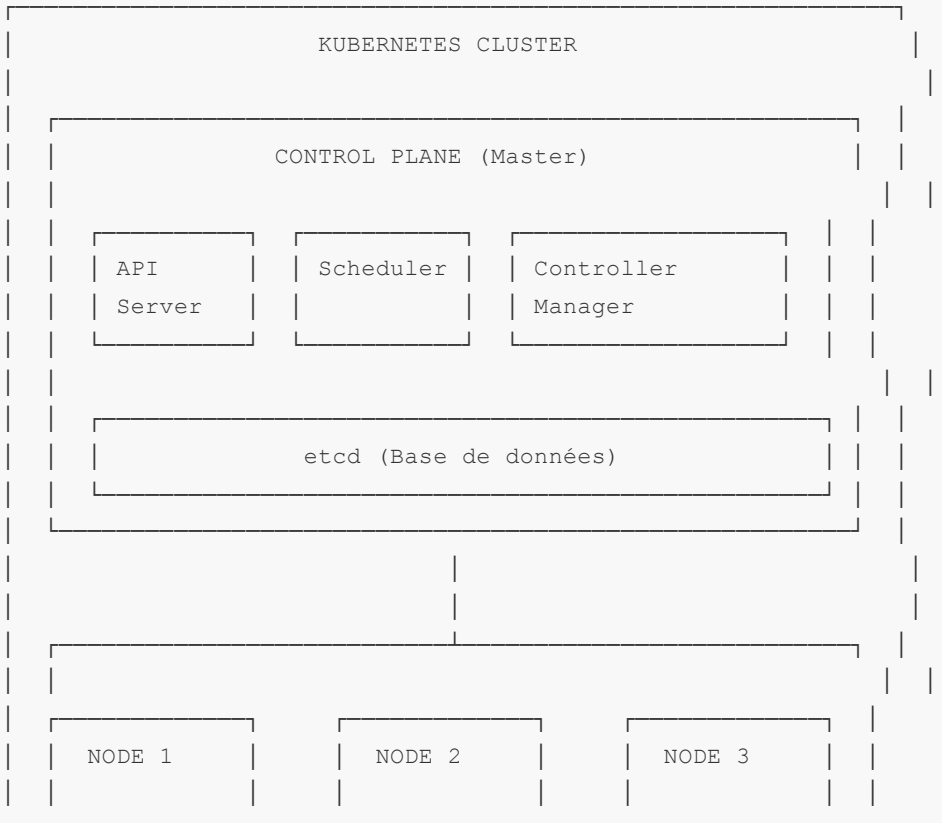
---

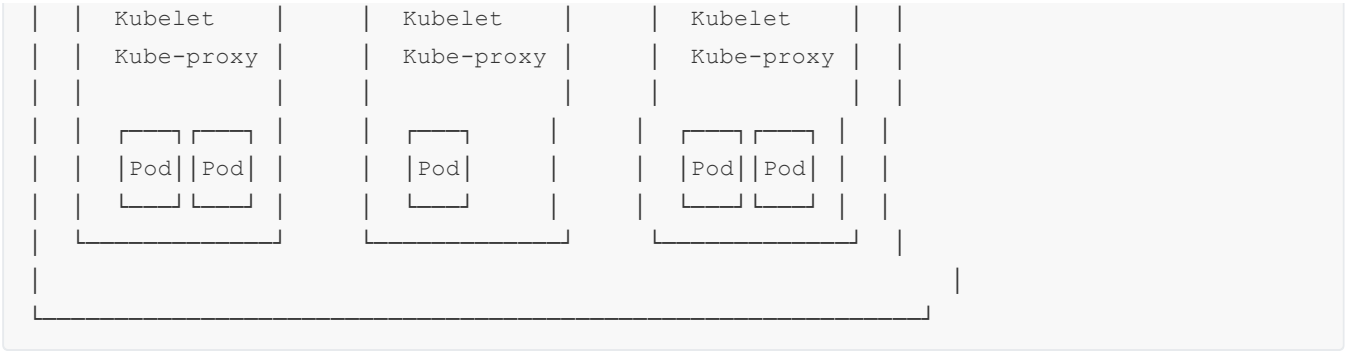
2 - Architecture Kubernetes

L'essentiel

Vue d'ensemble d'un cluster


Kubernetes Architecture
Developer
End Users
Kubernetes Cluster
Cloud Provider API
Control Plane (Master Node)
etcd
(key-value store)
Cloud Controller Manager*
API Server
Scheduler
Controller Manager
Data Plane (Worker Nodes)
Worker Node 1
kubelet
Container Runtime
K8s Objects*
kube-proxy
Worker Node 2
kubelet
Container Runtime
K8s Objects*
kube-proxy
kubectl
CLI/
API/
Dash-
board
*optional
SIMFORM


KUBERNETES CLUSTER
CONTROL PLANE (Master)
API Server
Scheduler
Controller Manager
etcd (Base de données)
NODE 1
NODE 2
NODE 3



Control Plane (Master)

Le **Control Plane** gère le cluster. Il prend les décisions et coordonne tout.

Composants :

1. API Server

```
Rôle : Point d'entrée unique du cluster
- Reçoit toutes les commandes kubectl
- Valide les requêtes
- Communique avec etcd
- Expose l'API REST Kubernetes
```

Quand vous faites `kubectl apply -f deployment.yaml`, c'est l'API Server qui reçoit.

2. etcd

```
Rôle : Base de données distribuée
- Stocke tout l'état du cluster
- Configuration, secrets, état des pods, etc.
- Hautement disponible (consensus Raft)
- Sauvegarde critique
```

Si etcd est perdu, le cluster perd sa mémoire.

3. Scheduler

```
Rôle : Décide où placer les pods
- Analyse les ressources disponibles sur chaque node
- Respecte les contraintes (affinity, taints, tolerations)
- Place le pod sur le meilleur node
```

Exemple : Un pod demande 2 CPU et 4GB RAM. Le Scheduler trouve un node avec ces ressources.

4. Controller Manager

```
Rôle : Surveille l'état et applique les changements
- Boucle de réconciliation permanente
- Si état actuel ≠ état souhaité → agit
- Gère ReplicaSets, Deployments, Services, etc.
```

Exemple : Vous déclarez 3 réplicas, un pod crash, le Controller Manager en recrée un.

Nodes (Workers)

Les **nodes** exécutent les pods (conteneurs applicatifs).

Composants sur chaque node :

1. Kubelet

```
Rôle : Agent qui tourne sur chaque node
- Reçoit les instructions de l'API Server
- Lance les conteneurs via le container runtime
- Surveille les pods et remonte leur état
- Exécute les health checks
```

2. Kube-proxy

```
Rôle : Gère le réseau sur le node
- Configure iptables pour le load balancing
- Permet la communication entre pods
- Gère les Services Kubernetes
```

3. Container Runtime

```
Rôle : Exécute les conteneurs
- Docker, containerd, CRI-O, etc.
- Kubernetes n'est PAS lié à Docker
- Utilise Container Runtime Interface (CRI)
```

Boucle de réconciliation

Le cœur de Kubernetes : la **reconciliation loop**.

```
┌─────────────────────────────────────────┐
│                                         │
│  1. État souhaité (desired state)       │
│     → Défini dans les fichiers YAML     │
│                                         │
│  2. État actuel (current state)         │
│     → Lu depuis etcd et les nodes       │
│                                         │
│  3. Comparaison                         │
│     → Controller Manager compare        │
│                                         │
│  4. Action                              │
│     → Si différence : agir pour converger │
│                                         │
│  5. Répéter en boucle (toutes les secondes)│
│                                         │
└─────────────────────────────────────────┘
```

Exemple concret :

```yaml
# État souhaité
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3  # Je veux 3 pods
```

Boucle :

```
Itération 1 :
- État actuel : 0 pods
- Action : Créer 3 pods

Itération 2 :
- État actuel : 3 pods running
- Action : Rien (état atteint)

Itération 3 :
- État actuel : 2 pods (1 a crashé)
- Action : Créer 1 pod

Itération 4 :
- État actuel : 3 pods running
- Action : Rien
```

Cette boucle tourne **en permanence**, assurant la résilience.

Modèle déclaratif

Différence fondamentale avec les approches impératives :

Impératif (commandes) :

```
kubectl run api --image=mon-api --replicas=3
kubectl expose deployment api --port=80
kubectl scale deployment api --replicas=5
```

Problème : Pas de source de vérité, difficile à reproduire.

Déclaratif (fichiers YAML) :

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3
  # ...
```

```yaml
---
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  # ...
```

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

Avantages :

- Versionné dans Git
- Reproductible
- Auditable
- GitOps-friendly

---

3 - Installation de Minikube et kubectl

L'essentiel

Qu'est-ce que Minikube ?

Minikube est un outil qui lance un cluster Kubernetes local sur votre machine.

```
Parfait pour :
✅ Apprentissage et tests
✅ Développement local
✅ CI/CD pipelines
✅ Démos

Limitations :
❌ Pas pour production
❌ Single-node uniquement (sauf config avancée)
❌ Moins performant qu'un vrai cluster
```

Alternatives :

- k3s : Kubernetes léger
- kind : Kubernetes in Docker
- Docker Desktop : Inclut Kubernetes

Nous utilisons Minikube car il est le plus standard et documenté.

kubectl

kubectl est le CLI pour interagir avec Kubernetes (équivalent de `docker` pour Docker).

```
# Syntaxe générale
kubectl [commande] [type] [nom] [flags]

# Exemples
kubectl get pods
kubectl describe deployment api
kubectl delete service web
kubectl apply -f deployment.yaml
```

Installation

Linux :

```
# Minikube
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube

# kubectl
curl -LO "https://dl.k8s.io/release/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

macOS :

```
# Avec Homebrew
brew install minikube
brew install kubectl
```

Windows :

```
# Avec Chocolatey
choco install minikube
choco install kubernetes-cli
```

Démarrer Minikube

```
# Démarrer le cluster
minikube start
```

Sortie attendue :

```
😄  minikube v1.32.0 on Ubuntu 22.04
✨  Automatically selected the docker driver
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=2200MB) ...
🐳  Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
🔎  Verifying Kubernetes components...
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster
```

Vérifier que tout fonctionne :

```
# Voir les nodes
kubectl get nodes
```

Sortie :

```
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   1m    v1.28.3
```

Un seul node (normal pour Minikube).

