# KUBERNETES

## MASTERCLASS

**MENRA W. ROMIAL**
PhD Student & Kubernetes Developper

Guide Complet pour Maîtriser
l'Orchestration de Conteneurs

De l'installation aux techniques avancées en production

ÉDITION 2025

# À Propos de ce Guide

Ce guide complet vous accompagne dans votre apprentissage de Kubernetes, de l'installation de votre premier cluster jusqu'aux techniques avancées d'orchestration en production. Chaque chapitre combine théorie, exemples pratiques et exercices pour une maîtrise progressive et solide.

**Objectifs pédagogiques :**

- Comprendre l'architecture et les concepts fondamentaux
- Maîtriser le déploiement et la gestion d'applications
- Implémenter la surveillance et la sécurité
- Optimiser les performances et la scalabilité

# Table des Matières

## Table des matières

# Table des figures



# Listings

# 1 Introduction et Préparation de l'Environnement

## §1.1 Pourquoi Kubernetes ?

Dans le paysage technologique actuel, le développement et le déploiement d'applications distribuées sont devenus la norme. Cependant, la gestion de ces applications à grande échelle, surtout lorsqu'elles sont conteneurisées, présente des défis significatifs : déploiement cohérent sur divers environnements, mise à l'échelle dynamique en fonction de la charge, garantie de la haute disponibilité, découverte de services, et mises à jour sans interruption de service.

Face à ces complexités, l'orchestration de conteneurs s'est imposée comme une solution incontournable. Kubernetes, projet initialement développé par Google et aujourd'hui maintenu par la Cloud Native Computing Foundation (CNCF), est rapidement devenu le standard de facto pour l'orchestration de conteneurs [7].

**Définition 1.1: Orchestration de Conteneurs**

L'orchestration de conteneurs est le processus d'automatisation du déploiement, de la gestion, de la mise à l'échelle, du réseautage et de la disponibilité des applications basées sur des conteneurs. Elle abstrait la complexité de l'infrastructure sous-jacente, permettant aux développeurs de se concentrer sur la création de valeur applicative.

Les principaux avantages offerts par Kubernetes incluent :

- **Haute Disponibilité et Résilience :** Kubernetes peut redémarrer automatiquement les conteneurs défaillants, remplacer et replanifier les conteneurs lorsque les nœuds meurent, et ne proposer du trafic qu'aux conteneurs prêts à le recevoir.
- **Scalabilité Horizontale :** Augmentez ou diminuez facilement le nombre de répliques de votre application, manuellement ou automatiquement en fonction de l'utilisation du CPU ou d'autres métriques.

- **Déploiements et Rollbacks Automatisés :** Kubernetes permet de décrire l'état désiré de votre déploiement, et il se charge de faire converger l'état actuel vers cet état désiré. Il supporte également les mises à jour progressives (rolling updates) et les retours en arrière (rollbacks) en cas de problème [9, voir Chapitre 5].
- **Découverte de Services et Équilibrage de Charge :** Kubernetes attribue des adresses IP stables à des groupes de conteneurs et peut équilibrer la charge entre eux.
- **Gestion de la Configuration et des Secrets :** Déployez et mettez à jour les configurations et les informations sensibles sans avoir à reconstruire vos images de conteneurs.
- **Optimisation des Ressources :** Kubernetes permet de spécifier les besoins en CPU et en mémoire pour chaque conteneur, optimisant ainsi l'utilisation des ressources de l'infrastructure.
- **Portabilité :** Exécutez vos applications de manière cohérente sur des infrastructures on-premise, des clouds publics ou des environnements hybrides.

**Pro Tips 1.1: Philosophie Déclarative**

Un des principes fondamentaux de Kubernetes est son approche déclarative. Plutôt que de spécifier une série d'instructions impératives (comment faire), vous décrivez l'état final souhaité de votre application (quoi faire). Kubernetes travaille ensuite en continu pour atteindre et maintenir cet état. Cela simplifie grandement la gestion et l'automatisation.

## §1.2 Qu'est-ce que Kubernetes ?

Kubernetes (souvent abrégé en K8s, où le "8" représente les huit lettres entre "K" et "s") est un système open-source portable et extensible pour l'automatisation du déploiement, de la mise à l'échelle et de la gestion des applications conteneurisées. Il regroupe les conteneurs qui constituent une application en unités logiques appelées Pods pour faciliter leur gestion et leur découverte.

**Note Importante 1.1: Origines : Google Borg**

Kubernetes s'inspire largement du système interne de Google appelé Borg, utilisé pendant plus d'une décennie pour gérer des milliards de conteneurs par semaine. De nombreuses idées et bonnes pratiques issues de Borg ont été intégrées dans Kubernetes, qui a ensuite été offert à la communauté open-source [7]. La documentation officielle de Kubernetes [25] offre également un aperçu historique.

Kubernetes fournit une "plateforme pour automatiser le déploiement, la mise à l'échelle et les opérations des conteneurs d'application sur des clusters de serveurs". Il n'est pas une solution PaaS (Platform as a Service) complète, mais il fournit les briques de base sur lesquelles de telles plateformes peuvent être construites.

**Exemple Pratique 1.1: Cas d'Usage Typiques**

Kubernetes est utilisé dans une multitude de scénarios, par exemple :

- Déploiement d'applications microservices hautement disponibles.
- Exécution de traitements batch et de tâches de calcul intensif.
- Migration d'applications monolithiques vers une architecture conteneurisée.
- Création de plateformes de développement et de test reproductibles.
- Gestion d'applications stateful comme des bases de données (avec des opérateurs).

De nombreuses entreprises, des startups aux grandes multinationales, ont adopté Kubernetes pour moderniser leur infrastructure et accélérer leurs cycles de développement.

## §1.3 Rappel Essentiel sur les Conteneurs (Docker)

Bien que Kubernetes puisse fonctionner avec différents runtimes de conteneurs (comme containerd ou CRI-O), Docker reste l'un des plus populaires et a grandement contribué à la démocratisation de la conteneurisation. Une compréhension de base des conteneurs est indispensable avant d'aborder Kubernetes.

**Définition 1.2: Conteneur**

Un conteneur est une unité logicielle standardisée qui empaquette le code et toutes ses dépendances (bibliothèques, outils système, code d'exécution) afin que l'application s'exécute rapidement et de manière fiable d'un environnement informatique à un autre. Les images Docker sont des modèles légers, autonomes et exécutables pour la création de conteneurs. Pour plus de détails, la documentation de Docker est une excellente ressource [4].

Les concepts clés à retenir sont :

- **Image Docker :** Un template en lecture seule avec des instructions pour créer un conteneur. Les images sont construites à partir d'un fichier appelé `Dockerfile`.
- **Conteneur Docker :** Une instance exécutable d'une image. Vous pouvez créer, démarrer, arrêter, déplacer et supprimer un conteneur à l'aide de l'API ou de la CLI Docker.
- **Dockerfile :** Un fichier texte qui contient toutes les commandes, dans l'ordre, nécessaires pour construire une image Docker donnée.
- **Docker Hub / Registre de Conteneurs :** Un service de registre (comme Docker Hub) où les images Docker sont stockées et partagées.

