UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes

Sommaire

Sommaire

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes et l’orchestration de containers

Kubernetes et l'orchestration de containers

1. **Pourquoi un orchestrateur?**

- Dans le monde moderne de la technologie de l'information, les applications sont souvent déployées à grande échelle, fonctionnant sur des centaines voire des milliers de conteneurs. Les défis à cette échelle comprennent:
    1. **Déploiement**: Comment déployer efficacement des milliers de conteneurs?
    2. **Réparation**: Comment réparer les conteneurs qui tombent en panne?
    3. **Mise à l'échelle**: Comment adapter les ressources pour des conteneurs en fonction de la demande?
    4. **Découverte et équilibrage de charge**: Comment les conteneurs peuvent-ils découvrir et communiquer entre eux?
- Un orchestrateur, comme Kubernetes, aide à répondre à ces questions en automatisant le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.

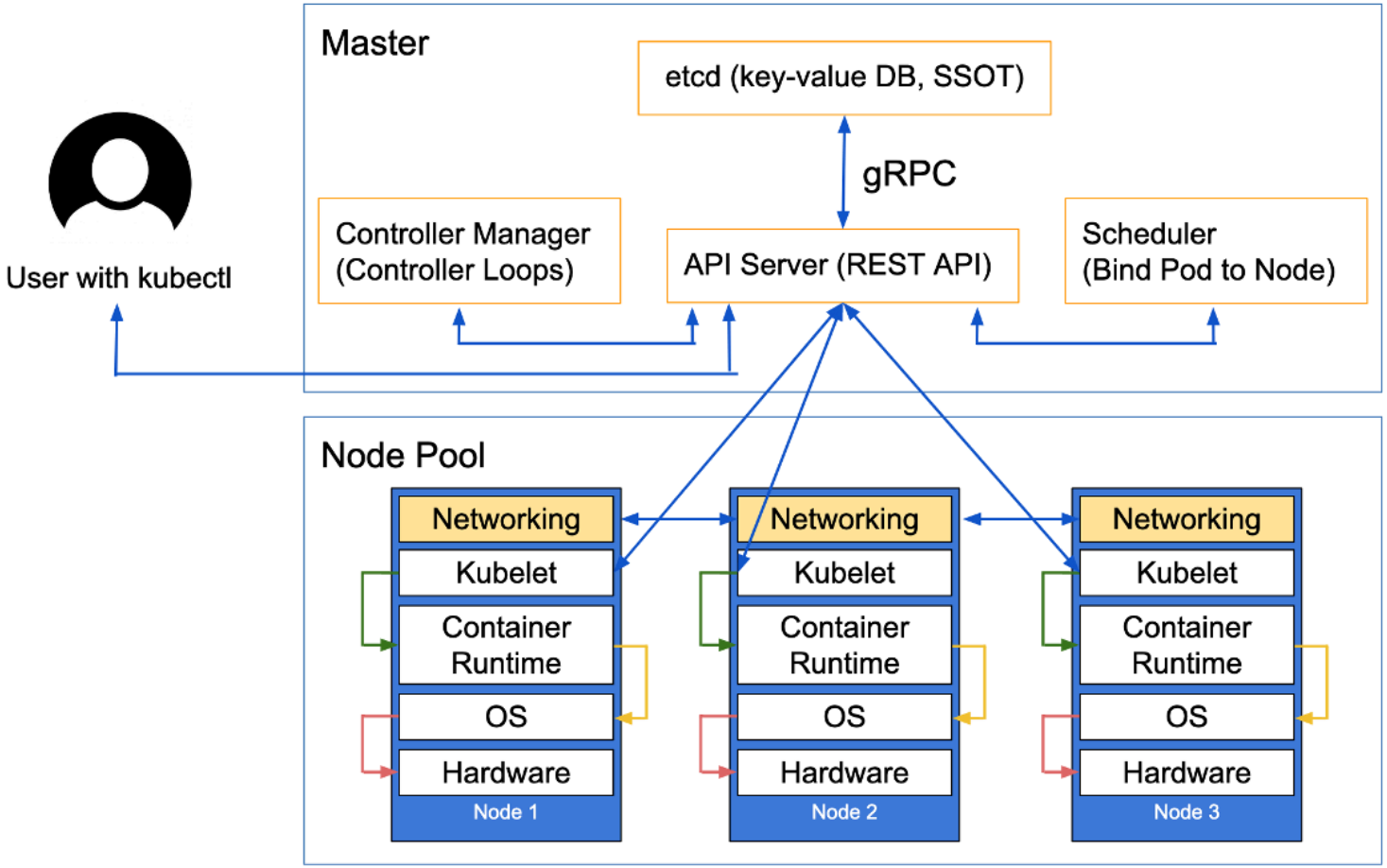
Kubernetes et l'Orchestration de Containers

2. **Avantages de Kubernetes**
    1. **Automatisation**: Kubernetes peut automatiquement déployer, échelonner et équilibrer les charges entre les conteneurs.
    2. **Santé et Auto-réparation**: Il surveille la santé des conteneurs et remplace ceux qui échouent, et peut également automatiser les mises à jour.
    3. **Gestion des ressources**: Il assure que chaque conteneur reçoit les ressources (CPU, mémoire) dont il a besoin.
    4. **Découvrabilité**: Avec son système de service intégré, Kubernetes facilite la découverte et la communication entre les conteneurs.
    5. **Stockage**: Il peut monter et ajouter des systèmes de stockage pour conserver les données persistentes.
    6. **Extensibilité**: Grâce à sa modularité et sa flexibilité, Kubernetes peut s'étendre pour répondre aux besoins les plus complexes.
    7. **Communauté active**: Étant open source, il bénéficie d'une grande et active communauté qui continue à contribuer et à améliorer le système.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Architecture de Kubernetes

Architecture de Kubernetes



Architecture de Kubernetes

1. **Principes de fonctionnement**

- Kubernetes est basé sur une architecture de type maître-esclave (ou master-worker). Le maître (ou master) prend des décisions concernant le cluster, telles que la planification, et répond aux demandes d'API, tandis que les esclaves (ou workers) exécutent les conteneurs.