Commandes kubectl essentielles

Voir les ressources :

```
# Pods
kubectl get pods

# Deployments
kubectl get deployments

# Services
kubectl get services
kubectl get svc  # Raccourci

# Tout dans un namespace
kubectl get all

# Format détaillé
kubectl get pods -o wide

# Format YAML
kubectl get pod mon-pod -o yaml
```

Inspecter en détail :

```
kubectl describe pod mon-pod
kubectl describe deployment api
kubectl describe service web
```

Logs :

```
# Logs d'un pod
kubectl logs mon-pod

# Suivre les logs en temps réel
kubectl logs -f mon-pod

# Logs d'un pod avec plusieurs conteneurs
kubectl logs mon-pod -c conteneur-specifique
```

Exécuter des commandes dans un pod :

```
# Shell interactif
kubectl exec -it mon-pod -- /bin/sh
kubectl exec -it mon-pod -- /bin/bash

# Commande unique
kubectl exec mon-pod -- ls /app
kubectl exec mon-pod -- cat /etc/config/app.conf
```

Créer/Supprimer des ressources :

```
# Créer depuis un fichier
kubectl apply -f deployment.yaml
kubectl apply -f dossier/  # Tous les fichiers du dossier

# Supprimer
kubectl delete -f deployment.yaml
kubectl delete pod mon-pod
kubectl delete deployment api
```

EXERCICE GUIDÉ 1 : Premier cluster Minikube

Objectif : Installer et configurer Minikube, explorer le cluster

Étape 1 : Installer Minikube et kubectl

Suivre les instructions d'installation selon votre OS.

Vérifier :

```
minikube version
kubectl version --client
```

Étape 2 : Démarrer Minikube

```
minikube start
```

Attendre que le cluster soit prêt.

Étape 3 : Vérifier le cluster

```
kubectl cluster-info
```

Sortie attendue :

```
Kubernetes control plane is running at https://192.168.49.2:8443
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
```

Étape 4 : Lister les nodes

```
kubectl get nodes
```

Vous devriez voir un node "minikube" en état Ready.

Étape 5 : Explorer les namespaces

```
kubectl get namespaces
```

Sortie :

```
NAME              STATUS   AGE
default           Active   5m
kube-node-lease   Active   5m
kube-public       Active   5m
kube-system       Active   5m
```

Étape 6 : Voir les pods système

```
kubectl get pods -n kube-system
```

Vous verrez les composants du Control Plane :

- coredns
- etcd
- kube-apiserver
- kube-controller-manager
- kube-scheduler
- kube-proxy

Étape 7 : Dashboard Minikube (optionnel)

```
minikube dashboard
```

Ouvre une interface web pour visualiser le cluster.

Étape 8 : Statut du cluster

```
minikube status
```

Sortie :

```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Questions checkpoint

- Quelle est la différence entre Minikube et un cluster Kubernetes de production ?
- À quoi sert kubectl ?
- Combien de nodes a un cluster Minikube par défaut ?
- Qu'est-ce qu'un namespace dans Kubernetes ?

---

4 - Pods : L'Unité de Base

L'essentiel

Qu'est-ce qu'un Pod ?

Un **Pod** est la plus petite unité déployable dans Kubernetes.

Différence avec un conteneur Docker :

```
Conteneur Docker :
└─> Processus isolé

Pod Kubernetes :
└─> Groupe de 1 ou plusieurs conteneurs
└─> Partagent le même network namespace
└─> Partagent le même storage
└─> Partagent la même IP
└─> Co-localisés sur le même node
```

La plupart du temps : 1 pod = 1 conteneur

Mais on peut avoir plusieurs conteneurs dans un pod (sidecar pattern).

Anatomie d'un Pod

Fichier YAML minimal :

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

Décortiquons :

- `apiVersion: v1` : Version de l'API K8s
- `kind: Pod` : Type de ressource
- `metadata` : Nom et labels
- `spec` : Spécification du pod
  - `containers` : Liste des conteneurs
    - `name` : Nom du conteneur
    - `image` : Image Docker
    - `ports` : Ports exposés

Créer un Pod

```bash
# Depuis un fichier
kubectl apply -f pod.yaml

# Directement en ligne de commande (impératif)
kubectl run nginx --image=nginx:alpine
```

Vérifier :

```bash
kubectl get pods
```

Sortie :

```
NAME        READY   STATUS    RESTARTS   AGE
nginx-pod   1/1     Running   0          10s
```

Colonnes :

- `READY` : 1/1 = 1 conteneur sur 1 est prêt
- `STATUS` : Running, Pending, Error, CrashLoopBackOff, etc.
- `RESTARTS` : Nombre de redémarrages

- `AGE` : Depuis combien de temps il existe

Inspecter un Pod

```
# Détails complets
kubectl describe pod nginx-pod
```

Affiche :

- Événements (création, scheduling, pulling image, etc.)
- État des conteneurs
- Conditions (PodScheduled, ContainersReady, Ready)
- IP du pod
- Node où il tourne

Voir les logs :

```
kubectl logs nginx-pod
```

Shell dans le pod :

```
kubectl exec -it nginx-pod -- /bin/sh
```

Cycle de vie d'un Pod

```
Pending → ContainerCreating → Running → Succeeded/Failed
                                  ↓
                            CrashLoopBackOff
                            (si crash répété)
```

États :

- **Pending** : En attente de scheduling ou pulling image
- **ContainerCreating** : Conteneur en cours de création
- **Running** : Pod en cours d'exécution
- **Succeeded** : Terminé avec succès (pods à exécution unique)
- **Failed** : Terminé en erreur
- **CrashLoopBackOff** : Redémarre en boucle après des crashes

Multi-container Pods

Exemple : App + sidecar logger

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-sidecar
spec:
```

```
  containers:
  - name: app
    image: mon-api:latest
    ports:
    - containerPort: 8080

  - name: log-collector
    image: fluentd:latest
    volumeMounts:
    - name: logs
      mountPath: /var/log

  volumes:
  - name: logs
    emptyDir: {}
```

Les deux conteneurs :

- Partagent la même IP
- Peuvent communiquer via localhost
- Partagent le volume `logs`

Cas d'usage :

- Sidecar proxy (Envoy, Istio)
- Log collector
- Monitoring agent

Limites des Pods seuls

Problème : Si vous créez un Pod directement et qu'il crash, il **ne redémarre pas automatiquement**.

```
kubectl run nginx --image=nginx:alpine
kubectl delete pod nginx
```

Le pod est supprimé définitivement.

Solution : Utiliser des **Deployments** (section suivante) qui gèrent les Pods automatiquement.

EXERCICE GUIDÉ 2 : Créer et manipuler des Pods

Objectif : Créer un pod nginx, l'inspecter, tester, supprimer

Étape 1 : Créer un fichier pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: web
spec:
  containers:
  - name: nginx
    image: nginx:alpine
    ports:
    - containerPort: 80
```

Étape 2 : Créer le pod

```
kubectl apply -f pod.yaml
```

Étape 3 : Vérifier

```
kubectl get pods
```

Attendre que STATUS soit Running.

Étape 4 : Détails du pod

```
kubectl describe pod nginx
```

Observer :

- Events : Scheduled, Pulling, Pulled, Created, Started
- IP du pod (ex: 10.244.0.5)
- Node où il tourne

Étape 5 : Logs

```
kubectl logs nginx
```

Devrait être vide (nginx n'a reçu aucune requête).

Étape 6 : Test de connexion au pod

Le pod a une IP interne au cluster. Pour y accéder :

```
# Port-forward : rediriger un port local vers le pod
kubectl port-forward nginx 8080:80
```

Dans un autre terminal :

```
curl http://localhost:8080
```

Vous devriez voir la page nginx.

Étape 7 : Exec dans le pod

```
kubectl exec -it nginx -- /bin/sh
```

Dans le pod :

```
ls /usr/share/nginx/html
cat /usr/share/nginx/html/index.html
exit
```

Étape 8 : Modifier le pod (tester l'immutabilité)

Essayer de changer l'image :

```
kubectl edit pod nginx
```

Changer `nginx:alpine` à `nginx:1.25-alpine`.

Sauvegarder et quitter.

Observation : Erreur ! Les pods sont **immutables**. Pour changer l'image, il faut :

1. Supprimer le pod
2. Le recréer avec la nouvelle image

Ou utiliser un Deployment (prochaine section).

Étape 9 : Supprimer le pod

```
kubectl delete pod nginx
```

Vérifier :

```
kubectl get pods
```

Plus de pods.

Questions checkpoint

- Quelle est la différence entre un pod et un conteneur ?
- Pourquoi un pod peut-il contenir plusieurs conteneurs ?
- Comment accéder à un pod depuis votre machine locale ?
- Peut-on modifier l'image d'un pod en cours d'exécution ?

---

5 - Deployments et ReplicaSets

L'essentiel

Le problème avec les Pods seuls

Créer des Pods directement a plusieurs limites :

```
❌ Pas de self-healing : si le pod crash, il ne redémarre pas
❌ Pas de scaling : difficile de gérer plusieurs réplicas
❌ Pas de rolling updates : impossible de mettre à jour sans downtime
❌ Immutables : modification = suppression + recréation
```

Solution : Les **Deployments**.

Qu'est-ce qu'un Deployment ?

Un **Deployment** est un contrôleur qui gère des Pods de manière déclarative.

```
Deployment
└─> Crée un ReplicaSet
    └─> Crée N Pods
```

ReplicaSet : Assure qu'un nombre spécifique de Pods tourne à tout moment.

Deployment : Gère les ReplicaSets et permet les rolling updates.

En pratique : Vous ne créez jamais de Pods ou ReplicaSets directement, vous créez des Deployments.

Anatomie d'un Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3                    # Nombre de pods souhaité
  selector:
    matchLabels:
      app: nginx                 # Sélecteur pour identifier les pods
  template:                      # Template du pod
    metadata:
      labels:
        app: nginx               # Labels des pods créés
    spec:
      containers:
      - name: nginx
        image: nginx:1.25-alpine
        ports:
        - containerPort: 80
```

Sections :

- `replicas: 3` : Kubernetes maintient toujours 3 pods
- `selector.matchLabels` : Comment le Deployment trouve ses pods
- `template` : Le modèle de pod à créer (identique à un Pod YAML)

Créer un Deployment

```
kubectl apply -f deployment.yaml
```

Vérifier :

```
# Voir le deployment
kubectl get deployments
```

Sortie :

```
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           30s
```

Colonnes :

- `READY` : 3/3 = 3 pods prêts sur 3 souhaités
- `UP-TO-DATE` : Pods à jour avec la dernière version
- `AVAILABLE` : Pods disponibles pour servir du trafic

```
# Voir les pods créés
kubectl get pods
```

Sortie :

```
NAME                                READY   STATUS    AGE
nginx-deployment-6d4cf56db6-7xkzq   1/1     Running   30s
nginx-deployment-6d4cf56db6-hmj9r   1/1     Running   30s
nginx-deployment-6d4cf56db6-xt2pl   1/1     Running   30s
```

Les noms sont générés : `[deployment]-[replicaset-hash]-[random]`

Scaling manuel

Augmenter le nombre de réplicas :

```
# Via commande
kubectl scale deployment nginx-deployment --replicas=5

# Ou modifier le fichier YAML et réappliquer
# replicas: 5
kubectl apply -f deployment.yaml
```

Kubernetes crée instantanément 2 pods supplémentaires.