**Code Block 1.1: Dockerfile simple pour une application Node.js**

```
$
# Utiliser une image de base officielle Node.js
FROM node:18-alpine

# Cr er le r pertoire de l'application
WORKDIR /usr/src/app

# Copier les fichiers package.json et package-lock.json
COPY package*.json ./

# Installer les d pendances de l'application
RUN npm install

# Copier le reste du code source de l'application
COPY . .

# Exposer le port sur lequel l'application  coute
EXPOSE 3000

# Commande pour d marrer l'application
CMD [ "node", "server.js" ]
```

**Listing 1.1 –** Exemple de Dockerfile minimaliste

Le Dockerfile présenté dans le Listing 1.1 illustre les étapes typiques pour conteneuriser une application Node.js.

**Attention 1.1: Prérequis : Familiarité avec Docker**

Ce guide suppose une certaine aisance avec les concepts et commandes de base de Docker. Si vous êtes nouveau dans le monde des conteneurs, nous vous recommandons vivement de vous familiariser avec Docker avant de plonger profondément dans Kubernetes. Des ressources comme Poulton [9] consacrent également des sections aux fondamentaux de Docker.

## §1.4 Mise en Place de l'Environnement de Travail Kubernetes

Pour suivre les exercices pratiques de ce cours, vous aurez besoin d'un cluster Kubernetes. Plusieurs options s'offrent à vous, notamment pour un environnement local :

- **Minikube :** C'est un outil qui permet d'exécuter un cluster Kubernetes à nœud unique localement sur votre machine (macOS, Linux ou Windows). Il est idéal pour apprendre et développer. L'installation est simple et bien documentée [28].

**Code Block 1.2: Démarrer Minikube**

```
- $
1           # D marrer un cluster Minikube (peut n cessiter un ↩
      ↪ driver sp cifique comme virtualbox, docker, etc.)
2           minikube start
3
4           # V rifier le statut du cluster
5           minikube status
6
7           # Arr ter le cluster
8           minikube stop
9
10          # Supprimer le cluster
11          minikube delete
12
```

**Listing 1.2 –** Commandes de base Minikube

- **kind (Kubernetes in Docker) :** 'kind' est un outil pour exécuter des clusters Kubernetes locaux en utilisant des conteneurs Docker comme "nœuds". Il est très rapide à démarrer et est souvent utilisé pour les tests CI/CD.
- **Docker Desktop Kubernetes :** Si vous utilisez Docker Desktop (sur Windows ou macOS), il inclut une option pour activer un cluster Kubernetes mono-nœud intégré.
- **Clusters Cloud Managés (AKS, EKS, GKE) :** Si vous avez accès à un fournisseur de cloud (Azure Kubernetes Service, Amazon Elastic Kubernetes Service, Google Kubernetes Engine), vous pouvez utiliser un cluster managé. Cela offre une expérience plus proche de la production mais peut engendrer des coûts.

Une fois votre cluster en place (local ou distant), vous interagirez principalement avec lui via l'outil en ligne de commande `kubectl`.

**Pro Tips 1.2: Installation de kubectl**

`kubectl` est l'outil en ligne de commande essentiel pour interagir avec n'importe quel cluster Kubernetes. Suivez les instructions d'installation officielles disponibles sur le site de Kubernetes [25] pour votre système d'exploitation. Une fois installé, vous devrez configurer `kubectl` pour qu'il pointe vers votre cluster (Minikube et Docker Desktop le font souvent automatiquement).

Pour vérifier que `kubectl` est correctement configuré et peut communiquer avec votre cluster, exécutez :

**Code Block 1.3: Vérification de la connexion au cluster**

```
$
# Afficher les informations du cluster
kubectl cluster-info

# Lister les n uds du cluster (pour un cluster mono-n ud local, vous devriez en voir un)
kubectl get nodes
```

**Listing 1.3 –** Test de la connexion kubectl

Si ces commandes retournent des informations sans erreur, votre environnement est prêt pour la suite de ce cours !

**Question Ouverte 1.1: Votre Configuration**

Quel type de cluster Kubernetes avez-vous choisi pour ce cours ? Avez-vous rencontré des difficultés lors de sa mise en place ou de la configuration de `kubectl` ? Partagez vos expériences et questions.

# 2 Architecture de Kubernetes

Comprendre l'architecture de Kubernetes est fondamental pour utiliser efficacement la plate-forme et pour diagnostiquer les problèmes. Ce chapitre décompose les principaux composants de Kubernetes, expliquant leurs rôles et interactions.

## §2.1 Vue d'Ensemble de l'Architecture

Un cluster Kubernetes est composé de deux types principaux de serveurs (ou nœuds) :

- Les **Nœuds du Control Plane** (anciennement appelés Master Nodes) : Ils hébergent les composants qui prennent les décisions globales du cluster (par exemple, la planification des applications), ainsi que la détection et la réponse aux événements du cluster.
- Les **Nœuds Workers** (ou Nœuds de Travail) : Ce sont les machines (physiques ou virtuelles) qui exécutent les applications conteneurisées. Chaque nœud worker est géré par le Control Plane.

L'ensemble de ces nœuds forme le cluster Kubernetes. Typiquement, pour la résilience, un cluster de production aura plusieurs nœuds de Control Plane et de nombreux nœuds workers [7].

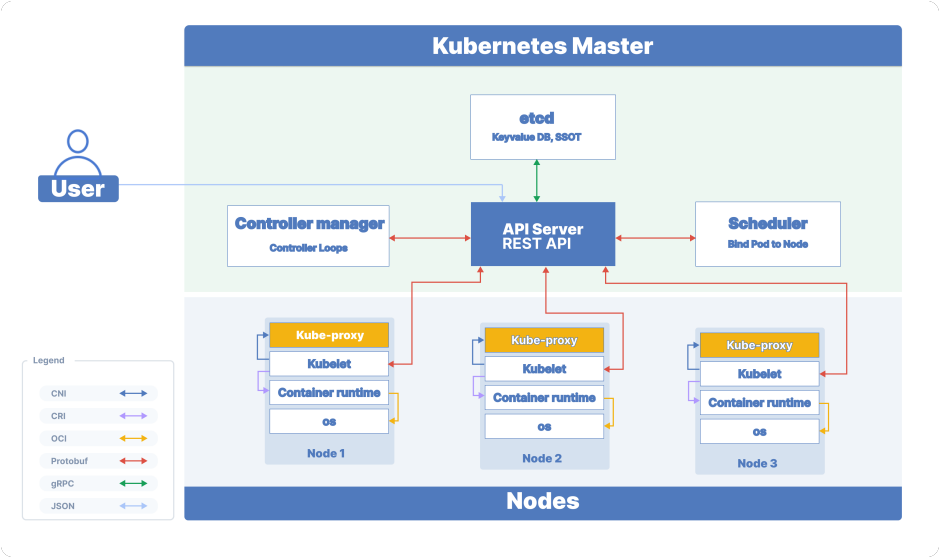


**FIGURE 2.1 –** Diagramme simplifié de l'architecture Kubernetes.

### 2.1.1 Composants du Control Plane

Les composants du Control Plane sont essentiels au fonctionnement du cluster. Ils sont responsables de la gestion globale et de l'état du cluster. La documentation officielle [24] détaille chacun de ces éléments.