    1. **État désiré vs état actuel**: L'une des idées fondamentales de Kubernetes est la notion d'état désiré. Vous définissez ce que vous souhaitez voir s'exécuter (par exemple, je veux 3 instances de mon application), et Kubernetes s'efforce de s'assurer que la réalité correspond à cet état.
    2. **Autoguérison**: Si un conteneur tombe en panne, Kubernetes le redémarre pour maintenir l'état désiré. De même, si une machine entière tombe en panne, les conteneurs qui s'y exécutaient sont redistribués.

Architecture de Kubernetes

2. **Composants de Kubernetes**
    1. **API Server (serveur API)**: Point d'entrée pour les commandes. Tout dans Kubernetes est traité comme une API.
    2. **etcd**: Base de données clé-valeur utilisée pour tout le stockage de configuration et d'état.
    3. **kubelet**: Agent qui s'exécute sur chaque noeud et s'assure que les conteneurs sont en cours d'exécution dans un pod.
    4. **kube-proxy**: Maintient les règles réseau sur les noeuds pour permettre la communication vers les conteneurs.
    5. **Scheduler (ordonnanceur)**: Décide quel noeud doit exécuter un conteneur.

Architecture de Kubernetes

3. **Masters vs Workers**
    1. **Master (Maître)**:
        - Gère le cluster.
        - Prend des décisions globales (par exemple, la planification).
        - Détecte les événements du cluster (par exemple, un conteneur qui a échoué).
    - *Composants typiques d'un noeud master: API Server, etcd, Scheduler, et autres composants de contrôle.*
    2. **Worker (Esclave)**:
        - Exécute les conteneurs.
        - Rapporte à master.
        - Chaque worker est équipé de Docker (ou une autre solution conteneur), kubelet, et kube-proxy.

Architecture de Kubernetes

4. **Couche réseau**

- La couche réseau dans Kubernetes est cruciale car elle permet la communication entre les conteneurs et aussi entre le cluster et l'extérieur.

    1. **Pod Networking**: Chaque pod reçoit sa propre adresse IP. Les conteneurs au sein d'un pod partagent cette adresse IP et le port, ce qui signifie qu'ils peuvent se communiquer via `localhost`.
    2. **Service Networking**: Expose un ensemble de pods en tant que service. Les services permettent la communication entre les pods et l'extérieur du cluster.
    3. **Network Policies**: Permet de contrôler la communication entre les pods.
    4. **CNI (Container Network Interface)**: Il s'agit d'un ensemble de normes et de plugins qui permettent l'intégration de différentes solutions réseau avec Kubernetes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Concepts de base

Concepts de base

1. **Concepts de base de Kubernetes**

- Kubernetes introduit un certain nombre de concepts et d'abstractions pour aider les utilisateurs à déployer, gérer et échelonner leurs applications.

2. **Kubernetes API**

- L'API Kubernetes est la colonne vertébrale du système. Elle est utilisée pour créer, mettre à jour et surveiller les diverses ressources disponibles dans Kubernetes.
  - **Versioning**: Kubernetes prend en charge plusieurs versions d'API en même temps pour assurer une compatibilité ascendante.
  - **Ressources**: Les objets dans Kubernetes, tels que les pods, les services, etc., sont tous représentés comme des ressources API.
  - **Opérations CRUD**: L'API Kubernetes permet d'effectuer des opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur ces ressources.

Concepts de base

3. **Outil `kubectl`**

- `kubectl` est l'outil en ligne de commande pour interagir avec le cluster Kubernetes. Il utilise l'API Kubernetes pour communiquer avec le cluster.
  - **Commandes de base**:
    - `kubectl get`: Affiche une ou plusieurs ressources.
    - `kubectl describe`: Montre les détails d'une ressource spécifique.
    - `kubectl create`: Crée une ressource.
    - `kubectl delete`: Supprime des ressources.
    - `kubectl apply`: Applique une configuration à une ressource.

Concepts de base - Ressources de base

1. **Pod**

Un pod est la plus petite unité déployable dans Kubernetes. Il peut contenir un ou plusieurs conteneurs.

- **Multi-container Pods**: Plusieurs conteneurs fonctionnant ensemble dans un seul pod partagent le même réseau et le même espace de stockage.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

Concepts de base - Ressources de base

2. **Deployment**

Gère le déploiement de pods. Il peut créer ou supprimer des pods pour maintenir l'état désiré.

- **Mise à jour et déploiement continu**: Avec les Deployments, vous pouvez mettre à jour vos pods sans interruption de service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```

Concepts de base - Ressources de base

3. **Label**
    - Les étiquettes (Labels) sont des paires clé-valeur associées aux ressources pour les organiser.
        - **Sélecteurs**: Utilisés pour filtrer les ressources basées sur leurs étiquettes.
4. **Namespace**

- Permet de diviser les ressources d'un cluster entre plusieurs utilisateurs ou projets.
    - **Isolation**: Chaque namespace fournit une portée pour les noms de ressources.

```
kubectl create namespace development
```

Concepts de base - Ressources de base

6. **Service**

- Expose un ensemble de pods comme un service réseau. Il fournit un IP stable et un DNS pour les pods.
  - **Types**: ClusterIP (interne), NodePort, LoadBalancer (externe), et ExternalName.