Diminuer :

```
kubectl scale deployment nginx-deployment --replicas=2
```

Kubernetes supprime 3 pods (garde les 2 plus anciens).

Self-healing

Test : Supprimer un pod manuellement.

```
# Lister les pods
kubectl get pods

# Supprimer un pod
kubectl delete pod nginx-deployment-6d4cf56db6-7xkzq
```

Observer :

```
watch kubectl get pods
```

Kubernetes détecte la suppression et **crée immédiatement un nouveau pod** pour maintenir 3 réplicas.

C'est la boucle de réconciliation en action !

Rolling Updates

Mettre à jour l'image :

```
# Via commande
kubectl set image deployment/nginx-deployment nginx=nginx:1.26-alpine

# Ou modifier le YAML et réappliquer
```

Stratégie par défaut :

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1    # Max 1 pod down pendant l'update
      maxSurge: 1          # Max 1 pod en plus temporairement
```

Déroulement :

```
État initial : [v1.25] [v1.25] [v1.25]

Étape 1 : Créer 1 pod v1.26
          [v1.25] [v1.25] [v1.25] [v1.26]
```

```
Étape 2 : Supprimer 1 pod v1.25
          [v1.25] [v1.25] [v1.26]

Étape 3 : Créer 1 pod v1.26
          [v1.25] [v1.25] [v1.26] [v1.26]

Étape 4 : Supprimer 1 pod v1.25
          [v1.25] [v1.26] [v1.26]

Étape 5 : Créer 1 pod v1.26
          [v1.25] [v1.26] [v1.26] [v1.26]

Étape 6 : Supprimer le dernier pod v1.25
          [v1.26] [v1.26] [v1.26]

Résultat : 0 downtime, transition progressive
```

Suivre le rollout :

```
kubectl rollout status deployment/nginx-deployment
```

Sortie :

```
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have
been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have
been updated...
deployment "nginx-deployment" successfully rolled out
```

Rollback

Si la nouvelle version a un problème, revenir en arrière :

```
# Rollback à la version précédente
kubectl rollout undo deployment/nginx-deployment
```

Voir l'historique :

```
kubectl rollout history deployment/nginx-deployment
```

Sortie :

```
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl set image deployment/nginx-deployment nginx=nginx:1.26-alpine
3         kubectl rollout undo deployment/nginx-deployment
```

Rollback à une révision spécifique :

```
kubectl rollout undo deployment/nginx-deployment --to-revision=1
```

Stratégies de déploiement

1. RollingUpdate (par défaut)

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

Transition progressive, 0 downtime.

2. Recreate

```
strategy:
  type: Recreate
```

Supprime tous les anciens pods, puis crée les nouveaux.

Downtime mais utile si les versions ne peuvent pas coexister.

EXERCICE GUIDÉ 3 : Deployments et Rolling Updates

Objectif : Créer un deployment, scaler, faire un rolling update, rollback

Étape 1 : Créer deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: nginx
        image: nginx:1.24-alpine
        ports:
        - containerPort: 80
```

Étape 2 : Créer le deployment

```
kubectl apply -f deployment.yaml
```

Étape 3 : Vérifier

```
kubectl get deployments
kubectl get pods
```

Vous devriez voir 3 pods webapp-xxx.

Étape 4 : Scaler à 5 réplicas

```
kubectl scale deployment webapp --replicas=5
```

Observer :

```
kubectl get pods
```

5 pods maintenant.

Étape 5 : Tester le self-healing

```
# Supprimer un pod
kubectl delete pod webapp-xxx-xxx
```

Immédiatement :

```
kubectl get pods
```

Un nouveau pod est créé. Toujours 5 pods au total.

Étape 6 : Rolling update vers nginx 1.25

```
kubectl set image deployment/webapp nginx=nginx:1.25-alpine
```

Étape 7 : Suivre le rollout

Dans un terminal :

```
watch kubectl get pods
```

Vous verrez les pods v1.24 se terminer progressivement et les pods v1.25 démarrer.

Étape 8 : Vérifier la nouvelle version

```
kubectl describe deployment webapp | grep Image
```

Devrait afficher `nginx:1.25-alpine`.

Étape 9 : Simuler une mauvaise version

```
kubectl set image deployment/webapp nginx=nginx:mauvaise-version
```

Observer :

```
kubectl get pods
```

Certains pods seront en état `ImagePullBackOff` (image introuvable).

Étape 10 : Rollback

```
kubectl rollout undo deployment/webapp
```

Kubernetes revient à nginx:1.25-alpine.

```
kubectl get pods
```

Tous les pods redeviennent Running.

Étape 11 : Voir l'historique

```
kubectl rollout history deployment/webapp
```

Étape 12 : Cleanup

```
kubectl delete deployment webapp
```

Questions checkpoint

- Quelle est la différence entre un Pod et un Deployment ?
- Que se passe-t-il si on supprime manuellement un pod géré par un Deployment ?
- Comment faire un rolling update sans downtime ?
- Comment revenir à une version précédente ?

---

6 - Services : Networking et Load Balancing

L'essentiel

Le problème

Les Pods ont des IPs dynamiques :

```
Pod 1 : 10.244.0.5
Pod 2 : 10.244.0.6
Pod 3 : 10.244.0.7
```

Problèmes :

```
❌ Si un pod redémarre : nouvelle IP
❌ Comment load-balancer entre les 3 pods ?
❌ Comment découvrir les pods disponibles ?
❌ Les IPs ne sont pas stables
```

Solution : Les **Services** Kubernetes.

Qu'est-ce qu'un Service ?

Un **Service** est une abstraction qui définit un ensemble logique de Pods et une politique d'accès.

```
Service "webapp"
├─> IP stable : 10.96.0.10
├─> DNS name : webapp.default.svc.cluster.local
└─> Load balance vers :
    ├─> Pod 1 (10.244.0.5)
    ├─> Pod 2 (10.244.0.6)
    └─> Pod 3 (10.244.0.7)
```

Les clients se connectent au Service (IP stable), le Service route vers les Pods (IPs dynamiques).

Types de Services

1. ClusterIP (par défaut)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  type: ClusterIP           # Accessible uniquement dans le cluster
  selector:
    app: webapp             # Sélectionne les pods avec ce label
  ports:
  - protocol: TCP
    port: 80                # Port du service
    targetPort: 80          # Port des pods
```

Usage : Communication inter-services (API → Database).

2. NodePort

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-nodeport
spec:
  type: NodePort
  selector:
    app: webapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080         # Port exposé sur tous les nodes (30000-32767)
```

Usage : Accès externe pour tests (ex: Minikube).

Accessible via : `http://<node-ip>:30080`

3. LoadBalancer

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp-lb
spec:
  type: LoadBalancer
  selector:
    app: webapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Usage : Production sur cloud providers (AWS, GCP, Azure).

Le cloud provider crée automatiquement un load balancer externe.

Service Discovery

Kubernetes fournit un **DNS interne** automatique.

Format : `<service-name>.<namespace>.svc.cluster.local`

Exemple :

```
Service "database" dans namespace "default"
└─> DNS: database.default.svc.cluster.local
└─> Ou raccourci: database (si même namespace)
```

Dans votre code :

```python
# Python
import psycopg2

# Se connecter via le nom du service
conn = psycopg2.connect(
    host="database",  # Pas d'IP, juste le nom !
    database="mydb",
    user="postgres"
)
```

Kubernetes résout automatiquement `database` vers l'IP du Service.

Sélecteurs et Labels

Les Services trouvent leurs Pods via les **labels**.

```yaml
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  template:
    metadata:
      labels:
        app: webapp       # Label du pod
        version: v1
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
---
# Service
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp           # Sélectionne tous les pods avec app=webapp
  ports:
  - port: 80
    targetPort: 80
```

Le Service envoie le trafic à **tous les pods** ayant `app: webapp`.

Load Balancing

Le Service fait du **load balancing** automatiquement entre les Pods.

```
Client → Service (10.96.0.10:80)
              ├─> Pod 1 (requête 1)
              ├─> Pod 2 (requête 2)
              └─> Pod 3 (requête 3)
```

Algorithme : Round-robin par défaut.

Endpoints

Kubernetes maintient automatiquement une liste d'**Endpoints** (IPs des Pods).

```
kubectl get endpoints webapp-service
```

Sortie :

```
NAME             ENDPOINTS                                      AGE
webapp-service   10.244.0.5:80,10.244.0.6:80,10.244.0.7:80   5m
```

Si un Pod crash ou redémarre, les Endpoints sont mis à jour automatiquement.

EXERCICE GUIDÉ 4 : Services et Networking

Objectif : Créer un deployment + service, tester le load balancing et DNS

Étape 1 : Créer deployment + service

```yaml
# webapp.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
```

```
  type: NodePort
  selector:
    app: webapp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30080
```

Étape 2 : Appliquer

```
kubectl apply -f webapp.yaml
```

Étape 3 : Vérifier

```
kubectl get deployments
kubectl get pods
kubectl get services
```

Sortie services :

```
NAME             TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
webapp-service   NodePort   10.96.45.123   <none>        80:30080/TCP    1m
```

Étape 4 : Tester via NodePort

Sur Minikube :

```
minikube service webapp-service --url
```

Retourne une URL (ex: `http://192.168.49.2:30080`).

```
curl http://192.168.49.2:30080
```

Page nginx.