- **kube-apiserver :**

**Définition 2.1: kube-apiserver**

Le `kube-apiserver` est le composant central du Control Plane. Il expose l'API Kubernetes, qui est le frontend du Control Plane. C'est par cette API que les utilisateurs, les outils en ligne de commande (comme `kubectl`), et les autres composants du cluster interagissent. Il est responsable de la validation et de la configuration des données pour les objets API tels que les Pods, Services, ReplicaSets, etc. [41].

Toutes les opérations de modification de l'état du cluster passent par l'API server.

- **etcd :**

**Définition 2.2: etcd**

`etcd` est une base de données clé-valeur distribuée, cohérente et hautement disponible, utilisée comme magasin de données principal de Kubernetes pour toutes les données du cluster. Toutes les configurations, les états et les métadonnées du cluster sont stockés dans `etcd`. Avoir un `etcd` fiable est critique pour la stabilité du cluster. Pour plus d'informations sur `etcd`, consultez sa documentation officielle [5].

**⚠ Attention 2.1: Importance d'etcd**

La perte des données d'**etcd** signifie la perte de l'état de votre cluster Kubernetes. Des sauvegardes régulières d'`etcd` sont cruciales pour les environnements de production.

- **kube-scheduler :** Le `kube-scheduler` surveille les Pods nouvellement créés qui n'ont pas encore été assignés à un nœud. Pour chaque Pod qu'il découvre, le scheduler prend une décision de placement, c'est-à-dire qu'il sélectionne un nœud optimal pour que le Pod s'exécute. Ce choix est basé sur divers facteurs tels que les demandes de ressources, les contraintes matérielles/logicielles/politiques, les spécifications d'affinité et d'anti-affinité, la localité des données, etc.
- **kube-controller-manager :** Le `kube-controller-manager` exécute les processus des contrôleurs. Les contrôleurs sont des boucles de contrôle qui observent l'état du cluster via l'API server et effectuent des changements pour tenter de faire correspondre l'état actuel à l'état désiré. Chaque contrôleur est responsable d'une ressource spécifique. Exemples de contrôleurs inclus :
  - *Node Controller :* Responsable de la détection et de la réponse lorsque des nœuds tombent en panne.
  - *Replication Controller (et ReplicaSet Controller) :* Maintient le nombre correct de Pods pour chaque objet de réplication dans le système.
  - *Endpoints Controller :* Populе l'objet Endpoints (c'est-à-dire, joint les Services et les Pods).
  - *Service Account & Token Controllers :* Crée les comptes de service par défaut et les jetons d'accès API pour les nouveaux namespaces.

  Poulton [9] offre une bonne explication du rôle des différents contrôleurs.
- **cloud-controller-manager (Optionnel) :** Ce composant exécute des contrôleurs spécifiques aux fournisseurs de cloud. Il permet de lier votre cluster à l'API de votre fournisseur de cloud et sépare les composants qui interagissent avec la plateforme cloud des composants qui interagissent uniquement avec votre cluster. Si vous exécutez Kubernetes on-premise, ou dans un environnement d'apprentissage sur votre propre PC, le cluster n'a pas de cloud-controller-manager.

### 2.1.2 Composants des Nœuds Workers

Les nœuds workers sont les machines où vos charges de travail (Pods) s'exécutent.

- **kubelet :**

**🎓 Définition 2.3: kubelet**

Le `kubelet` est un agent qui s'exécute sur chaque nœud du cluster. Il s'assure que les conteneurs décrits dans les PodSpecs (spécifications de Pods) fournis par l'API server sont en cours d'exécution et en bonne santé sur son nœud. Le `kubelet` ne

gère pas les conteneurs qui n'ont pas été créés par Kubernetes.

- **kube-proxy :** Le `kube-proxy` est un proxy réseau qui s'exécute sur chaque nœud de votre cluster, implémentant une partie du concept de Service Kubernetes. Il maintient les règles réseau sur les nœuds, permettant la communication réseau vers vos Pods depuis des sessions réseau à l'intérieur ou à l'extérieur de votre cluster. `kube-proxy` utilise soit la couche de filtrage de paquets du système d'exploitation (comme iptables ou IPVS sous Linux), soit transfère lui-même le trafic.
- **Container Runtime :** Le runtime de conteneur est le logiciel responsable de l'exécution des conteneurs. Kubernetes prend en charge plusieurs runtimes de conteneurs : Docker, containerd, CRI-O, ainsi que toute implémentation de la Kubernetes CRI (Container Runtime Interface).

**Note Importante 2.1: Docker et containerd**

Historiquement, Docker était le runtime le plus couramment utilisé avec Kubernetes. Cependant, Kubernetes a évolué pour utiliser directement des runtimes conformes à la CRI, comme containerd (qui est un composant central de Docker lui-même) ou CRI-O. Pour l'utilisateur final, le choix du runtime est souvent transparent.

## §2.2 Le Modèle d'Objet Kubernetes et YAML

Dans Kubernetes, tout ce que vous créez ou gérez est une *ressource* ou un *objet*. Ces objets représentent l'état de votre cluster : les applications déployées, leurs configurations, les politiques réseau, etc. La plupart des objets Kubernetes incluent deux champs imbriqués qui gouvernent leur configuration : la `spec` (spécification) et le `status` (état).

- `spec` : Décrit l'état *désiré* de l'objet, que vous fournissez.
- `status` : Décrit l'état *actuel* de l'objet, géré et mis à jour par Kubernetes.

Kubernetes travaille constamment pour que l'état actuel corresponde à l'état désiré.

**Pro Tips 2.1: Tout est API**

Tous les objets Kubernetes sont accessibles et manipulables via l'API Kubernetes. `kubectl` est un client qui utilise cette API pour interagir avec le cluster [41].

Pour définir ces objets, on utilise le plus souvent le format YAML (YAML Ain't Markup Language), bien que JSON soit également supporté. YAML est privilégié pour sa lisibilité par les humains.

Un fichier YAML typique pour un objet Kubernetes possède au minimum les champs suivants :

- `apiVersion` : La version de l'API Kubernetes utilisée pour créer cet objet (ex : `v1`, `apps/v1`).
- `kind` : Le type d'objet que vous souhaitez créer (ex : `Pod`, `Deployment`, `Service`).
- `metadata` : Des données qui aident à identifier l'objet de manière unique, comme un `name`, un `UID`, et optionnellement un `namespace` et des `labels`.
- `spec` : La spécification de l'état désiré pour l'objet. La structure exacte du champ `spec` varie en fonction du type d'objet.