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Kubernetes au quotidien

Kubernetes au quotidien

Introduction à kubectl : Commandes de base et usage recommandé

Avant d'utiliser kubectl, vous devez configurer l'accès à votre cluster. Le fichier de configuration se trouve généralement dans `~/.kube/config` et contient les informations de connexion, les certificats et les contextes.

```
# Vérifier la configuration actuelle
kubectl config view

# Lister les contextes disponibles
kubectl config get-contexts

# Changer de contexte
kubectl config use-context nom-du-contexte
```

Kubernetes au quotidien

Commandes de base essentielles

Gestion des ressources

```
# Créer une ressource à partir d'un fichier YAML
kubectl apply -f fichier.yaml

# Créer une ressource à partir d'un répertoire
kubectl apply -f ./manifests/

# Supprimer une ressource
kubectl delete -f fichier.yaml
kubectl delete pod nom-du-pod
kubectl delete deployment nom-deployment
```

Kubernetes au quotidien

Consultation et inspection

```
# Lister les ressources
kubectl get pods
kubectl get services
kubectl get deployments
kubectl get nodes

# Obtenir des informations détaillées
kubectl describe pod nom-du-pod
kubectl describe service nom-service

# Consulter les logs
kubectl logs nom-du-pod
kubectl logs -f nom-du-pod  # suivi en temps réel
kubectl logs nom-du-pod -c nom-container  # logs d'un container spécifique
```

Kubernetes au quotidien

Débogage et dépannage

```
# Accéder à un pod en mode interactif
kubectl exec -it nom-du-pod -- /bin/bash

# Copier des fichiers vers/depuis un pod
kubectl cp fichier.txt nom-du-pod:/tmp/
kubectl cp nom-du-pod:/tmp/fichier.txt ./fichier-local.txt

# Port forwarding pour accéder aux services localement
kubectl port-forward service/nom-service 8080:80
kubectl port-forward pod/nom-pod 8080:8080
```

Kubernetes au quotidien

Commandes de surveillance

```
# Surveiller les ressources en temps réel
kubectl get pods -w
kubectl get events --sort-by=.metadata.creationTimestamp

# Vérifier l'état des nœuds
kubectl top nodes
kubectl top pods

# Obtenir des informations sur les ressources du cluster
kubectl cluster-info
kubectl api-resources
```

Kubernetes au quotidien

Bonnes pratiques et usage recommandé en interne

Conventions de nommage

Adoptez une convention de nommage cohérente pour vos ressources :

- Utilisez des noms descriptifs et standardisés
- Incluez l'environnement dans le nom (dev, staging, prod)
- Exemple : `app-frontend-prod`, `database-backend-staging`

Étiquetage et sélection

Les labels sont cruciaux pour organiser et sélectionner vos ressources :

```
# Ajouter des labels
kubectl label pods mon-pod env=production tier=frontend

# Sélectionner par labels
kubectl get pods -l env=production
kubectl get pods -l tier=frontend,env=production
```

Kubernetes au quotidien

Automatisation et scripts

Pour les tâches répétitives, créez des scripts réutilisables :

```
# Exemple de script de déploiement
#!/bin/bash
NAMESPACE=${1:-default}
kubectl apply -f ./k8s-manifests/ -n $NAMESPACE
kubectl rollout status deployment/mon-app -n $NAMESPACE
```

Sauvegarde et restauration

```
# Exporter la configuration d'une ressource
kubectl get deployment mon-app -o yaml > mon-app-backup.yaml

# Sauvegarder tous les manifestes d'un namespace
kubectl get all -n mon-namespace -o yaml > namespace-backup.yaml
```

Kubernetes au quotidien

Conseils pour un usage quotidien

1. **Utilisez des alias** pour les commandes fréquentes :

```
alias k='kubectl'
alias kgp='kubectl get pods'
alias kgs='kubectl get services'
```

2. **Activez l'autocomplétion** bash/zsh pour kubectl
3. **Utilisez des outils complémentaires** comme k9s pour une interface plus conviviale
4. **Documentez vos déploiements** avec des annotations dans vos manifests YAML
5. **Testez toujours** sur des environnements de développement avant la production

Approche déclarative vs impérative

Approche impérative

L'approche impérative consiste à donner des commandes spécifiques au système sur **comment** faire quelque chose, étape par étape.

```
# Exemples d'approche impérative
kubectl create deployment nginx --image=nginx:1.20
kubectl scale deployment nginx --replicas=3
kubectl expose deployment nginx --port=80 --type=LoadBalancer
kubectl set image deployment/nginx nginx=nginx:1.21
```

Avantages :

- Rapide pour les tests et le prototypage
- Idéal pour les tâches ponctuelles
- Apprentissage progressif des concepts

Inconvénients :

- Difficile à reproduire
- Pas de traçabilité des changements
- Gestion manuelle des configurations
- Risque d'incohérences entre environnements

Approche déclarative vs impérative

Approche déclarative

L'approche déclarative consiste à décrire l'état souhaité du système dans des fichiers de configuration. Kubernetes se charge ensuite d'atteindre et de maintenir cet état.

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
```

```
# Application de la configuration déclarative
kubectl apply -f deployment.yaml
```

Avantages :

- Reproductibilité parfaite
- Versioning avec Git
- Infrastructure as Code (IaC)
- Facilite les déploiements automatisés
- Maintenance cohérente entre environnements

Approche déclarative vs impérative

Structure et syntaxe YAML

Bases du format YAML

YAML (YAML Ain't Markup Language) est un format de sérialisation de données lisible par l'humain.