Étape 5 : Voir les endpoints

```
kubectl get endpoints webapp-service
```

3 IPs listées (les 3 pods).

Étape 6 : Tester le DNS interne

Créer un pod temporaire pour tester :

```
kubectl run test-pod --rm -it --image=alpine -- sh
```

Dans le pod :

```
# Installer curl
apk add curl

# Tester le service via DNS
curl http://webapp-service
```

Devrait afficher la page nginx.

Le nom `webapp-service` est automatiquement résolu par le DNS interne.

```
# Voir la résolution DNS
nslookup webapp-service
```

Affiche l'IP ClusterIP du service.

```
exit
```

Étape 7 : Tester le load balancing

Modifier nginx pour afficher le nom du pod :

On va créer un nouveau deployment avec des pods identifiables.

```
# webapp-custom.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-custom
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp-custom
  template:
    metadata:
      labels:
        app: webapp-custom
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        command: ["/bin/sh"]
        args:
        - -c
        - |
          echo "Hello from pod $(hostname)" > /usr/share/nginx/html/index.html
          nginx -g 'daemon off;'
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: webapp-custom-service
spec:
  type: NodePort
  selector:
    app: webapp-custom
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30081
```

```
kubectl apply -f webapp-custom.yaml
```

Tester :

```
for i in {1..10}; do curl -s http://$(minikube ip):30081; done
```

Sortie :

```
Hello from pod webapp-custom-xxx-aaa
Hello from pod webapp-custom-xxx-bbb
Hello from pod webapp-custom-xxx-ccc
Hello from pod webapp-custom-xxx-aaa
Hello from pod webapp-custom-xxx-bbb
...
```

Le trafic est distribué entre les 3 pods.

Étape 8 : Cleanup

```
kubectl delete -f webapp.yaml
kubectl delete -f webapp-custom.yaml
```

Questions checkpoint

- Quelle est la différence entre ClusterIP, NodePort et LoadBalancer ?
- Comment les Services trouvent-ils leurs Pods ?
- Qu'est-ce que le Service Discovery ?
- Comment fonctionne le load balancing dans un Service ?

---

7 - ConfigMaps et Secrets

L'essentiel

Le problème

Hard-coder la configuration = mauvaise pratique :

```
containers:
- name: app
  image: mon-api
  env:
  - name: DATABASE_URL
    value: "postgres://db:5432/mydb"  # Hard-codé !
  - name: API_KEY
    value: "secret123"                # Mot de passe en clair !
```

Problèmes :

```
❌ Configuration mélangée avec le code
❌ Impossible de changer sans rebuild l'image
❌ Secrets visibles en clair
❌ Pas de réutilisation
```

Solutions : ConfigMaps et Secrets.

ConfigMaps : Configuration non-sensible

Un **ConfigMap** stocke des données de configuration (variables, fichiers).

Créer un ConfigMap :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  DATABASE_HOST: "database"
  DATABASE_NAME: "mydb"
  LOG_LEVEL: "info"
  APP_ENV: "production"
```

```
kubectl apply -f configmap.yaml
```

Ou depuis un fichier :

```
# config.properties
database.host=database
database.name=mydb
log.level=info

kubectl create configmap app-config --from-file=config.properties
```

Utiliser dans un Pod (variables d'environnement) :

```
apiVersion: v1
kind: Pod
metadata:
  name: app
spec:
  containers:
  - name: app
    image: mon-api
    envFrom:
    - configMapRef:
        name: app-config  # Injecte toutes les clés comme variables
```

Ou sélectivement :

```
env:
- name: DB_HOST
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: DATABASE_HOST
```

Utiliser comme volume (fichier) :

```
containers:
- name: app
  image: mon-api
  volumeMounts:
  - name: config-volume
    mountPath: /etc/config

volumes:
- name: config-volume
  configMap:
    name: app-config
```

Le fichier sera monté à `/etc/config/DATABASE_HOST`, etc.

Secrets : Données sensibles

Un **Secret** stocke des données sensibles (mots de passe, tokens, certificats).

Différence avec ConfigMap :

```
ConfigMap : Données non-sensibles (configuration)
Secret : Données sensibles (encodées base64)
```

Créer un Secret :

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  password: cGFzc3dvcmQxMjM=  # "password123" encodé en base64
  api-key: YWJjZGVmZ2hpamts
```

Encoder en base64 :

```
echo -n "password123" | base64
# Résultat : cGFzc3dvcmQxMjM=
```

Créer depuis la ligne de commande :

```
kubectl create secret generic db-secret \
  --from-literal=password=password123 \
  --from-literal=api-key=abcdefghijkl
```

Utiliser dans un Pod :

```yaml
containers:
- name: app
  image: mon-api
  env:
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: db-secret
        key: password
```

Ou comme volume :

```yaml
containers:
- name: app
  image: mon-api
  volumeMounts:
  - name: secret-volume
    mountPath: /run/secrets
    readOnly: true

volumes:
- name: secret-volume
  secret:
    secretName: db-secret
```

Le secret sera monté à `/run/secrets/password` et `/run/secrets/api-key`.

Important : Sécurité des Secrets

Les Secrets ne sont PAS chiffrés par défaut dans etcd !

Ils sont seulement encodés en base64 (facilement décodable).

Pour une vraie sécurité :

- Activer le chiffrement at-rest dans etcd
- Utiliser des outils externes : Vault, Sealed Secrets, External Secrets Operator
- Limiter l'accès via RBAC

Base64 n'est PAS du chiffrement :

```bash
echo "cGFzc3dvcmQxMjM=" | base64 -d
# Résultat : password123
```

Bonnes pratiques

```
✅ Séparer configuration et secrets
✅ Utiliser ConfigMaps pour config non-sensible
✅ Utiliser Secrets pour données sensibles
✅ Ne jamais committer les Secrets en clair dans Git
✅ Utiliser des outils de gestion de secrets en production
✅ Monter les Secrets en volumes plutôt qu'en variables d'env
   (les variables sont visibles dans docker inspect)
```

EXERCICE GUIDÉ 5 : ConfigMaps et Secrets

Objectif : Créer un ConfigMap et un Secret, les utiliser dans un pod

Étape 1 : Créer un ConfigMap

```yaml
# configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_NAME: "My Awesome App"
  APP_ENV: "development"
  LOG_LEVEL: "debug"
```

```
kubectl apply -f configmap.yaml
```

Vérifier :

```
kubectl get configmaps
kubectl describe configmap app-config
```

Étape 2 : Créer un Secret

```
kubectl create secret generic app-secret \
  --from-literal=db-password=supersecret \
  --from-literal=api-token=abc123def456
```

Vérifier :

```
kubectl get secrets
kubectl describe secret app-secret
```

Note : `kubectl describe` n'affiche PAS les valeurs (sécurité).

Pour voir les valeurs :

```
kubectl get secret app-secret -o yaml
```

Les valeurs sont en base64.

Étape 3 : Créer un pod qui utilise ConfigMap et Secret

```yaml
# pod-with-config.yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: alpine
    command: ["/bin/sh"]
    args:
    - -c
```

```
    - |
      echo "APP_NAME: $APP_NAME"
      echo "APP_ENV: $APP_ENV"
      echo "LOG_LEVEL: $LOG_LEVEL"
      echo "DB_PASSWORD: $DB_PASSWORD"
      echo "API_TOKEN: $API_TOKEN"
      echo "Secret from file: $(cat /run/secrets/db-password)"
      sleep 3600
    env:
    - name: APP_NAME
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: APP_NAME
    - name: APP_ENV
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: APP_ENV
    - name: LOG_LEVEL
      valueFrom:
        configMapKeyRef:
          name: app-config
          key: LOG_LEVEL
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: app-secret
          key: db-password
    - name: API_TOKEN
      valueFrom:
        secretKeyRef:
          name: app-secret
          key: api-token
    volumeMounts:
    - name: secret-volume
      mountPath: /run/secrets
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: app-secret
```

```
kubectl apply -f pod-with-config.yaml
```

Étape 4 : Vérifier les variables

```
kubectl logs app-pod
```

Sortie :

```
APP_NAME: My Awesome App
APP_ENV: development
LOG_LEVEL: debug
DB_PASSWORD: supersecret
API_TOKEN: abc123def456
Secret from file: supersecret
```

Étape 5 : Exec dans le pod

```
kubectl exec -it app-pod -- /bin/sh
```

Dans le pod :

```
# Variables d'environnement
env | grep APP

# Fichiers secrets
ls /run/secrets/
cat /run/secrets/db-password
cat /run/secrets/api-token

exit
```

Étape 6 : Modifier le ConfigMap

```
kubectl edit configmap app-config
```

Changer `LOG_LEVEL: "info"`.

Sauvegarder et quitter.

Important : Les variables d'environnement ne sont PAS mises à jour automatiquement.

Il faut redémarrer le pod :

```
kubectl delete pod app-pod
kubectl apply -f pod-with-config.yaml
kubectl logs app-pod
```

Maintenant `LOG_LEVEL: info`.

Mais : Les volumes sont mis à jour automatiquement (après quelques secondes).

Étape 7 : Cleanup

```
kubectl delete pod app-pod
kubectl delete configmap app-config
kubectl delete secret app-secret
```

Questions checkpoint

- Quelle est la différence entre un ConfigMap et un Secret ?
- Les Secrets sont-ils vraiment sécurisés par défaut ?
- Comment monter un Secret en tant que fichier ?
- Les variables d'environnement sont-elles mises à jour automatiquement ?

---

8 - Persistent Volumes

L'essentiel

Problème :

Les conteneurs sont éphémères. Si un pod PostgreSQL meurt, toutes les données sont perdues.

Solution Kubernetes :

- **PersistentVolume (PV)** : Stockage physique (disque local, NFS, cloud storage)
- **PersistentVolumeClaim (PVC)** : Demande de stockage par un pod
- **StorageClass** : Provisionnement dynamique de volumes

Schéma :

```
Pod → PVC (demande 10Gi) → PV (volume créé) → Stockage physique
```

Différence avec Docker volumes :

- Docker : volume = dossier sur l'hôte
- K8s : abstraction, peut être cloud storage, NFS, etc.