**Code Block 2.1: Structure YAML d'un objet Kubernetes (générique)**

```
- $
1  apiVersion: group/version # Exemple: apps/v1, v1, networking.k8s.io↩
       ↪ /v1
2  kind: KindOfObject      # Exemple: Deployment, Pod, Service, ↩
       ↪ Ingress
3  metadata:
4    name: my-object-name
5    namespace: my-namespace # Optionnel, d pend si l'objet est ↩
       ↪ namespac
6    labels:                 # Optionnel
7      key1: value1
8      key2: value2
9    annotations:            # Optionnel
10     annotation-key1: annotation-value1
11 spec:
12   # ... La structure de 'spec' d pend du 'kind' de l'objet
13   # Par exemple, pour un Deployment:
14   # replicas: 3
15   # selector:
16   #   matchLabels:
17   #     app: my-app
18   # template:
19   #   # ... d finition du template de Pod
20   # Pour un Service:
21   # type: ClusterIP
22   # selector:
23   #   app: my-app
24   # ports:
25   # - protocol: TCP
26   #   port: 80
27   #   targetPort: 8080
28 status:
29   # ... Champ en lecture seule, g  r   par Kubernetes pour ↩
       ↪ refl ter l' tat  actuel.
30   # Non sp cifi  dans les manifestes que vous appliquez.
```

**Listing 2.1 –** Structure de base d'un manifeste Kubernetes

Le Listing 2.1 montre la structure générale. Nous explorerons des exemples concrets pour chaque type d'objet dans les chapitres suivants.

## §2.3 Interaction avec Kubernetes via `kubectl`

`kubectl` est l'outil en ligne de commande principal pour interagir avec le Control Plane de Kubernetes via son API. Il vous permet de déployer des applications, inspecter et gérer les ressources du cluster, consulter les logs, et bien plus encore.

La syntaxe générale d'une commande `kubectl` est : `kubectl [commande] [TYPE] [NOM] [flags]`

- `commande` : Spécifie l'opération à effectuer (ex : `get`, `describe`, `apply`, `delete`, `logs`, `exec`).
- `TYPE` : Spécifie le type de ressource (ex : `pods`, `services`, `deployments`). Sensible à la casse et peut être abrégé (ex : `po` pour pods, `svc` pour services).
- `NOM` : Spécifie le nom de la ressource spécifique. Ce champ est optionnel pour certaines commandes (ex : `kubectl get pods` liste tous les pods).
- `flags` : Spécifie des options supplémentaires (ex : `-n <namespace>`, `-o yaml`, `--all-namespaces`).

**Exemple Pratique 2.1: Commandes `kubectl` courantes**

Voici quelques commandes fondamentales pour commencer :

- `kubectl get pods` : Lister tous les Pods dans le namespace courant.
- `kubectl get deployments -n kube-system` : Lister les Deployments dans le namespace `kube-system`.
- `kubectl describe pod <nom-du-pod>` : Afficher des informations détaillées sur un Pod spécifique.
- `kubectl apply -f mon-fichier.yaml` : Créer ou mettre à jour une ressource à partir d'un fichier YAML.
- `kubectl delete service <nom-du-service>` : Supprimer un Service spécifique.
- `kubectl logs <nom-du-pod>` : Afficher les logs d'un conteneur dans un Pod (si un seul conteneur).
- `kubectl exec -it <nom-du-pod> -- /bin/sh` : Exécuter une commande interactive (shell) dans un conteneur.

Nous utiliserons intensivement `kubectl` tout au long de ce guide. Il est recommandé de consulter sa documentation de référence [disponible via "kubectl --help" ou sur 25] pour découvrir l'étendue de ses capacités.

**Pro Tips 2.2: Approche Déclarative vs Impérative avec kubectl**

`kubectl` supporte les deux approches :

- **Impérative :** Vous donnez des ordres directs (ex : `kubectl run nginx –image=nginx`, `kubectl expose deployment nginx –port=80`). Utile pour des tâches rapides ou pour apprendre.
- **Déclarative :** Vous décrivez l'état désiré dans des fichiers manifestes (YAML) et utilisez `kubectl apply -f <fichier.yaml>`. C'est l'approche recommandée pour la gestion des applications en production car elle permet le versionnement des configurations et facilite l'automatisation (GitOps) [1].

Ce cours se concentrera principalement sur l'approche déclarative.

**Question Ouverte 2.1: Votre Première Interaction**

Avez-vous déjà utilisé `kubectl` ? Si oui, quelles sont les commandes que vous trouvez les plus utiles ou celles qui vous ont posé le plus de questions au début ?

# 3 Objets Fondamentaux de Kubernetes

Maintenant que nous avons une compréhension de l'architecture de Kubernetes et de la manière d'interagir avec le cluster via `kubectl`, il est temps de plonger dans les objets fondamentaux qui constituent le cœur de tout déploiement Kubernetes. Ces objets sont les briques de base que vous utiliserez pour définir, déployer et gérer vos applications.

## §3.1 Pods : L'Unité de Base du Déploiement

**Définition 3.1: Pod**

Un Pod est la plus petite et la plus simple unité déployable dans le modèle d'objet Kubernetes que vous créez ou gérez. Un Pod représente un groupe d'un ou plusieurs conteneurs (comme des conteneurs Docker), avec des ressources de stockage/réseau partagées, et une spécification sur la manière d'exécuter les conteneurs. Les conteneurs au sein d'un même Pod partagent le même contexte réseau (adresse IP, ports), le même espace de noms IPC, et peuvent partager des volumes de stockage. La documentation officielle [35] est une ressource exhaustive sur les Pods.

Les Pods sont typiquement utilisés pour héberger des applications étroitement couplées qui nécessitent de partager des ressources. Par exemple, un conteneur principal d'application web et un conteneur "sidecar" qui exporte les logs de l'application principale vers un système de logging centralisé.

**Pro Tips 3.1: Un Conteneur par Pod : La Règle Générale**

Bien qu'un Pod puisse contenir plusieurs conteneurs, la pratique la plus courante est d'avoir un seul conteneur principal par Pod. Les conteneurs multiples dans un Pod sont généralement réservés aux cas où ils sont très étroitement liés et doivent partager le cycle de vie et les ressources de manière intime (par exemple, les conteneurs sidecar). Pensez à

un Pod comme à un "hôte logique" pour un conteneur [7].

### 3.1.1 Manifeste YAML d'un Pod

Voici un exemple de manifeste YAML pour un Pod simple qui exécute un unique conteneur Nginx.

**Code Block 3.1: Pod Nginx simple avec labels**

```
$
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx-pod-example
5    labels:
6      app: my-nginx-app # Utilis  pour le grouping et la s lection
7      environment: development
8  spec:
9    containers:
10   - name: nginx-webserver # Nom du conteneur dans le Pod
11     image: nginx:1.25.3   # Image Docker    utiliser (version ↩
       ↪ sp cifique)
12     ports:
13     - containerPort: 80   # Port expos  par le conteneur
14       name: http          # Nom du port (optionnel mais recommand ↩
       ↪ )
```

**Listing 3.1 –** pod-nginx-example.yaml

Dans ce manifeste (Listing 3.1) :

- `apiVersion: v1` indique que nous utilisons la version principale de l'API pour les objets Pods.
- `kind: Pod` spécifie le type d'objet.
- `metadata` contient le nom du Pod (`nginx-pod-example`) et des étiquettes (`labels`) que nous aborderons bientôt.
- `spec.containers` est un tableau définissant les conteneurs à exécuter. Ici, un seul conteneur nommé `nginx-webserver`, basé sur l'image `nginx:1.25.3`, qui expose le port 80.