```yaml
# Commentaire en YAML
cle: valeur
nombre: 42
booleen: true
liste:
  - element1
  - element2
  - element3

objet_imbrique:
  sous_cle: sous_valeur
  autre_sous_cle: 123

# Liste d'objets
utilisateurs:
  - nom: Alice
    age: 30
  - nom: Bob
    age: 25
```

Utiliser k3s/k3d pour les environnements Kubernetes locaux

1. Pourquoi choisir k3s/k3d pour le développement local

Lorsqu'on développe des applications destinées à être déployées sur Kubernetes, il est essentiel de pouvoir simuler un environnement proche de la production, tout en conservant :

- une installation simple et rapide,
- un démarrage léger,
- une compatibilité totale avec les outils de l'écosystème Kubernetes.

Utiliser k3s/k3d pour les environnements Kubernetes locaux

1. k3s : Kubernetes simplifié

k3s est une distribution allégée de Kubernetes, conçue par Rancher/SUSE. Elle propose :

- un binaire unique,
- l'intégration directe de composants utiles (traefik, containerd, flannel, etc.),
- un remplacement d'`etcd` par SQLite par défaut,
- une réduction de l'utilisation mémoire et CPU.

Utiliser k3s/k3d pour les environnements Kubernetes locaux

2. k3d : exécuter k3s dans Docker

k3d est un wrapper qui permet de déployer k3s dans des conteneurs Docker. Il apporte plusieurs avantages :

- aucune installation système de Kubernetes,
- isolation totale dans Docker,
- création et suppression rapide de clusters,
- compatibilité directe avec `kubectl`,
- possibilité de créer des **clusters multi-nœuds**,
- possibilité de **connecter un registre Docker local**.

Déployer un cluster local avec k3d

1. Prérequis

- Docker installé (obligatoire)
- k3d installé (via `brew`, `curl`, ou binaire)

```
brew install k3d
# ou :
curl -s https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
```

Déployer un cluster local avec k3d

2. Création d'un cluster simple

```
k3d cluster create mon-cluster
```

Cela crée un cluster k3s avec un seul nœud serveur, accessible via `kubectl` :

```
kubectl get nodes
```

Déployer un cluster local avec k3d

3. Utiliser un registre local avec k3d

Dans un environnement de développement, on ne souhaite pas pousser chaque image vers un registre distant (Docker Hub, Harbor, etc.). k3d permet de créer un **registre local** automatiquement connecté au cluster.

- **Création d'un registre Docker local**

```
k3d registry create mon-registre.localhost --port 5000
```

Ce registre est disponible sur `localhost:5000` et utilisable comme tout registre privé.

Déployer un cluster local avec k3d

- **Création du cluster en l'associant au registre**

```
k3d cluster create mon-cluster \
  --registry-use k3d-mon-registre.localhost:5000 \
  --port "8080:80@loadbalancer"
```

- **Pousser une image vers le registre**

On peut ensuite construire une image et la pousser :

```
docker build -t localhost:5000/mon-app:dev .
docker push localhost:5000/mon-app:dev
```

Puis la déployer dans Kubernetes via un `Deployment` utilisant cette image.

Déployer un cluster local avec k3d

- **Vérification dans le cluster**

```
kubectl create deployment mon-app --image=localhost:5000/mon-app:dev
kubectl expose deployment mon-app --port=80 --type=LoadBalancer
```

Déployer un cluster local avec k3d

Critère	k3s/k3d	Minikube	Docker Desktop	Cluster managé (GKE, EKS, AKS)
Installation	Très simple	Moyenne	Très simple	Complexe
Lancement de cluster	Quelques secondes	Lent (VM ou Docker)	Instantané	Très lent (provisionnement Cloud)
Réalisme de l'environnement	Élevé (vrai cluster)	Élevé	Faible (émulation)	Très élevé
Support multi-nœuds	Oui	Oui	Non	Oui
Registre local intégré	Oui (avec `k3d`)	Possible mais manuel	Non	Non
CI/CD friendly	Oui	Peu adapté	Non	Pas local

Déployer un cluster local avec k3d

5. Bonnes pratiques pour le développement local en équipe

Travailler à plusieurs sur un environnement Kubernetes local nécessite de la rigueur et de la reproductibilité. Voici les recommandations essentielles.

1. Versionner l'environnement

- Tout ce qui configure l'environnement doit être dans Git :
  - fichiers YAML de déploiement,
  - fichiers `k3d` de création de cluster,
  - fichiers Helm/Kustomize si utilisés.

Déployer un cluster local avec k3d

2. Script d'automatisation

- Fournir un `Makefile` ou des scripts shell pour :
  - créer le cluster (`k3d cluster create`)
  - importer les images locales (`docker push localhost:5000/...`)
  - déployer les composants (`kubectl apply -f ...`)

Déployer un cluster local avec k3d

3. Déploiement reproductible

- Utiliser `kustomize`, Helm ou un fichier `values-dev.yaml` pour les environnements de développement.
- Ne jamais modifier les ressources directement avec `kubectl edit`.

4. Intégration dans CI/CD

- Utiliser k3d dans les runners locaux GitHub Actions ou GitLab pour tester le déploiement.
- Utiliser le registre local pour éviter les push vers un registre externe dans les tests.

Conteneur principal vs Init Container vs Sidecar

1. Conteneur principal

C'est le conteneur **responsable de l'application principale** dans un Pod.

- Il exécute le code métier ou l'élément central du service.
- Il est généralement celui auquel les utilisateurs accèdent via un `Service` Kubernetes.
- Exemple : un serveur web Nginx, une API Java, un backend Node.js, etc.

Conteneur principal vs Init Container vs Sidecar

2. Init Container

Un **Init Container** est un conteneur qui s'exécute **avant les conteneurs applicatifs (principaux)** dans le Pod.

Caractéristiques :

- Ils sont **séquentiels** : si plusieurs Init Containers sont déclarés, ils s'exécutent un par un.
- Le Pod **n'exécute pas les conteneurs principaux tant que les init containers ne sont pas terminés avec succès**.

Conteneur principal vs Init Container vs Sidecar

- Utilisation typique :
  - Initialisation d'une base de données.
  - Téléchargement d’un fichier de configuration.
  - Test de connectivité à une ressource externe.
  - Changement de permissions sur un volume monté.

Conteneur principal vs Init Container vs Sidecar

Exemple d'usage :