Types de PersistentVolumes

Types courants :

- `hostPath` : Dossier sur le node (Minikube uniquement)
- `nfs` : Network File System
- `awsElasticBlockStore` : AWS EBS
- `gcePersistentDisk` : Google Cloud Persistent Disk
- `azureDisk` : Azure Disk

Modes d'accès :

- `ReadWriteOnce (RWO)` : Un seul node en lecture/écriture
- `ReadOnlyMany (ROX)` : Plusieurs nodes en lecture seule
- `ReadWriteMany (RWX)` : Plusieurs nodes en lecture/écriture

Politiques de récupération :

- `Retain` : Garde les données après suppression du PVC
- `Delete` : Supprime le volume après suppression du PVC
- `Recycle` : Efface les données et réutilise le volume (déprécié)

PersistentVolume statique

Fichier `pv.yaml` :

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: postgres-pv
spec:
  capacity:
    storage: 5Gi
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  hostPath:
    path: /mnt/data/postgres
```

Créer le PV :

```
kubectl apply -f pv.yaml
kubectl get pv
```

Sortie :

```
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      AGE
postgres-pv   5Gi        RWO            Retain           Available   5s
```

STATUS :

- `Available` : Disponible
- `Bound` : Lié à un PVC
- `Released` : PVC supprimé, données encore présentes
- `Failed` : Erreur

PersistentVolumeClaim

Le pod ne parle PAS directement au PV.

Le pod demande du stockage via un **PVC**, et Kubernetes trouve un PV compatible.

Fichier `pvc.yaml` :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Créer le PVC :

```
kubectl apply -f pvc.yaml
kubectl get pvc
```

Sortie :

```
NAME           STATUS   VOLUME        CAPACITY   ACCESS MODES   AGE
postgres-pvc   Bound    postgres-pv   5Gi        RWO            3s
```

STATUS :

- `Pending` : Aucun PV compatible trouvé
- `Bound` : Lié à un PV

Vérifier le PV :

```
kubectl get pv
```

Sortie :

```
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
AGE
postgres-pv   5Gi        RWO            Retain           Bound    default/postgres-pvc
1m
```

Le PV est maintenant `Bound` au PVC `default/postgres-pvc`.

Utiliser un PVC dans un Pod

Fichier `postgres-pod.yaml` :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: postgres
spec:
  containers:
  - name: postgres
```

```
    image: postgres:14-alpine
    env:
    - name: POSTGRES_PASSWORD
      value: mysecretpassword
    volumeMounts:
    - name: postgres-storage
      mountPath: /var/lib/postgresql/data
  volumes:
  - name: postgres-storage
    persistentVolumeClaim:
      claimName: postgres-pvc
```

Créer le pod :

```
kubectl apply -f postgres-pod.yaml
kubectl get pods
```

Vérifier que les données persistent :

```
# Exec dans le pod
kubectl exec -it postgres -- psql -U postgres

# Créer une table
CREATE TABLE test (id SERIAL PRIMARY KEY, name TEXT);
INSERT INTO test (name) VALUES ('hello');
SELECT * FROM test;

# Quitter
\q
exit

# Supprimer le pod
kubectl delete pod postgres

# Recréer le pod
kubectl apply -f postgres-pod.yaml

# Vérifier les données
kubectl exec -it postgres -- psql -U postgres -c "SELECT * FROM test;"
```

Résultat :

```
 id | name
----+-------
  1 | hello
```

Les données ont persisté.

StorageClass et provisionnement dynamique

Problème :

Créer manuellement un PV pour chaque PVC, c'est lourd.

Solution : StorageClass

Kubernetes crée automatiquement un PV quand un PVC est demandé.

Vérifier les StorageClass disponibles :

```
kubectl get storageclass
```

Sortie (Minikube) :

```
NAME                 PROVISIONER                RECLAIMPOLICY   AGE
standard (default)   k8s.io/minikube-hostpath   Delete          10m
```

Créer un PVC avec StorageClass :

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc-dynamic
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: standard
```

Appliquer :

```
kubectl apply -f pvc-dynamic.yaml
kubectl get pvc
kubectl get pv
```

Un PV a été créé automatiquement.

Avantage :

Pas besoin de créer manuellement les PV. Kubernetes les crée à la demande.

En production :

Les StorageClass sont configurées pour AWS EBS, GCP Persistent Disk, Azure Disk, etc.

Exercice guidé : PostgreSQL avec PersistentVolume

Objectif :

Déployer PostgreSQL avec un volume persistant.

Étape 1 : Créer un PVC

Fichier `postgres-pvc.yaml` :

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  storageClassName: standard
```

```
kubectl apply -f postgres-pvc.yaml
kubectl get pvc
```

Étape 2 : Créer un Deployment PostgreSQL

Fichier `postgres-deployment.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:14-alpine
        env:
        - name: POSTGRES_PASSWORD
          value: mysecretpassword
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        ports:
        - containerPort: 5432
```

```
        volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: postgres-storage
        persistentVolumeClaim:
          claimName: postgres-pvc
```

Note importante :

`PGDATA: /var/lib/postgresql/data/pgdata` est nécessaire pour PostgreSQL.

Sans ça, PostgreSQL refuse de démarrer car `/var/lib/postgresql/data` contient des fichiers K8s (lost+found).

```
kubectl apply -f postgres-deployment.yaml
kubectl get pods
```

Étape 3 : Créer un Service

Fichier `postgres-service.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
  type: ClusterIP
```

```
kubectl apply -f postgres-service.yaml
kubectl get svc
```

Étape 4 : Tester la persistance

Créer une table :

```
kubectl exec -it deployment/postgres -- psql -U postgres
```

Dans psql :

```
CREATE DATABASE testdb;
\c testdb
CREATE TABLE users (id SERIAL PRIMARY KEY, name TEXT);
INSERT INTO users (name) VALUES ('Alice'), ('Bob');
SELECT * FROM users;
\q
```

Étape 5 : Supprimer le pod

```
kubectl delete pod -l app=postgres
kubectl get pods -w
```

Le Deployment recrée automatiquement un nouveau pod.

Étape 6 : Vérifier les données

```
kubectl exec -it deployment/postgres -- psql -U postgres -d testdb -c "SELECT * FROM users;"
```

Résultat :

```
 id | name
----+-------
  1 | Alice
  2 | Bob
```

Les données ont persisté.

Étape 7 : Cleanup

```
kubectl delete deployment postgres
kubectl delete service postgres
kubectl delete pvc postgres-pvc
```

Note :

Avec `Retain` policy, le PV reste disponible même après suppression du PVC.

Avec `Delete` policy (par défaut sur `standard` StorageClass), le PV est supprimé.

Questions checkpoint

- Quelle est la différence entre un PV et un PVC ?
- Pourquoi utiliser un PVC plutôt que de monter directement un PV dans un pod ?
- Qu'est-ce qu'un StorageClass ?
- Que se passe-t-il si on supprime un PVC avec `Retain` policy ?
- Pourquoi PostgreSQL nécessite-t-il la variable `PGDATA` dans Kubernetes ?

---

9 - Projet : Stack Flask sur Kubernetes

Objectif

Déployer une stack complète sur Minikube avec :

- **Backend** : API Flask (Python)
- **Frontend** : Nginx servant du HTML/CSS/JS statique
- **Base de données** : PostgreSQL avec PersistentVolume
- **Cache** : Redis (optionnel)

Pourquoi Flask et pas Rails ?

- Dockerisation simple et directe
- Pas de problèmes d'installation Windows/Mac
- Code minimal, focus sur Kubernetes
- Compatible multi-OS via Docker

Progression :

1. Créer la structure du projet
2. Dockeriser chaque service
3. Tester en local avec docker-compose
4. Écrire les manifests Kubernetes
5. Déployer sur Minikube
6. Configurer ConfigMaps et Secrets
7. Tester scaling et résilience
8. Bonus : Ingress pour exposer l'application

Étape 1 : Structure du projet

Créer la structure :

```
mkdir flask-k8s-stack
cd flask-k8s-stack

# Backend Flask
mkdir -p backend
touch backend/app.py backend/requirements.txt backend/Dockerfile

# Frontend Nginx
mkdir -p frontend/html
touch frontend/Dockerfile frontend/nginx.conf frontend/html/index.html
frontend/html/app.js

# Manifests Kubernetes
mkdir k8s
```

Étape 2 : Backend Flask

Fichier `backend/app.py` :

```
from flask import Flask, jsonify, request
from flask_cors import CORS
import psycopg2
import os
import redis

app = Flask(__name__)
CORS(app)

# Configuration depuis les variables d'environnement
DB_HOST = os.getenv('DATABASE_HOST', 'localhost')
DB_NAME = os.getenv('DATABASE_NAME', 'mydb')
DB_USER = os.getenv('DATABASE_USER', 'postgres')
DB_PASSWORD = os.getenv('DATABASE_PASSWORD', 'password')
REDIS_HOST = os.getenv('REDIS_HOST', 'localhost')

# Connexion Redis
try:
    cache = redis.Redis(host=REDIS_HOST, port=6379, decode_responses=True)
except:
    cache = None

def get_db_connection():
    return psycopg2.connect(
        host=DB_HOST,
        database=DB_NAME,
        user=DB_USER,
        password=DB_PASSWORD
    )

@app.route('/health')
def health():
    return jsonify({"status": "healthy"}), 200

@app.route('/api/items', methods=['GET'])
def get_items():
    # Vérifier le cache
    if cache:
        cached_items = cache.get('items')
        if cached_items:
            return jsonify({"items": eval(cached_items), "source": "cache"}), 200

    # Sinon, requête BDD
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute('SELECT id, name FROM items ORDER BY id')
    items = [{"id": row[0], "name": row[1]} for row in cur.fetchall()]
    cur.close()
    conn.close()
```

```
    # Mettre en cache
    if cache:
        cache.setex('items', 60, str(items))

    return jsonify({"items": items, "source": "database"}), 200

@app.route('/api/items', methods=['POST'])
def create_item():
    data = request.json
    name = data.get('name')

    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute('INSERT INTO items (name) VALUES (%s) RETURNING id', (name,))
    item_id = cur.fetchone()[0]
    conn.commit()
    cur.close()
    conn.close()

    # Invalider le cache
    if cache:
        cache.delete('items')

    return jsonify({"id": item_id, "name": name}), 201

@app.route('/init-db', methods=['POST'])
def init_db():
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute('''
        CREATE TABLE IF NOT EXISTS items (
            id SERIAL PRIMARY KEY,
            name TEXT NOT NULL
        )
    ''')
    conn.commit()
    cur.close()
    conn.close()
    return jsonify({"message": "Database initialized"}), 200

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Fichier `backend/requirements.txt` :

```
Flask==2.3.0
flask-cors==4.0.0
psycopg2-binary==2.9.6
redis==4.5.5
```

Fichier `backend/Dockerfile` :

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

EXPOSE 5000

CMD ["python", "app.py"]
```