### 3.1.2 Cycle de Vie d'un Pod

Les Pods sont conçus pour être des entités éphémères et non persistantes. Ils ne se "guérissent" pas ou ne se réparent pas d'eux-mêmes en cas de défaillance d'un nœud ou d'arrêt du conteneur (sauf si une politique de redémarrage est configurée au niveau du Pod). C'est pourquoi les Pods sont presque toujours gérés par des contrôleurs de plus haut niveau (comme les Deployments ou StatefulSets) qui s'occupent de leur cycle de vie, de leur réplication et de leur santé.

Un Pod passe par différentes phases durant son existence :

- **Pending :** Le Pod a été accepté par le cluster Kubernetes, mais un ou plusieurs de ses conteneurs ne sont pas encore créés. Cela inclut le temps de téléchargement des images.
- **Running :** Le Pod a été lié à un nœud, et tous ses conteneurs ont été créés. Au moins un conteneur est encore en cours d'exécution, ou est en train de démarrer ou de redémarrer.
- **Succeeded :** Tous les conteneurs du Pod se sont terminés avec succès (code de sortie 0), et ne seront pas redémarrés. Typique pour les Jobs.
- **Failed :** Tous les conteneurs du Pod se sont terminés, et au moins un conteneur s'est terminé en échec (code de sortie non nul).
- **Unknown :** Pour une raison quelconque, l'état du Pod n'a pas pu être obtenu, typiquement en raison d'une erreur de communication avec le nœud où il est censé s'exécuter.

**⚠ Attention 3.1: Les Pods sont du Bétail, pas des Animaux de Compagnie**

Ne vous attachez pas à un Pod spécifique. En raison de leur nature éphémère, vous devriez toujours concevoir vos applications pour qu'elles puissent être remplacées. Les données persistantes doivent être stockées en dehors du Pod, en utilisant des Volumes persistants (abordés plus tard). Cette philosophie est souvent résumée par l'analogie "bétail vs animaux de compagnie" [9].

## §3.2 Labels et Sélecteurs : Organisation et Sélection

**🎓 Définition 3.2: Labels (Étiquettes)**

Les Labels sont des paires clé/valeur qui sont attachées aux objets Kubernetes, tels que les Pods. Ils sont destinés à être utilisés pour spécifier des attributs identifiants des objets qui sont significatifs et pertinents pour les utilisateurs, mais qui n'affectent pas directement la sémantique du système central. Les labels peuvent être utilisés pour organiser et sélectionner des sous-ensembles d'objets. Chaque objet peut avoir un ensemble de labels clé/valeur. Chaque clé doit être unique pour un objet donné [26].

Exemples de labels que vous pourriez utiliser :

- `environment: production`, `environment: development`
- `app: frontend`, `app: backend`
- `tier: cache`, `tier: database`
- `release: stable`, `release: canary`
- `owner: team-alpha`

**Définition 3.3: Sélecteurs (Selectors)**

Les Sélecteurs de labels permettent de sélectionner un groupe d'objets Kubernetes en fonction de leurs labels. Kubernetes supporte deux types de sélecteurs :

- **Sélecteurs basés sur l'égalité (Equality-based) :** Sélectionnent les objets qui ont des clés et des valeurs de labels spécifiques. Par exemple, `environment=production` ou `app!=nginx`.
- **Sélecteurs basés sur les ensembles (Set-based) :** Permettent de filtrer les clés en fonction d'un ensemble de valeurs. Par exemple, `environment in (production, qa)` ou `tier notin (frontend, backend)`.

Les sélecteurs sont fondamentaux dans Kubernetes. Par exemple, un objet Service utilise un sélecteur pour identifier l'ensemble des Pods auxquels il doit router le trafic. Un Deployment utilise un sélecteur pour savoir quels Pods il doit gérer.

**Exemple Pratique 3.1: Utilisation des Labels et Sélecteurs avec `kubectl`**

Supposons que nous ayons plusieurs Pods avec différents labels :

**Code Block 3.2: Exemples de commandes kubectl avec labels**

```
$
# Lister tous les Pods ayant le label 'app=my-nginx-app'
kubectl get pods -l app=my-nginx-app

# Lister tous les Pods dans l'environnement de développement
kubectl get pods -l environment=development

# Lister tous les Pods qui ne sont PAS dans l'environnement de production
kubectl get pods -l 'environment!=production'

# Lister tous les Pods avec le label 'app' (quelle que soit sa valeur)
kubectl get pods -L app

# Ajouter un label à un Pod existant
kubectl label pods nginx-pod-example version=1.0

# Mettre à jour un label (nécessite --overwrite)
kubectl label pods nginx-pod-example environment=staging --overwrite
```

**Listing 3.2 –** Interroger les Pods par labels

## §3.3 Annotations : Métadonnées Non Identifiantes

**Définition 3.4: Annotations**

Les Annotations sont également des paires clé/valeur attachées aux objets, similaires aux labels. Cependant, elles sont conçues pour contenir des métadonnées non identifiantes, souvent plus riches ou plus complexes. Contrairement aux labels, les annotations ne sont pas utilisées pour sélectionner des objets. Elles peuvent être utilisées par des outils et des bibliothèques pour stocker des informations telles que des descriptions, des numéros de build, des pointeurs vers des systèmes de logging/monitoring, des informations de contact, etc. [25, cf. documentation sur les objets].

Exemple d'utilisation dans la section 'metadata' d'un Pod :

**Code Block 3.3: Exemple d'Annotations dans un Pod**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: annotated-pod
  labels:
    app: my-app
  annotations:
    buildVersion: "1.3.4"
    description: "Pod de d monstration avec annotations"
    contact: "dev-team@example.com"
    documentation: "https://docs.example.com/my-app"
spec:
  # ...
```

**Listing 3.3 –** Annotations dans un manifeste de Pod

## §3.4 ReplicaSets : Assurer la Disponibilité des Pods

**Définition 3.5: ReplicaSet**

Un ReplicaSet a pour objectif de maintenir un ensemble stable de répliques de Pods en cours d'exécution à un moment donné. Il garantit qu'un nombre spécifié de Pods identiques (basés sur un template de Pod) s'exécutent. Si un Pod tombe en panne ou est supprimé, le ReplicaSet en créera un nouveau pour le remplacer. S'il y a trop de Pods, il

en supprimera.

Un ReplicaSet est défini par :

- Un nombre de `replicas` désiré.
- Un `selector` de labels pour identifier les Pods qu'il gère.
- Un `template` de Pod utilisé pour créer de nouvelles répliques.

**Pro Tips 3.2: Utilisez les Deployments plutôt que les ReplicaSets directs**

Bien qu'il soit possible de créer des ReplicaSets directement, la pratique recommandée est d'utiliser un objet de plus haut niveau appelé Deployment. Les Deployments gèrent les ReplicaSets pour vous et offrent des fonctionnalités supplémentaires cruciales comme les mises à jour déclaratives (rolling updates) et les rollbacks. Vous interagirez rarement directement avec les ReplicaSets, sauf peut-être pour le débogage.

## §3.5 Deployments : Gestion Déclarative des Mises à Jour

**Définition 3.6: Deployment**

Un Deployment fournit des mises à jour déclaratives pour les Pods et les ReplicaSets. Vous décrivez un état désiré dans un objet Deployment, et le contrôleur de Deployment change l'état actuel vers l'état désiré à une vitesse contrôlée. Vous pouvez définir des Deployments pour créer de nouveaux ReplicaSets, ou pour supprimer des Deployments existants et adopter toutes leurs ressources avec de nouveaux ReplicaSets [20].

Les Deployments sont l'un des objets les plus couramment utilisés pour déployer des applications stateless.

**Exemple Pratique 3.2: Manifeste YAML d'un Deployment**

Voici un exemple de Deployment qui gère trois répliques d'un Pod Nginx :

**Code Block 3.4: Deployment Nginx simple**