```yaml
initContainers:
  - name: init-config
    image: busybox
    command: ["sh", "-c", "wget http://config-server/config.yaml -O /app/config.yaml"]
    volumeMounts:
      - name: config-volume
        mountPath: /app
```

Conteneur principal vs Init Container vs Sidecar

3. Sidecar

Un **sidecar** est un conteneur **qui s'exécute en parallèle du conteneur principal**, dans le même Pod.

- Il **partage le réseau et les volumes** avec le conteneur principal.
- Il apporte des **capacités annexes ou transverses** à l'application :
  - collecte de logs,
  - synchronisation de fichiers,
  - proxy HTTP/HTTPS,
  - observabilité ou sécurité (Ex. envoy, istio-proxy).

Conteneur principal vs Init Container vs Sidecar

Exemple d'usage :

- Un conteneur principal exécute une API, le sidecar collecte les logs et les envoie à un système central.
- Un conteneur principal expose une application, le sidecar injecte des certificats TLS ou assure le chiffrement.

Cycle de vie d'un conteneur dans Kubernetes

Chaque conteneur dans un Pod suit un **cycle de vie** : création, démarrage, exécution, arrêt. Kubernetes fournit des **mécanismes pour surveiller et contrôler** ce cycle à travers les probes et les hooks.

1. Probes

Les **probes** sont des mécanismes de vérification de l'état des conteneurs.

Cycle de vie d’un conteneur dans Kubernetes

a. Liveness Probe

- Vérifie si le conteneur **est toujours en vie** (c'est-à-dire qu’il ne s’est pas figé).
- Si la probe échoue, **le conteneur est redémarré**.
- Typiquement utilisée pour détecter un blocage logiciel.

Cycle de vie d'un conteneur dans Kubernetes

Exemple :

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

Cycle de vie d'un conteneur dans Kubernetes

b. Readiness Probe

- Vérifie si le conteneur est **prêt à recevoir du trafic**.
- Si elle échoue, le pod est **retiré des endpoints** des services.
- Utile pour des applications qui prennent du temps à démarrer ou à charger des données.

Exemple :

```
readinessProbe:
  exec:
    command: ["cat", "/tmp/ready"]
  initialDelaySeconds: 5
  periodSeconds: 3
```

Cycle de vie d'un conteneur dans Kubernetes

c. Startup Probe

- Spécialement conçue pour les applications lentes à démarrer.
- Permet de différer l'exécution des autres probes.
- Si définie, bloque les `livenessProbe` jusqu'à réussite.

Cycle de vie d'un conteneur dans Kubernetes

2. Hooks

Les **hooks** permettent d'exécuter des actions à des moments précis de la vie d'un conteneur.

a. `postStart`

- S'exécute **immédiatement après le démarrage** du conteneur.
- Peut servir à initialiser un service, vérifier un fichier, écrire un log.

```
lifecycle:
  postStart:
    exec:
      command: ["sh", "-c", "echo 'Started' >> /var/log/status"]
```

Cycle de vie d'un conteneur dans Kubernetes

b. `preStop`

- S'exécute **juste avant l'arrêt du conteneur**, mais **avant l'envoi du signal SIGTERM**.
- Sert à :
  - vider une file de messages,
  - sauvegarder l'état temporaire,
  - avertir un système externe.

```
lifecycle:
  preStop:
    exec:
```

Architecture multi-conteneurs dans un Pod

Un Pod peut contenir plusieurs conteneurs qui **collaborent** dans un espace partagé :

- **IP partagée** (tous les conteneurs utilisent la même adresse IP).
- **Volumes partagés** (chaque conteneur peut lire/écrire dans les mêmes dossiers).

Cette architecture permet de découper les responsabilités, et de réutiliser des composants génériques.

Architecture multi-conteneurs dans un Pod

1. Pattern Sidecar

Un **sidecar** ajoute une **fonctionnalité secondaire** au conteneur principal.

Exemples :

Conteneur principal	Sidecar associé	Fonction
Serveur web	tail + forward des logs	Centralisation des logs
Application Node.js	envoy ou istio-proxy	Sécurité, mTLS
API Python	conteneur certbot	Renouvellement TLS

Architecture multi-conteneurs dans un Pod

Avantages :

- Réutilisabilité.
- Séparation des préoccupations.
- Partage des volumes, logs, secrets, etc.

Architecture multi-conteneurs dans un Pod

2. Pattern Adapter (ou Ambassador)

Un **adapter** ou **ambassador** agit comme une **passerelle** entre l'application et un service externe.

Exemples :

- Un conteneur principal qui communique avec une base de données, et un adapter qui transforme les requêtes dans un format spécifique.
- Un proxy TCP qui modifie dynamiquement les adresses des destinations.

Introduction aux Volumes

Un volume dans Kubernetes est une abstraction qui permet aux conteneurs de stocker et partager des données de manière persistante. Contrairement aux volumes Docker qui sont liés à la durée de vie d'un conteneur, les volumes Kubernetes existent tant que le pod existe.

Types de Volumes

1. **Volumes éphémères**

- **emptyDir** : Créé lorsqu'un pod est assigné à un nœud et dure toute la durée de vie du pod. Idéal pour stocker des données temporaires ou échanger des données entre conteneurs d'un même pod.
- **configMap** et **secret** : Utilisés pour injecter des configurations et des secrets (comme des mots de passe ou des clés API) dans les pods sous forme de fichiers.

2. **Volumes persistants**
   - **PersistentVolume (PV) et **PersistentVolumeClaim (PVC)** : Les PV sont des ressources de stockage allouées par l'administrateur, et les PVC sont des demandes de stockage par les utilisateurs. Cette séparation permet une gestion flexible et un provisionnement dynamique du stockage.

- **hostPath** : Monte un fichier ou un répertoire du système de fichiers du nœud dans un pod. Il doit être utilisé avec précaution car il lie directement le cycle de vie des données à celui du nœud.

Types de Volumes

3. **Volumes réseau**

- **NFS** (Network File System) : Monte un répertoire distant via NFS. Il permet de partager des données entre différents pods et nœuds.
- **CephFS** et **GlusterFS** : Fournissent des systèmes de fichiers distribués qui peuvent être montés sur plusieurs nœuds et pods.

Création et Utilisation des Volumes

1. **Définition d'un Volume dans un Pod**
   Vous pouvez définir des volumes dans le spec d'un pod. Voici un exemple basique :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: nginx
      volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: html-volume
  volumes:
    - name: html-volume
      emptyDir: {}
```

Dans cet exemple, un volume `emptyDir` est monté dans le conteneur à `/usr/share/nginx/html`.