Étape 3 : Frontend Nginx

Fichier `frontend/html/index.html` :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Flask K8s Stack</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 800px;
            margin: 50px auto;
            padding: 20px;
        }
        h1 { color: #326ce5; }
        button {
            padding: 10px 20px;
            margin: 5px;
            cursor: pointer;
            background-color: #326ce5;
            color: white;
            border: none;
            border-radius: 4px;
        }
        button:hover { background-color: #285bb5; }
        input {
            padding: 10px;
            width: 300px;
            margin: 5px;
        }
        #items {
            margin-top: 20px;
            padding: 10px;
            background-color: #f4f4f4;
            border-radius: 4px;
        }
```

```
        .item {
            padding: 10px;
            margin: 5px 0;
            background-color: white;
            border-radius: 4px;
        }
        .source {
            color: #666;
            font-size: 0.9em;
        }
    </style>
</head>
<body>
    <h1>Flask Kubernetes Stack</h1>

    <div>
        <button onclick="initDB()">Initialiser la base de données</button>
        <button onclick="loadItems()">Charger les items</button>
    </div>

    <div>
        <input type="text" id="itemName" placeholder="Nom de l'item">
        <button onclick="createItem()">Ajouter</button>
    </div>

    <div id="items"></div>

    <script src="app.js"></script>
</body>
</html>
```

Fichier `frontend/html/app.js` :

```
const API_URL = 'http://localhost:30500';

async function initDB() {
    try {
        const response = await fetch(`${API_URL}/init-db`, { method: 'POST' });
        const data = await response.json();
        alert(data.message);
    } catch (error) {
        alert('Erreur: ' + error.message);
    }
}

async function loadItems() {
    try {
        const response = await fetch(`${API_URL}/api/items`);
        const data = await response.json();

        const itemsDiv = document.getElementById('items');
        itemsDiv.innerHTML = `
            <p class="source">Source: ${data.source}</p>
```

```
            ${data.items.map(item => `
                <div class="item">${item.id} - ${item.name}</div>
            `).join('')}
        `;
    } catch (error) {
        alert('Erreur: ' + error.message);
    }
}

async function createItem() {
    const name = document.getElementById('itemName').value;
    if (!name) {
        alert('Veuillez entrer un nom');
        return;
    }

    try {
        const response = await fetch(`${API_URL}/api/items`, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ name })
        });
        const data = await response.json();
        document.getElementById('itemName').value = '';
        alert('Item créé: ' + data.name);
        loadItems();
    } catch (error) {
        alert('Erreur: ' + error.message);
    }
}
```

Fichier `frontend/nginx.conf` :

```
events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    server {
        listen 80;
        server_name localhost;

        root /usr/share/nginx/html;
        index index.html;

        location / {
            try_files $uri $uri/ /index.html;
        }
    }
```

```
}
```

Fichier `frontend/Dockerfile` :

```
FROM nginx:alpine

COPY nginx.conf /etc/nginx/nginx.conf
COPY html /usr/share/nginx/html

EXPOSE 80
```

Étape 4 : Docker Compose pour tester en local

Fichier `docker-compose.yml` à la racine :

```
version: '3.8'

services:
  postgres:
    image: postgres:14-alpine
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: mysecretpassword
    volumes:
      - postgres_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  redis:
    image: redis:alpine
    ports:
      - "6379:6379"

  backend:
    build: ./backend
    environment:
      DATABASE_HOST: postgres
      DATABASE_NAME: mydb
      DATABASE_USER: postgres
      DATABASE_PASSWORD: mysecretpassword
      REDIS_HOST: redis
    ports:
      - "5000:5000"
    depends_on:
      - postgres
      - redis

  frontend:
    build: ./frontend
    ports:
      - "8080:80"
```

```
    depends_on:
      - backend

volumes:
  postgres_data:
```

Tester en local :

```
docker-compose up --build
```

Aller sur `http://localhost:8080`, cliquer sur "Initialiser la base de données", puis ajouter des items.

Stopper :

```
docker-compose down
```

Étape 5 : Construire et publier les images

Option 1 : Utiliser Minikube Docker daemon

```
# Configurer le shell pour utiliser le daemon Docker de Minikube
eval $(minikube docker-env)

# Build les images
docker build -t flask-backend:1.0 ./backend
docker build -t flask-frontend:1.0 ./frontend

# Revenir au daemon Docker local (optionnel)
eval $(minikube docker-env -u)
```

Option 2 : Docker Hub (pour partager avec l'équipe)

```
# Remplacer 'votre-username' par votre username Docker Hub
docker build -t votre-username/flask-backend:1.0 ./backend
docker build -t votre-username/flask-frontend:1.0 ./frontend

docker push votre-username/flask-backend:1.0
docker push votre-username/flask-frontend:1.0
```

Pour ce projet, on va utiliser l'option 1 (Minikube Docker daemon).

Étape 6 : Manifests Kubernetes

Fichier `k8s/postgres-pvc.yaml` :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

Fichier `k8s/postgres-secret.yaml` :

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
stringData:
  password: mysecretpassword
```

Fichier `k8s/postgres-deployment.yaml` :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:14-alpine
        env:
        - name: POSTGRES_DB
          value: mydb
        - name: POSTGRES_USER
          value: postgres
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: password
```

```
        - name: PGDATA
          value: /var/lib/postgresql/data/pgdata
        ports:
        - containerPort: 5432
        volumeMounts:
        - name: postgres-storage
          mountPath: /var/lib/postgresql/data
      volumes:
      - name: postgres-storage
        persistentVolumeClaim:
          claimName: postgres-pvc
```

Fichier `k8s/postgres-service.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
  - protocol: TCP
    port: 5432
    targetPort: 5432
  type: ClusterIP
```

Fichier `k8s/redis-deployment.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:alpine
        ports:
        - containerPort: 6379
```

Fichier `k8s/redis-service.yaml` :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
  - protocol: TCP
    port: 6379
    targetPort: 6379
  type: ClusterIP
```

Fichier `k8s/backend-configmap.yaml` :

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: backend-config
data:
  DATABASE_HOST: postgres
  DATABASE_NAME: mydb
  DATABASE_USER: postgres
  REDIS_HOST: redis
```

Fichier `k8s/backend-deployment.yaml` :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: flask-backend:1.0
        imagePullPolicy: Never
        envFrom:
        - configMapRef:
            name: backend-config
        env:
        - name: DATABASE_PASSWORD
          valueFrom:
```

```
            secretKeyRef:
              name: postgres-secret
              key: password
        ports:
        - containerPort: 5000
```

Note : `imagePullPolicy: Never` force Kubernetes à utiliser l'image locale (construite avec Minikube Docker daemon).

Fichier `k8s/backend-service.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
  - protocol: TCP
    port: 5000
    targetPort: 5000
    nodePort: 30500
  type: NodePort
```

Fichier `k8s/frontend-deployment.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: flask-frontend:1.0
        imagePullPolicy: Never
        ports:
        - containerPort: 80
```

Fichier `k8s/frontend-service.yaml` :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
    nodePort: 30800
  type: NodePort
```

Étape 7 : Déployer sur Minikube

Démarrer Minikube :

```
minikube start
```

Build les images dans Minikube :

```
eval $(minikube docker-env)
docker build -t flask-backend:1.0 ./backend
docker build -t flask-frontend:1.0 ./frontend
```

Appliquer tous les manifests :

```
kubectl apply -f k8s/
```

Vérifier les déploiements :

```
kubectl get all
```

Sortie attendue :

```
NAME                            READY   STATUS    RESTARTS   AGE
pod/backend-xxx                 1/1     Running   0          30s
pod/backend-yyy                 1/1     Running   0          30s
pod/frontend-xxx                1/1     Running   0          30s
pod/frontend-yyy                1/1     Running   0          30s
pod/postgres-xxx                1/1     Running   0          30s
pod/redis-xxx                   1/1     Running   0          30s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/backend      NodePort    10.96.100.10     <none>        5000:30500/TCP   30s
service/frontend     NodePort    10.96.100.20     <none>        80:30800/TCP     30s
service/postgres     ClusterIP   10.96.100.30     <none>        5432/TCP         30s
service/redis        ClusterIP   10.96.100.40     <none>        6379/TCP         30s
```

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend      2/2     2            2           30s
deployment.apps/frontend     2/2     2            2           30s
deployment.apps/postgres     1/1     1            1           30s
deployment.apps/redis        1/1     1            1           30s
```

Obtenir l'URL Minikube :

```
minikube service frontend --url
minikube service backend --url
```

Exemple de sortie :

```
http://192.168.49.2:30800  (frontend)
http://192.168.49.2:30500  (backend)
```

Modifier `frontend/html/app.js` avec l'URL du backend :

Remplacer `const API_URL = 'http://localhost:30500';` par l'URL retournée par `minikube service backend --url`.

Rebuild le frontend :

```
eval $(minikube docker-env)
docker build -t flask-frontend:1.0 ./frontend
kubectl rollout restart deployment/frontend
```

Ouvrir l'application :

```
minikube service frontend
```

Cliquer sur "Initialiser la base de données", puis ajouter des items.

Étape 8 : Tester le scaling

Scaler le backend :

```
kubectl scale deployment backend --replicas=5
kubectl get pods -l app=backend -w
```

Vérifier le load balancing :

```
# Faire plusieurs requêtes et voir les logs des différents pods
kubectl logs -l app=backend --tail=10
```

Réduire :

```
kubectl scale deployment backend --replicas=2
```

Étape 9 : Tester la résilience

Supprimer un pod backend :

```
kubectl delete pod -l app=backend --force --grace-period=0
kubectl get pods -w
```

Le Deployment recrée immédiatement un nouveau pod.

Supprimer le pod PostgreSQL :

```
kubectl delete pod -l app=postgres --force --grace-period=0
kubectl get pods -w
```

Vérifier que les données persistent :

```
# Une fois le pod recréé
kubectl exec -it deployment/postgres -- psql -U postgres -d mydb -c "SELECT * FROM items;"
```

Les données sont toujours là grâce au PersistentVolume.