```
$
apiVersion: apps/v1 # Notez l'apiVersion pour les Deployments
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3 # Nombre de Pods d sir s
  selector:
    matchLabels: # Doit correspondre aux labels du template de Pod
      app: nginx-frontend
  template: # C'est le template pour cr er les Pods
    metadata:
      labels: # Les Pods  c r   s auront ces labels
        app: nginx-frontend
    spec:
      containers:
      - name: nginx
        image: nginx:1.25.3
        ports:
        - containerPort: 80
```

**Listing 3.4 –** deployment-nginx.yaml

Dans ce Deployment (Listing 3.4) :

- `spec.replicas: 3` indique que nous voulons trois Pods Nginx.
- `spec.selector.matchLabels` définit comment le Deployment trouve les Pods qu'il gère. Il doit correspondre aux labels définis dans `spec.template.metadata.labels`.
- `spec.template` est la spécification du Pod que le Deployment va créer.

Lorsque vous appliquez ce manifeste, le Deployment créera un ReplicaSet, qui à son tour créera trois Pods Nginx.

### 3.5.1 Stratégies de Mise à Jour des Deployments

Les Deployments supportent deux stratégies principales pour mettre à jour les Pods vers une nouvelle version :

- **Recreate :** Tous les Pods existants sont tués avant que les nouveaux ne soient créés. Cela entraîne une interruption de service pendant la mise à jour.
- **RollingUpdate (par défaut) :** La mise à jour se fait de manière progressive. Kubernetes s'assure qu'un certain nombre de Pods de la nouvelle version sont démarrés et prêts avant de

supprimer les anciens Pods, et qu'un certain nombre de Pods (anciens ou nouveaux) restent disponibles pendant la mise à jour pour éviter toute interruption de service. Les paramètres `maxUnavailable` et `maxSurge` permettent de contrôler ce processus.

La stratégie `RollingUpdate` est généralement préférée pour les applications en production. Nous explorerons plus en détail les mises à jour et les rollbacks dans un chapitre ultérieur.

## §3.6 Namespaces : Isolation Logique des Ressources

**Définition 3.7: Namespace**

Les Namespaces fournissent un mécanisme pour partitionner les ressources d'un cluster Kubernetes en groupes logiques isolés. Ils sont destinés à être utilisés dans des environnements avec de nombreux utilisateurs répartis sur plusieurs équipes ou projets. Un nom de ressource doit être unique au sein d'un namespace, mais pas à travers les namespaces. Les namespaces ne fournissent pas une isolation complète au niveau réseau ou sécurité par défaut, mais ils sont une base pour cela [29].

Cas d'usage courants des namespaces :

- Séparer les environnements (par exemple, `development`, `staging`, `production`).
- Isoler les ressources par équipe ou par projet.
- Gérer les quotas de ressources par groupe.

Kubernetes démarre avec quelques namespaces par défaut :

- `default` : Le namespace par défaut pour les objets sans autre namespace spécifié.
- `kube-system` : Pour les objets créés par le système Kubernetes lui-même (Control Plane, addons).
- `kube-public` : Ce namespace est lisible par tous les utilisateurs (y compris non authentifiés). Il est principalement réservé à l'usage du cluster lui-même.
- `kube-node-lease` : Contient des objets Lease pour chaque nœud, utilisés pour la détection de la santé des nœuds.

**Note Importante 3.1: Objets "Namespacés" vs "Non-Namespacés"**

La plupart des ressources Kubernetes (comme les Pods, Deployments, Services) sont "namespacées", c'est-à-dire qu'elles existent à l'intérieur d'un namespace spécifique. Cependant, certaines ressources sont globales au cluster et ne sont pas namespacées, comme les Nodes, les PersistentVolumes, et les Namespaces eux-mêmes.

**Exemple Pratique 3.3: Utilisation des Namespaces avec `kubectl`**

Pour illustrer la gestion des Namespaces en ligne de commande, voici quelques commandes `kubectl` essentielles. Elles permettent de lister, créer, utiliser et supprimer des Namespaces, ainsi que les ressources qu'ils contiennent.

**Code Block 3.5: Commandes kubectl pour les Namespaces**

```
$
# Lister tous les namespaces
kubectl get namespaces
# ou kubectl get ns

# Cr er un nouveau namespace
kubectl create namespace mon-projet-dev

# Cr er un Pod dans un namespace sp cifique
kubectl apply -f ./mon-pod.yaml -n mon-projet-dev
# ou kubectl run mon-pod --image=nginx -n mon-projet-dev (↩
    ↪ imp ratif)

# Lister les Pods dans un namespace sp cifique
kubectl get pods -n mon-projet-dev

# Changer le contexte kubectl pour utiliser un namespace par ↩
    ↪ d faut
kubectl config set-context --current --namespace=mon-projet-dev
# (Attention: cela change votre contexte par d faut pour toutes ↩
    ↪ les commandes kubectl)