Création et Utilisation des Volumes

PersistentVolume :

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

PersistentVolumeClaim :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Création et Utilisation des Volumes

Utilisation du PVC dans un Pod :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
    - name: example-container
      image: nginx
      volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: html-volume
  volumes:
    - name: html-volume
      persistentVolumeClaim:
        claimName: example-pvc
```

Volumes Dynamiques

Kubernetes supporte le provisionnement dynamique de volumes, ce qui signifie que les volumes peuvent être créés à la demande lorsque des PVC sont créés, en utilisant des `StorageClass`.

StorageClass :

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

PersistentVolumeClaim utilisant une StorageClass :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-fast-pvc
spec:
  storageClassName: fast
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volumes Bonnes Pratiques

- **Séparation des préoccupations** : Utilisez des PVC pour permettre aux développeurs de demander du stockage sans se soucier des détails de l'implémentation.
- **Sécurité** : Utilisez des volumes secrets pour les informations sensibles et gérez les accès avec RBAC.
- **Réplication et Sauvegarde** : Pour les données critiques, utilisez des solutions de stockage qui supportent la réplication et les sauvegardes automatiques.

Concepts fondamentaux des Services Kubernetes

Un Service Kubernetes est une abstraction qui définit un ensemble logique de pods et une politique d'accès à ces pods. Il résout le problème de l'éphéméralité des pods en fournissant une interface réseau stable.

- **IPs dynamiques** : Les pods peuvent être créés/détruits avec des IPs changeantes
- **Découverte de services** : Comment les applications trouvent-elles d'autres services ?
- **Load balancing** : Répartition du trafic entre plusieurs instances
- **Exposition réseau** : Comment exposer une application à l'extérieur

Concepts fondamentaux des Services Kubernetes

- **Selectors** : Étiquettes pour identifier les pods cibles
- **Endpoints** : Liste dynamique des IPs des pods correspondants
- **kube-proxy** : Composant qui implémente les règles de routage
- **iptables/IPVS** : Technologies de routage réseau sous-jacentes

Concepts fondamentaux des Services Kubernetes

ClusterIP (Service interne)

ClusterIP est le type de service par défaut. Il expose le service uniquement à l'intérieur du cluster Kubernetes via une adresse IP virtuelle interne.

Concepts fondamentaux des Services Kubernetes

- **Portée** : Accessible uniquement depuis l'intérieur du cluster
- **IP virtuelle** : Kubernetes assigne automatiquement une IP de la plage de service (service CIDR)
- **Stabilité** : L'IP reste constante pendant toute la durée de vie du service
- **DNS** : Résolution automatique via `<service-name>.<namespace>.svc.cluster.local`

Concepts fondamentaux des Services Kubernetes

1. **Création** : Kubernetes assigne une IP virtuelle (VIP) depuis la plage de service
2. **Endpoints Controller** : Surveille les pods correspondant aux selectors
3. **kube-proxy** : Configure les règles de routage (iptables/IPVS) sur chaque nœud
4. **Load balancing** : Distribution round-robin par défaut entre les endpoints

Concepts fondamentaux des Services Kubernetes

- Communication inter-services dans le cluster
- Bases de données internes
- Services de cache (Redis, Memcached)
- APIs internes
- **Round-robin** : Répartition équitable (par défaut)
- **Session affinity** : Routage basé sur l'IP client
- **Weighted routing** : Possible avec des solutions tierces

Concepts fondamentaux des Services Kubernetes

NodePort (Exposé sur le nœud)

NodePort expose le service sur un port statique de chaque nœud du cluster. Il crée automatiquement un service ClusterIP et le rend accessible depuis l'extérieur.

- **Portée** : Accessible depuis l'extérieur via `<NodeIP>:<NodePort>`
- **Plage de ports** : 30000-32767 par défaut (configurable)
- **Haute disponibilité** : Disponible sur tous les nœuds du cluster
- **Redirection** : Chaque nœud redirige le trafic vers les pods cibles

Concepts fondamentaux des Services Kubernetes

1. **ClusterIP automatique** : Création d'un service ClusterIP sous-jacent
2. **Port allocation** : Assignation d'un port dans la plage NodePort
3. **iptables rules** : Configuration de règles de redirection sur chaque nœud
4. **Cross-node routing** : Le trafic peut être routé vers des pods sur d'autres nœuds

Concepts fondamentaux des Services Kubernetes

- Simple à configurer
- Pas besoin de load balancer externe
- Fonctionne sur tous les environnements Kubernetes

Concepts fondamentaux des Services Kubernetes

Inconvénients :

- Exposition directe des nœuds (sécurité)
- Gestion manuelle des ports
- Pas de terminaison SSL native
- Scalabilité limitée (nombre de ports)

Concepts fondamentaux des Services Kubernetes

- **Firewall** : Nécessité de configurer les règles firewall
- **Exposition** : Les nœuds deviennent des points d'entrée
- **DDoS** : Vulnérabilité aux attaques directes

Concepts fondamentaux des Services Kubernetes

LoadBalancer (via provider cloud ou MetalLB)

LoadBalancer expose le service via un load balancer externe fourni par l'infrastructure cloud ou une solution comme MetalLB.

Concepts fondamentaux des Services Kubernetes

Architecture cloud :

- **Intégration** : Kubernetes communique avec l'API du provider cloud