Étape 10 : Bonus - Ingress

Activer Ingress dans Minikube :

```
minikube addons enable ingress
```

Fichier `k8s/ingress.yaml` :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: flask-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: flask.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
      - path: /api
        pathType: Prefix
        backend:
          service:
```

```
            name: backend
            port:
              number: 5000
```

Appliquer :

```
kubectl apply -f k8s/ingress.yaml
kubectl get ingress
```

Ajouter `flask.local` à `/etc/hosts` :

```
minikube ip
```

Copier l'IP (ex: `192.168.49.2`).

Éditer `/etc/hosts` (Linux/Mac) ou `C:\Windows\System32\drivers\etc\hosts` (Windows) :

```
192.168.49.2  flask.local
```

Accéder via Ingress :

```
http://flask.local
http://flask.local/api/items
```

Modifier `frontend/html/app.js` :

```
const API_URL = 'http://flask.local/api';
```

Rebuild et redéployer le frontend.

Étape 11 : Cleanup

```
kubectl delete -f k8s/
kubectl delete pvc postgres-pvc
```

Questions checkpoint

- Pourquoi utilise-t-on `imagePullPolicy: Never` ?
- Quelle est la différence entre un Service ClusterIP et NodePort ?
- Comment vérifier que le load balancing fonctionne entre les replicas ?
- Pourquoi les données PostgreSQL persistent-elles après suppression du pod ?
- Qu'est-ce qu'un Ingress et pourquoi l'utiliser ?

---

10 - Introduction à Helm

L'essentiel

Problème :

Gérer des dizaines de fichiers YAML pour chaque application devient vite ingérable :

- Duplication de code (même structure pour chaque service)
- Pas de versioning des déploiements
- Difficile de partager des configurations
- Impossible de rollback facilement

Solution : Helm

Helm est le **package manager de Kubernetes**.

Analogie :

- APT pour Debian/Ubuntu
- YUM pour Red Hat/CentOS
- NPM pour Node.js
- PIP pour Python
- **Helm pour Kubernetes**

Concepts :

- **Chart** : Package contenant tous les manifests K8s d'une application
- **Release** : Instance d'un chart déployée sur K8s
- **Repository** : Collection de charts (comme Docker Hub pour images)

Exemple :

Au lieu de gérer 10 fichiers YAML pour PostgreSQL, on installe un chart :

```
helm install my-postgres bitnami/postgresql
```

Et c'est tout.

Pourquoi Helm ?

1. DRY (Don't Repeat Yourself)

Au lieu de dupliquer les manifests pour dev, staging, prod :

```
# values-dev.yaml
replicaCount: 1

# values-prod.yaml
replicaCount: 5
```

Un seul chart, plusieurs configurations.

2. Versioning

```
helm list
```

Sortie :

```
NAME        NAMESPACE  REVISION  STATUS    CHART           APP VERSION
my-app      default    3         deployed  my-app-1.2.0    2.5.1
```

On sait quelle version est déployée, on peut rollback à la version précédente.

3. Partage

Des milliers de charts prêts à l'emploi :

- PostgreSQL, MySQL, MongoDB
- Redis, RabbitMQ, Kafka
- WordPress, GitLab, Jenkins
- Prometheus, Grafana

4. Templates

Les charts utilisent Go templates pour générer les manifests :

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.serviceName }}
spec:
  replicas: {{ .Values.replicaCount }}
```

Installer Helm

Linux :

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

macOS :

```
brew install helm
```

Windows :

```
choco install kubernetes-helm
```

Vérifier l'installation :

```
helm version
```

Sortie :

```
version.BuildInfo{Version:"v3.13.0", GitCommit:"...", GitTreeState:"clean"}
```

Rechercher des charts

Ajouter un repository :

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

Rechercher des charts :

```
helm search repo postgres
```

Sortie :

```
NAME                          CHART VERSION  APP VERSION  DESCRIPTION
bitnami/postgresql            12.10.0        15.4.0       PostgreSQL is an advanced object-
relational...
bitnami/postgresql-ha         11.9.0         15.4.0       PostgreSQL with HA
architecture...
```

Voir les détails d'un chart :

```
helm show chart bitnami/postgresql
helm show values bitnami/postgresql
```

Installer un chart

Installer PostgreSQL :

```
helm install my-postgres bitnami/postgresql
```

Helm va :

1. Télécharger le chart
2. Générer les manifests K8s
3. Les appliquer sur le cluster
4. Créer une release

Vérifier :

```
helm list
kubectl get all -l app.kubernetes.io/instance=my-postgres
```

Obtenir le mot de passe PostgreSQL :

Helm affiche des notes après installation :

```
export POSTGRES_PASSWORD=$(kubectl get secret --namespace default my-postgres-postgresql -o jsonpath="{.data.postgres-password}" | base64 -d)
```

Se connecter à PostgreSQL :

```
kubectl run my-postgres-postgresql-client --rm --tty -i --restart='Never' \
  --namespace default \
  --image docker.io/bitnami/postgresql:15.4.0-debian-11-r0 \
  --env="PGPASSWORD=$POSTGRES_PASSWORD" \
  --command -- psql --host my-postgres-postgresql -U postgres -d postgres -p 5432
```

Personnaliser un chart

Fichier `values.yaml` :

```
auth:
  username: myuser
  password: mypassword
  database: mydb

primary:
  persistence:
    enabled: true
    size: 2Gi

replicaCount: 2
```

Installer avec ces valeurs :

```
helm install my-postgres bitnami/postgresql -f values.yaml
```

Ou en ligne de commande :

```
helm install my-postgres bitnami/postgresql \
  --set auth.username=myuser \
  --set auth.password=mypassword \
  --set auth.database=mydb \
  --set primary.persistence.size=2Gi
```

Mettre à jour une release

Modifier `values.yaml` :

```
replicaCount: 5
```

Appliquer les changements :

```
helm upgrade my-postgres bitnami/postgresql -f values.yaml
```

Vérifier :

```
helm list
```

Sortie :

```
NAME         NAMESPACE  REVISION  STATUS    CHART              APP VERSION
my-postgres  default    2         deployed  postgresql-12.10.0 15.4.0
```

`REVISION: 2` → Deuxième version de la release.

Rollback

Problème après un upgrade ?

Rollback à la version précédente :

```
helm rollback my-postgres
```

Rollback à une révision spécifique :

```
helm rollback my-postgres 1
```

Historique des releases :

```
helm history my-postgres
```

Sortie :

```
REVISION  UPDATED                   STATUS      CHART              DESCRIPTION
1         Mon Oct  2 10:00:00 2025  superseded  postgresql-12.10.0 Install complete
2         Mon Oct  2 11:00:00 2025  superseded  postgresql-12.10.0 Upgrade complete
3         Mon Oct  2 11:30:00 2025  deployed    postgresql-12.10.0 Rollback to 1
```

Désinstaller une release

```
helm uninstall my-postgres
```

Helm supprime tous les objets K8s créés par le chart.

Vérifier :

```
kubectl get all -l app.kubernetes.io/instance=my-postgres
```

Résultat : rien.

Créer son propre chart

Générer la structure d'un chart :

```
helm create my-app
```

Structure créée :

```
my-app/
├── Chart.yaml          # Métadonnées du chart
├── values.yaml         # Valeurs par défaut
├── templates/          # Templates de manifests K8s
│   ├── deployment.yaml
│   ├── service.yaml
│   ├── ingress.yaml
│   ├── _helpers.tpl    # Fonctions helper
│   └── NOTES.txt       # Notes affichées après install
└── charts/             # Dépendances
```

Fichier `Chart.yaml` :

```yaml
apiVersion: v2
name: my-app
description: A Helm chart for my Flask app
type: application
version: 1.0.0
appVersion: "1.0"
```

Fichier `values.yaml` :

```yaml
replicaCount: 2

image:
  repository: flask-backend
  tag: "1.0"
  pullPolicy: Never

service:
  type: NodePort
  port: 5000
  nodePort: 30500

database:
  host: postgres
  name: mydb
  user: postgres
```

Fichier `templates/deployment.yaml` :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "my-app.fullname" . }}
  labels:
    {{- include "my-app.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "my-app.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "my-app.selectorLabels" . | nindent 8 }}
    spec:
      containers:
      - name: {{ .Chart.Name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        ports:
        - name: http
          containerPort: {{ .Values.service.port }}
        env:
        - name: DATABASE_HOST
          value: {{ .Values.database.host }}
        - name: DATABASE_NAME
          value: {{ .Values.database.name }}
        - name: DATABASE_USER
          value: {{ .Values.database.user }}
```

Fichier `templates/service.yaml` :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ include "my-app.fullname" . }}
  labels:
    {{- include "my-app.labels" . | nindent 4 }}
spec:
  type: {{ .Values.service.type }}
  ports:
  - port: {{ .Values.service.port }}
    targetPort: http
    protocol: TCP
    name: http
    {{- if and (eq .Values.service.type "NodePort") .Values.service.nodePort }}
    nodePort: {{ .Values.service.nodePort }}
    {{- end }}
  selector:
    {{- include "my-app.selectorLabels" . | nindent 4 }}
```

Installer le chart :

```
helm install my-app ./my-app
```

Vérifier :

```
kubectl get all -l app.kubernetes.io/instance=my-app
```

Débugger les templates :

```
helm template my-app ./my-app
```

Affiche les manifests générés sans les appliquer.

Valider le chart :

```
helm lint ./my-app
```

Exercice guidé : Packager la stack Flask en Helm

Objectif :

Créer un Helm chart pour la stack Flask du projet précédent.

Étape 1 : Créer le chart

```
cd flask-k8s-stack
helm create flask-chart
```

Étape 2 : Supprimer les templates par défaut

```
rm -rf flask-chart/templates/*
```

Étape 3 : Copier les manifests K8s

```
cp k8s/*.yaml flask-chart/templates/
```

Étape 4 : Modifier `values.yaml`

```
backend:
  replicaCount: 2
  image:
    repository: flask-backend
    tag: "1.0"
    pullPolicy: Never

frontend:
  replicaCount: 2
  image:
    repository: flask-frontend
```

```
    tag: "1.0"
    pullPolicy: Never

postgres:
  enabled: true
  persistence:
    size: 1Gi
  auth:
    password: mysecretpassword
    database: mydb

redis:
  enabled: true
```

Étape 5 : Templater `backend-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: {{ .Values.backend.replicaCount }}
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: backend
        image: "{{ .Values.backend.image.repository }}:{{ .Values.backend.image.tag }}"
        imagePullPolicy: {{ .Values.backend.image.pullPolicy }}
        envFrom:
        - configMapRef:
            name: backend-config
        env:
        - name: DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: postgres-secret
              key: password
        ports:
        - containerPort: 5000
```

Étape 6 : Templater `postgres-secret.yaml`

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
stringData:
  password: {{ .Values.postgres.auth.password }}
```

Étape 7 : Installer le chart

```
# S'assurer que les images sont dans Minikube
eval $(minikube docker-env)
docker build -t flask-backend:1.0 ./backend
docker build -t flask-frontend:1.0 ./frontend

# Installer le chart
helm install flask-app ./flask-chart
```

Étape 8 : Vérifier

```
helm list
kubectl get all
```

Étape 9 : Mettre à jour

Modifier `values.yaml` :

```
backend:
  replicaCount: 5
```

```
helm upgrade flask-app ./flask-chart
```

Étape 10 : Rollback

```
helm rollback flask-app
```

Étape 11 : Cleanup

```
helm uninstall flask-app
```

Questions checkpoint

- Qu'est-ce qu'un chart Helm ?
- Quelle est la différence entre un chart et une release ?
- Comment rollback à une version précédente ?
- Pourquoi utiliser Helm plutôt que kubectl apply directement ?
- Comment débugger les templates Helm avant de les appliquer ?