# Supprimer un namespace (et toutes les ressources qu'il contient ↩
    ↪ !)
kubectl delete namespace mon-projet-dev
```

**Listing 3.5 –** Gestion des Namespaces

**Attention 3.2: Suppression de Namespace**

La suppression d'un namespace est une opération destructive qui supprime **toutes** les ressources qu'il contient (Pods, Services, Deployments, etc.). Soyez extrêmement prudent lorsque vous supprimez un namespace, surtout en production.

### Question Ouverte 3.1: Organisation de vos Projets

Comment envisagez-vous d'utiliser les labels et les namespaces pour organiser vos propres applications ou projets sur Kubernetes ? Quels seraient les avantages de votre approche ?

# 4 Réseau dans Kubernetes

Le réseau est un aspect fondamental et parfois complexe de Kubernetes. Il permet aux différents composants de vos applications de communiquer entre eux, et d'exposer vos services au monde extérieur. Ce chapitre explore les concepts clés du réseau Kubernetes, notamment les Services, les Ingress et les Network Policies.

## §4.1 Concepts Fondamentaux du Réseau Kubernetes

**Définition 4.1: Modèle Réseau de Kubernetes**

Kubernetes impose un ensemble de conditions fondamentales au réseau :

- **Chaque Pod a sa propre adresse IP unique** au sein du cluster. Il n'y a pas besoin de NAT (Network Address Translation) pour la communication entre Pods.
- Les conteneurs au sein d'un même Pod partagent la même adresse IP et le même espace de ports, et peuvent communiquer entre eux via `localhost`.
- Tous les Pods d'un cluster peuvent communiquer avec tous les autres Pods sans NAT, quel que soit le nœud sur lequel ils s'exécutent.
- Tous les nœuds peuvent communiquer avec tous les Pods (et vice-versa) sans NAT.

Ce modèle simplifie grandement la configuration réseau pour les applications, car elles peuvent être portées depuis des machines virtuelles ou physiques vers des conteneurs sans modification majeure de leur logique réseau [25, cf. section networking].

Ce modèle est généralement implémenté à l'aide d'un plugin CNI (Container Network Interface). Des plugins CNI populaires incluent Calico, Flannel, Cilium, et Weave Net. Le choix du plugin CNI peut influencer les fonctionnalités réseau disponibles, comme le support des Network Policies.

## §4.2 Services : Exposition et Découverte d'Applications

Les Pods sont éphémères : ils peuvent être créés, détruits, et leurs adresses IP peuvent changer. Comment alors accéder de manière fiable à un ensemble de Pods qui fournissent une fonctionnalité donnée ? C'est là qu'interviennent les Services.

**Définition 4.2: Service Kubernetes**

Un Service Kubernetes est une abstraction qui définit un ensemble logique de Pods et une politique permettant d'y accéder. L'ensemble des Pods ciblés par un Service est généralement déterminé par un sélecteur de labels. Un Service fournit une adresse IP stable (appelée ClusterIP) et un nom DNS par lesquels les Pods qu'il cible peuvent être atteints [38].

Lorsqu'un Service est créé, il se voit attribuer une adresse IP virtuelle (la ClusterIP) qui est routable uniquement à l'intérieur du cluster. Lorsqu'un client (un autre Pod, par exemple) envoie une requête à cette ClusterIP, `kube-proxy` sur chaque nœud intercepte cette requête et la redirige vers l'un des Pods "backend" associés à ce Service, en effectuant un équilibrage de charge basique.

### 4.2.1 Types de Services

Kubernetes propose plusieurs types de Services, chacun adapté à un cas d'usage spécifique :

- **ClusterIP (par défaut) :** Expose le Service sur une adresse IP interne au cluster. Ce type rend le Service accessible uniquement depuis l'intérieur du cluster. C'est le type de Service par défaut.
- **NodePort :** Expose le Service sur un port statique (le NodePort) sur l'adresse IP de chaque nœud du cluster. Une requête à `<AdresseIPDuNoeud>:<NodePort>` est routée vers le Service. Cela permet d'accéder au Service depuis l'extérieur du cluster, mais ce n'est généralement pas la méthode recommandée pour une exposition en production (plutôt pour le développement ou des cas spécifiques).
- **LoadBalancer :** Expose le Service à l'extérieur en utilisant un équilibreur de charge du fournisseur de cloud (par exemple, un ELB sur AWS, un Azure Load Balancer, un Google Cloud Load Balancer). Le fournisseur de cloud crée un load balancer externe qui route le trafic vers le Service (généralement via un NodePort créé automatiquement). Ce type est spécifique aux environnements cloud qui le supportent.
- **ExternalName :** Mappe le Service à un nom DNS externe (par exemple, `my.database.example.com`) en retournant un enregistrement CNAME. Aucune redirection de port ou proxy n'est effectuée. Utile pour permettre aux applications dans le cluster d'utiliser un service externe via un nom interne au cluster.

Poulton [9] fournit d'excellents exemples pour chaque type de service.

**Exemple Pratique 4.1: Manifeste YAML d'un Service ClusterIP**

Supposons que nous ayons un Deployment de Pods Nginx avec le label `app: nginx-frontend` (comme dans le Listing 3.4 du chapitre précédent). Nous pouvons créer un Service de type ClusterIP pour les exposer :

**Code Block 4.1: Service ClusterIP pour Pods Nginx**

```
$
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip-service
spec:
  type: ClusterIP # Peut  tre  omis car c'est le type par ↩
   ↪ d faut
  selector:
    app: nginx-frontend # S lectionne les Pods avec ce ↩
   ↪ label
  ports:
  - protocol: TCP
    port: 80          # Port sur lequel le Service  coute  (↩
   ↪ sa ClusterIP)
    targetPort: 80    # Port sur lequel les conteneurs des ↩
   ↪ Pods  coutent
    name: http
```

**Listing 4.1 –** service-nginx-clusterip.yaml

- Une fois ce Service (Listing 4.1) créé, les autres Pods du cluster pourront accéder aux Pods Nginx via le nom DNS `nginx-clusterip-service` (ou `nginx-clusterip-service.<namespace>.svc.cluster.local`) sur le port 80, ou directement via la ClusterIP attribuée.

### 4.2.2 Découverte de Services

Kubernetes offre deux mécanismes principaux pour la découverte de Services :

- **Variables d'Environnement :** Lorsque qu'un Pod est démarré, le `kubelet` injecte des variables d'environnement pour chaque Service actif au moment du démarrage du Pod. Par exemple, pour un Service nommé `mon-service` exposant le port `8080`, des variables comme `MON_SERVICE_SERVICE_HOST` et `MON_SERVICE_SERVICE_PORT` seraient créées. Cette méthode présente des limitations, notamment l'ordre de création (le Service doit exister avant le Pod).
- **DNS (recommandé) :** Kubernetes fournit un service DNS interne (généralement CoreDNS) qui s'exécute dans le cluster. Chaque Service se voit attribuer un enregistrement DNS de la forme `<nom-du-service>.<namespace>.svc.cluster.local`. Les Pods sont configurés pour utiliser ce DNS interne, leur permettant de résoudre les noms de Services en leurs ClusterIPs. C'est la méthode de découverte la plus flexible et la plus couramment utilisée.

## §4.3 Ingress : Routage HTTP/S Avancé

Alors que les Services de type LoadBalancer ou NodePort permettent d'exposer des services à l'extérieur, ils opèrent principalement au niveau 4 (TCP/UDP). Pour un routage plus avancé basé sur les requêtes HTTP/S (niveau 7), comme le routage basé sur l'hôte (virtual hosts) ou le chemin (URL path), Kubernetes utilise un objet appelé Ingress.

**Définition 4.3: Ingress Kubernetes**

Un Ingress est un objet API qui gère l'accès externe aux services d'un cluster, typiquement HTTP et HTTPS. Un Ingress peut fournir un équilibrage de charge, la terminaison SSL/TLS, et un routage basé sur le nom d'hôte ou le chemin. Pour qu'un Ingress fonctionne, le cluster doit avoir un Ingress Controller en cours d'exécution [21].

**Note Importante 4.1: Ingress Controller**

Un Ingress Controller est un composant (souvent un load balancer ou un reverse proxy comme Nginx, HAProxy, Traefik) qui écoute l'API Kubernetes pour les objets Ingress et configure dynamiquement le proxy sous-jacent pour satisfaire les règles définies dans ces objets Ingress. Vous devez déployer un Ingress Controller dans votre cluster pour que les ressources Ingress aient un effet. La documentation du contrôleur Nginx Ingress [8] est un bon point de départ pour l'un des contrôleurs les plus populaires.

**Exemple Pratique 4.2: Manifeste YAML d'un Ingress**

Cet exemple d'Ingress route le trafic externe en fonction du chemin :

**Code Block 4.2: Ingress basé sur le chemin**