- **Provisioning** : Création automatique d'un load balancer externe
- **IP publique** : Attribution d'une IP publique dédiée
- **Health checks** : Surveillance automatique de la santé des nœuds

Concepts fondamentaux des Services Kubernetes

Providers supportés :

- **AWS** : Elastic Load Balancer (ELB/ALB/NLB)
- **GCE** : Google Cloud Load Balancer
- **Azure** : Azure Load Balancer
- **DigitalOcean** : DigitalOcean Load Balancer

Concepts fondamentaux des Services Kubernetes

MetalLB (Load Balancer on-premise) :

1. **Layer 2 Mode** :
   - Un nœud annonce l'IP virtuelle via ARP
   - Failover automatique en cas de panne
   - Simple mais limité à un subnet
2. **BGP Mode** :
   - Annonce des routes via Border Gateway Protocol
   - Véritable load balancing

Concepts fondamentaux des Services Kubernetes

- **Address pools** : Définition des plages d'IPs disponibles
- **BGP peers** : Configuration des routeurs BGP
- **Advertisement** : Contrôle de l'annonce des IPs

Avantages :

- IP dédiée et stable
- Intégration native avec l'infrastructure
- Health checks automatiques
- Haute disponibilité

Concepts fondamentaux des Services Kubernetes

Headless Services

Un Headless Service est un service sans ClusterIP (spec.clusterIP: "None"). Au lieu de load balancer le trafic, il retourne directement les IPs des pods via DNS.

Caractéristiques :

- **Pas de ClusterIP** : Aucune IP virtuelle assignée
- **DNS multi-A records** : Retourne toutes les IPs des pods
- **Découverte directe** : Les clients peuvent choisir le pod à contacter
- **StatefulSets** : Utilisé pour les applications stateful

Concepts fondamentaux des Services Kubernetes

Fonctionnement DNS :

- **Avec selector** : Retourne les IPs de tous les pods matching
- **Sans selector** : Permet de pointer vers des services externes
- **SRV records** : Information sur les ports nommés

Concepts fondamentaux des Services Kubernetes

- **Bases de données clustérisées** : MongoDB, Cassandra
- **Applications stateful** : Qui nécessitent une identité stable
- **Service discovery** : Quand l'application gère elle-même le load balancing
- **Proxies** : Services qui routent vers des backends spécifiques

Concepts fondamentaux des Services Kubernetes

Ingress Controllers

Un Ingress Controller est un composant qui gère l'accès externe aux services via HTTP/HTTPS. Il implémente les règles définies dans les ressources Ingress.

Concepts fondamentaux des Services Kubernetes

- **Ingress Resource** : Définition déclarative des règles de routage
- **Ingress Controller** : Implémentation qui applique les règles
- **Load Balancer** : Point d'entrée externe (souvent un LoadBalancer)

Concepts fondamentaux des Services Kubernetes

1. **Host-based routing** :
   - Routage basé sur le nom d'hôte (SNI)
   - Virtual hosting
   - Certificats SSL par domaine
2. **Path-based routing** :
   - Routage basé sur le chemin URL
   - Microservices sur différents paths
   - Réécriture d'URL possible

Concepts fondamentaux des Services Kubernetes

NGINX Ingress Controller :

- **Performance** : Très performant
- **Flexibilité** : Nombreuses annotations
- **SSL/TLS** : Support complet
- **Websockets** : Support natif

Traefik :

- **Auto-discovery** : Configuration automatique
- **Dashboard** : Interface web intégrée
- **Middlewares** : Système de plugins

Concepts fondamentaux des Services Kubernetes

HAProxy Ingress :

- **Performance** : Très haute performance
- **Équilibrage** : Algorithmes avancés
- **Observabilité** : Métriques détaillées

Concepts fondamentaux des Services Kubernetes

Istio Gateway :

- **Service Mesh** : Intégration avec Istio
- **Sécurité** : mTLS, policies avancées
- **Observabilité** : Tracing distribué

Les CNI (Container Network Interface)

Les CNI sont des plugins qui gèrent la connectivité réseau des pods dans Kubernetes. Ils sont responsables de l'attribution des adresses IP aux pods, de la création des interfaces réseau virtuelles et de la gestion du routage entre les différents nœuds du cluster.
K3s (et donc k3d) utilise par défaut **Flannel** comme CNI, mais vous pouvez le remplacer par Calico pour bénéficier de fonctionnalités réseau plus avancées, notamment les Network Policies.

Calico : Présentation et avantages

Calico est un CNI particulièrement populaire qui offre plusieurs avantages :

- **Sécurité réseau avancée** : Support complet des Network Policies Kubernetes
- **Performance** : Utilise le routage IP natif (BGP) plutôt que l'overlay
- **Scalabilité** : Conçu pour les grands clusters
- **Observabilité** : Outils de monitoring et debugging intégrés

Configuration de Calico avec k3d

Pour utiliser Calico avec k3d, vous devez désactiver le CNI par défaut et installer Calico manuellement :

1. Création du cluster k3d sans CNI

```
# Créer un cluster k3d sans le CNI par défaut
k3d cluster create mon-cluster \
  --k3s-arg "--flannel-backend=none@server:*" \
  --k3s-arg "--disable-network-policy@server:*"
```

Configuration de Calico avec k3d

2. Installation de Calico

```
# Installer l'opérateur Calico
kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/manifests/tigera-operator.yaml

# Configurer Calico
kubectl create -f - <<EOF
apiVersion: operator.tigera.io/v1
kind: Installation
metadata:
  name: default
spec:
  calicoNetwork:
    ipPools:
    - blockSize: 26
      cidr: 10.42.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
EOF
```

Configuration de Calico avec k3d

3. Vérification de l'installation