Checklist de validation

Avant de terminer ce cours, assurez-vous de pouvoir :

Installation et setup :

- [ ] Installer Minikube sur votre machine
- [ ] Installer kubectl et vérifier la connexion au cluster
- [ ] Démarrer et stopper Minikube
- [ ] Utiliser `kubectl get`, `describe`, `logs`, `exec`

Pods :

- [ ] Créer un pod depuis un fichier YAML
- [ ] Créer un pod en ligne de commande avec `kubectl run`
- [ ] Inspecter un pod avec `kubectl describe`
- [ ] Voir les logs d'un pod
- [ ] Exec dans un pod

Deployments :

- [ ] Créer un Deployment
- [ ] Scaler un Deployment manuellement
- [ ] Faire un rolling update
- [ ] Rollback à une version précédente
- [ ] Vérifier l'historique des révisions

Services :

- [ ] Créer un Service ClusterIP
- [ ] Créer un Service NodePort
- [ ] Tester la résolution DNS interne
- [ ] Accéder à un service depuis l'extérieur du cluster

ConfigMaps et Secrets :

- [ ] Créer un ConfigMap depuis un fichier
- [ ] Créer un Secret
- [ ] Injecter un ConfigMap en variable d'environnement
- [ ] Monter un Secret en volume

Persistent Volumes :

- [ ] Créer un PersistentVolumeClaim
- [ ] Utiliser un PVC dans un Deployment
- [ ] Vérifier que les données persistent après suppression du pod
- [ ] Comprendre la différence entre PV et PVC

Projet Flask :

- [ ] Dockeriser une application Flask
- [ ] Écrire les manifests K8s pour une stack complète
- [ ] Déployer la stack sur Minikube
- [ ] Tester le scaling et la résilience
- [ ] Configurer un Ingress

Helm :

- [ ] Installer Helm
- [ ] Rechercher et installer un chart depuis un repository
- [ ] Personnaliser un chart avec `values.yaml`
- [ ] Créer un chart personnalisé
- [ ] Faire un upgrade et un rollback avec Helm

P'tit quizz

1. Quelle est la différence entre un pod et un conteneur ?

Un pod est la plus petite unité déployable dans Kubernetes. Il peut contenir un ou plusieurs conteneurs qui partagent le même réseau et stockage. La plupart du temps, 1 pod = 1 conteneur.

2. Pourquoi utiliser un Deployment plutôt que créer des pods manuellement ?

Un Deployment gère automatiquement la création, la mise à jour et le scaling des pods. Il assure le self-healing (recréation automatique des pods en cas de crash), les rolling updates, et les rollbacks.

3. Quelle est la différence entre un Service ClusterIP et NodePort ?

- ClusterIP : Service accessible uniquement depuis l'intérieur du cluster
- NodePort : Service exposé sur un port de chaque node, accessible depuis l'extérieur

4. Les Secrets sont-ils vraiment sécurisés par défaut ?

Non. Les Secrets sont encodés en base64, pas chiffrés. N'importe qui avec accès à l'API K8s peut les décoder. Pour une vraie sécurité, il faut activer le chiffrement at-rest ou utiliser des solutions externes comme HashiCorp Vault.

5. Pourquoi utiliser un PVC plutôt que monter directement un volume ?

Le PVC est une abstraction qui découple la demande de stockage de l'implémentation physique. Le développeur demande "j'ai besoin de 10Gi", et Kubernetes trouve automatiquement un volume compatible. Cela facilite la portabilité entre différents environnements (local, cloud, etc.).

6. Qu'est-ce que la boucle de réconciliation dans Kubernetes ?

C'est le mécanisme par lequel Kubernetes compare en permanence l'état désiré (manifests YAML) avec l'état réel du cluster, et applique les changements nécessaires pour les faire correspondre. Par exemple, si un pod meurt, Kubernetes le recrée automatiquement.

7. Quelle est la différence entre un chart Helm et une release ?

- Chart : Package contenant les templates et manifests K8s (comme un paquet .deb ou .rpm)
- Release : Instance d'un chart déployée sur le cluster (comme une application installée)

8. Pourquoi utiliser Helm plutôt que kubectl apply directement ?

Helm apporte le versioning, les rollbacks faciles, la réutilisation de configurations, et la gestion de dépendances. Au lieu de gérer des dizaines de fichiers YAML, on gère un seul chart avec des valeurs personnalisables.

9. Que se passe-t-il si on supprime un Deployment ?

Tous les pods gérés par ce Deployment sont également supprimés. Les Services associés restent, mais n'ont plus de pods backend.

10. Comment exposer une application sur un nom de domaine (ex: app.example.com) ?

Utiliser un Ingress qui route le trafic HTTP/HTTPS vers les Services internes en fonction du nom de domaine et du chemin.

Pour aller plus loin

Concepts avancés :

- **StatefulSets** : Pour les applications stateful (bases de données, Kafka, etc.)
- **DaemonSets** : Pour exécuter un pod sur chaque node (monitoring agents, log collectors)
- **Jobs et CronJobs** : Pour les tâches batch et planifiées
- **Horizontal Pod Autoscaler (HPA)** : Scaling automatique basé sur CPU/mémoire
- **Network Policies** : Firewall entre pods
- **RBAC (Role-Based Access Control)** : Gestion fine des permissions
- **Custom Resource Definitions (CRD)** : Étendre l'API Kubernetes
- **Operators** : Automatiser la gestion d'applications complexes

Outils et écosystème :

- **Kustomize** : Gestion de configurations sans templates
- **ArgoCD / Flux** : GitOps pour déploiements automatiques
- **Prometheus + Grafana** : Monitoring et alerting
- **Istio / Linkerd** : Service mesh pour trafic avancé
- **Lens** : IDE graphique pour Kubernetes
- **K9s** : CLI interactif pour naviguer dans un cluster
- **kubectl plugins** : Étendre kubectl (kubectl-tree, kubectl-neat, etc.)

Production :

- Managed Kubernetes : EKS (AWS), GKE (Google Cloud), AKS (Azure)
- Cluster multi-nodes avec haute disponibilité
- Backup et disaster recovery (Velero)

- Secrets management avec Vault ou Sealed Secrets
- Ingress Controllers : Nginx, Traefik, HAProxy
- Cert-Manager : Gestion automatique des certificats SSL

Apprendre en pratiquant :

- Kubernetes by Example
- Play with Kubernetes
- Katacoda Kubernetes
- Documentation officielle : kubernetes.io
- Kubernetes the Hard Way : Installer K8s from scratch

Commandes essentielles

kubectl basics

```
# Infos cluster
kubectl cluster-info
kubectl get nodes
kubectl version

# Pods
kubectl get pods
kubectl get pods -o wide
kubectl describe pod <nom>
kubectl logs <pod>
kubectl logs <pod> -f
kubectl exec -it <pod> -- /bin/sh
kubectl delete pod <nom>

# Deployments
kubectl get deployments
kubectl describe deployment <nom>
kubectl scale deployment <nom> --replicas=5
kubectl rollout status deployment/<nom>
kubectl rollout history deployment/<nom>
kubectl rollout undo deployment/<nom>
kubectl delete deployment <nom>

# Services
kubectl get services
kubectl get svc
kubectl describe service <nom>
kubectl delete service <nom>

# ConfigMaps et Secrets
kubectl get configmaps
kubectl get secrets
kubectl describe configmap <nom>
```

```
kubectl describe secret <nom>
kubectl create configmap <nom> --from-file=<fichier>
kubectl create secret generic <nom> --from-literal=key=value
kubectl delete configmap <nom>
kubectl delete secret <nom>

# PersistentVolumes
kubectl get pv
kubectl get pvc
kubectl describe pv <nom>
kubectl describe pvc <nom>
kubectl delete pvc <nom>

# Ingress
kubectl get ingress
kubectl describe ingress <nom>

# Namespaces
kubectl get namespaces
kubectl get pods -n <namespace>
kubectl create namespace <nom>

# Appliquer des manifests
kubectl apply -f fichier.yaml
kubectl apply -f repertoire/
kubectl delete -f fichier.yaml

# Debug
kubectl get events
kubectl get all
kubectl top nodes
kubectl top pods
```

Minikube

```
# Démarrer/Stopper
minikube start
minikube stop
minikube delete

# Infos
minikube status
minikube ip
minikube ssh

# Services
minikube service <nom>
minikube service <nom> --url

# Addons
minikube addons list
minikube addons enable ingress
```

```
minikube addons enable metrics-server

# Docker daemon
eval $(minikube docker-env)
eval $(minikube docker-env -u)

# Dashboard
minikube dashboard
```

Helm

```
# Repos
helm repo add <nom> <url>
helm repo update
helm search repo <mot-clé>

# Charts
helm show chart <chart>
helm show values <chart>

# Releases
helm list
helm install <nom-release> <chart>
helm install <nom-release> <chart> -f values.yaml
helm install <nom-release> <chart> --set key=value
helm upgrade <nom-release> <chart>
helm rollback <nom-release>
helm rollback <nom-release> <revision>
helm uninstall <nom-release>
helm history <nom-release>

# Créer un chart
helm create <nom>
helm lint <chart>
helm template <nom-release> <chart>
helm package <chart>

# Debug
helm get values <nom-release>
helm get manifest <nom-release>
```

Bravo, vous avez terminé le cours Kubernetes !

Vous savez maintenant orchestrer des applications containerisées à grande échelle avec Kubernetes et Helm. Vous êtes prêts pour déployer en production.