```
# Vérifier que tous les pods Calico sont en cours d'exécution
kubectl get pods -n calico-system

# Vérifier les nœuds
kubectl get nodes -o wide
```

Network Policies : Concepts et implémentation

Les Network Policies permettent de contrôler le trafic réseau entre les pods. Elles fonctionnent comme un firewall au niveau applicatif.

1. Politique Deny-All (Bloquer tout le trafic)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: production
spec:
  podSelector: {}  # Sélectionne tous les pods du namespace
  policyTypes:
  - Ingress
  # Aucune règle ingress = tout est bloqué
```

Network Policies : Concepts et implémentation

2. Politique Allow-Ingress basique

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-web-access
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: web-server
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
```

Stratégies d'implémentation recommandées

1. **Commencer par l'observation** : Utilisez des outils comme `kubectl logs` et les métriques Calico pour comprendre les flux actuels
2. **Implémenter une politique deny-all** : Commencez par bloquer tout le trafic
3. **Ajouter des exceptions graduellement** : Autorisez les communications nécessaires une par une
4. **Tester intensivement** : Vérifiez que les applications fonctionnent correctement

Jobs : Exécution de tâches unitaires

Principe fondamental

Les Jobs dans Kubernetes incarnent le paradigme de l'exécution de tâches à durée déterminée, contrairement aux Deployments qui maintiennent un état permanent. Ils représentent une abstraction pour des processus qui doivent s'exécuter jusqu'à leur completion réussie, puis se terminer proprement.

Jobs : Exécution de tâches unitaires

Sémantique d'exécution

Un Job garantit qu'un nombre spécifique de pods exécutent une tâche jusqu'à sa terminaison avec succès. Cette garantie repose sur le principe de **terminaison contrôlée** : le Job monitore continuellement l'état des pods et s'assure que la tâche soit menée à son terme selon les critères définis (code de sortie 0, par exemple).

Jobs : Exécution de tâches unitaires

Modèle de completion

Le Job opère selon trois modes principaux :

- **Non-parallel** : un seul pod exécute la tâche
- **Parallel with fixed completion count** : plusieurs pods exécutent la tâche avec un nombre fixe de complétions
- **Parallel with work queue** : plusieurs pods consomment des éléments d'une queue jusqu'à épuisement

CronJobs : Orchestration temporelle

Paradigme de planification

Les CronJobs étendent la logique des Jobs en y ajoutant une dimension temporelle basée sur la syntaxe cron Unix. Ils représentent l'implémentation Kubernetes du concept de **tâches planifiées récurrentes**, permettant l'automatisation de processus selon des cycles temporels définis.

CronJobs : Orchestration temporelle

Mécanisme de déclenchement

Le contrôleur CronJob évalue périodiquement l'expression cron et détermine si une nouvelle instance de Job doit être créée. Cette évaluation suit une logique de **fenêtre temporelle** : si le système détermine qu'une exécution aurait dû avoir lieu (basée sur l'horloge système), il génère automatiquement le Job correspondant.

CronJobs : Orchestration temporelle

Gestion de la concurrence

Les CronJobs implémentent des politiques de concurrence sophistiquées :

- **Allow** : permet l'exécution simultanée de plusieurs instances
- **Forbid** : empêche le démarrage si une instance est déjà en cours
- **Replace** : termine l'instance en cours et démarre une nouvelle

CronJobs : Orchestration temporelle

Cas d'usage concrets et leurs implications théoriques

Génération de rapports

Ce cas illustre le pattern **Extract-Transform-Load (ETL)** dans un environnement containerisé. Le Job accède aux sources de données, applique les transformations nécessaires, et produit un artefact (rapport) dans un système de stockage persistant. La théorie sous-jacente repose sur l'**idempotence** : exécuter le Job plusieurs fois avec les mêmes paramètres doit produire le même résultat.

CronJobs : Orchestration temporelle

Envoi de mails

Représente un pattern de **notification asynchrone** où le Job découple l'événement déclencheur de l'action de notification. Théoriquement, cela implémente le principe de **séparation des préoccupations** : le système principal continue son fonctionnement tandis que la notification est traitée indépendamment.

CronJobs : Orchestration temporelle

Synchronisation de données

Illustre le concept de **réconciliation périodique** entre différents systèmes. Le Job agit comme un agent de synchronisation qui évalue l'état actuel contre l'état désiré et applique les modifications nécessaires pour atteindre la convergence.

CronJobs : Orchestration temporelle

Stratégies de contrôle et de résilience

Politiques de redémarrage

Les Jobs implémentent des stratégies de redémarrage sophistiquées basées sur la nature de l'échec :

- **Never** : aucun redémarrage automatique
- **OnFailure** : redémarrage du container en cas d'échec
- **Always** : redémarrage systématique (généralement inadapté pour les Jobs)

CronJobs : Orchestration temporelle

Mécanisme de gestion des échecs

Le système suit une logique de **backoff exponentiel** optionnel et maintient un compteur d'échecs. La théorie du **circuit breaker** peut être appliquée : après un certain nombre d'échecs consécutifs, le Job peut être marqué comme définitivement échoué pour éviter la consommation inutile de ressources.

CronJobs : Orchestration temporelle

Limitation des exécutions

Les concepts de **activeDeadlineSeconds** et **backoffLimit** implémentent des garde-fous contre les exécutions infinies ou les boucles d'échec. Ces mécanismes reposent sur la théorie de **bounded execution** : toute tâche doit avoir des limites définies en termes de temps et de tentatives pour garantir la stabilité du système.

CronJobs : Orchestration temporelle

Nettoyage et gouvernance

Les Jobs implémentent des politiques de **garbage collection** avec des paramètres comme **ttlSecondsAfterFinished** et **successfulJobsHistoryLimit**. Cette approche théorique suit le principe de **lifecycle management** : chaque ressource a un cycle de vie défini avec des phases de création, exécution, completion, et nettoyage.