```
$
apiVersion: networking.k8s.io/v1 # API pour Ingress
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: / # Annotation spécifique au contrôleur Nginx
spec:
  rules:
  - http:
      paths:
      - path: /foo
        pathType: Prefix # Type de correspondance de chemin
        backend:
          service:
            name: service-foo # Service vers lequel router pour /foo
            port:
              number: 8080
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service-bar # Service vers lequel router pour /bar
            port:
              number: 8090
```

**Listing 4.2 –** ingress-path-based.yaml

Dans cet exemple (Listing 4.2), les requêtes arrivant sur l'adresse IP de l'Ingress Controller avec le chemin `/foo` seraient routées vers le Service `service-foo` sur le port `8080`, tandis que celles avec `/bar` iraient vers `service-bar` sur le port `8090`.

Les Ingress peuvent également gérer la terminaison SSL/TLS en référençant des Secrets Kubernetes contenant les certificats et clés privées.

## §4.4 Network Policies : Sécurisation du Réseau Inter-Pods

Par défaut, dans Kubernetes, tous les Pods peuvent communiquer avec tous les autres Pods au sein du cluster. Pour des raisons de sécurité, il est souvent nécessaire de restreindre cette communication. Les Network Policies permettent de définir des règles de pare-feu au niveau des Pods.

**Définition 4.4: Network Policy**

Une Network Policy est une spécification de la manière dont des groupes de Pods sont autorisés à communiquer entre eux et avec d'autres points de terminaison réseau. Les Network Policies utilisent des sélecteurs de labels pour identifier les Pods auxquels elles s'appliquent et pour définir les règles d'entrée (ingress) et de sortie (egress) [30]. Pour que les Network Policies soient appliquées, votre cluster doit utiliser un plugin réseau (CNI) qui les supporte (par exemple, Calico, Cilium, Weave Net).

Une Network Policy peut spécifier :

- Les Pods auxquels elle s'applique (via `podSelector`).
- Les types de trafic autorisés (`ingress` pour le trafic entrant, `egress` pour le trafic sortant).
- Les sources (pour `ingress`) ou destinations (pour `egress`) autorisées, basées sur :
  - D'autres Pods (via `podSelector`).
  - Des Namespaces (via `namespaceSelector`).
  - Des blocs d'adresses IP (via `ipBlock`).
- Les ports et protocoles concernés.

**Attention 4.1: Politique par Défaut : Deny All**

Si une Network Policy sélectionne un Pod pour le trafic entrant (ingress), ce Pod rejettera tout trafic qui n'est pas explicitement autorisé par au moins une règle `ingress` de cette policy (ou d'une autre policy qui le sélectionne). De même pour le trafic sortant (egress). Si aucune Network Policy ne sélectionne un Pod, tout le trafic entrant et sortant est autorisé par défaut.

**Exemple Pratique 4.3: Exemple de Network Policy**

Cette politique autorise le trafic entrant sur le port TCP 80 vers les Pods ayant le label `app: webserver` uniquement depuis les Pods ayant le label `role: frontend` dans le même namespace :

**Code Block 4.3: Network Policy : Autoriser trafic frontend vers webserver**

```
$
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webserver-allow-frontend
  namespace: default # S'applique aux Pods dans ce namespace
spec:
  podSelector: # S'applique aux Pods avec ce label
    matchLabels:
      app: webserver
  policyTypes:
  - Ingress # Cette politique concerne le trafic entrant
  ingress:
  - from:
    - podSelector: # Autorise le trafic depuis les Pods avec↩
    ↪  ce label
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 80
```

**Listing 4.3 –** networkpolicy-allow-frontend.yaml

Les Network Policies sont un outil puissant pour implémenter une segmentation réseau et améliorer la posture de sécurité de vos applications dans Kubernetes.

**Question Ouverte 4.1: Exposition de vos Applications**

Pour un service web que vous souhaiteriez déployer sur Kubernetes et rendre accessible depuis Internet avec son propre nom de domaine (par exemple, `myapp.example.com`), quelle combinaison d'objets réseau (Services, Ingress) utiliseriez-vous et pourquoi ? Comment assureriez-vous la sécurité des communications ?

# 5 Gestion de la Configuration et des Secrets

La gestion de la configuration d'application et des données sensibles (comme les mots de passe, les clés API, ou les certificats TLS) est un aspect crucial du déploiement d'applications. Kubernetes offre des mécanismes pour découpler ces informations des images de conteneurs, permettant une plus grande flexibilité, sécurité et maintenabilité. Ce chapitre explore les ConfigMaps pour la configuration non sensible et les Secrets pour les données confidentielles.

## §5.1 ConfigMaps : Gestion de la Configuration Applicative

**Définition 5.1: ConfigMap**

Un ConfigMap est un objet API utilisé pour stocker des données de configuration non confidentielles sous forme de paires clé-valeur. Les Pods peuvent consommer les ConfigMaps soit comme des variables d'environnement, soit comme des arguments de ligne de commande, soit comme des fichiers de configuration montés dans un volume. Cela permet de garder les configurations séparées du code applicatif, facilitant les modifications de configuration sans avoir à reconstruire les images de conteneurs [14].

Les ConfigMaps sont idéaux pour stocker des éléments tels que :

- Les URLs de services externes.
- Les paramètres de configuration spécifiques à un environnement (développement, staging, production).
- Des fichiers de configuration entiers (par exemple, `nginx.conf`, `settings.xml`).

**Pro Tips 5.1: Principe de la Configuration Externalisée**

L'utilisation de ConfigMaps s'aligne avec le troisième facteur des "Twelve-Factor App" qui stipule que la configuration doit être stockée dans l'environnement, et non dans le code [48]. Cela rend les applications plus portables et plus faciles à gérer dans différents déploiements.

### 5.1.1 Création de ConfigMaps

Vous pouvez créer des ConfigMaps de plusieurs manières :

1. **À partir de valeurs littérales (via `kubectl`) :**

**Code Block 5.1: Créer un ConfigMap à partir de littéraux**

```
$
kubectl create configmap app-config \
  --from-literal=app.properties.database.url=jdbc:mysql://↩
↪ mydb:3306/prod \
  --from-literal=app.properties.ui.theme=darkblue

```

**Listing 5.1 –** kubectl create configmap –from-literal

Cela crée un ConfigMap nommé `app-config` avec deux clés : `app.properties.database.url` et `app.properties.ui.theme`.

2. **À partir d'un fichier (via `kubectl`) :** Supposez que vous avez un fichier `app-settings.properties` :

**Code Block 5.2: Contenu du fichier app-settings.properties**

```
$
# Fichier de configuration
log_level=INFO
feature_flags.new_dashboard=true

```

**Listing 5.2 –** app-settings.properties

Vous pouvez créer un ConfigMap où chaque ligne du fichier devient une clé :

**Code Block 5.3: Créer un ConfigMap à partir d'un fichier (chaque ligne une clé)**

```
$
kubectl create configmap app-settings-env --from-env-file=↩
```

**Code Block 5.3: Créer un ConfigMap à partir d'un fichier (chaque ligne une clé)**

```
$    ↪ app-settings.properties
2
```

**Listing 5.3 –** kubectl create configmap –from-env-file

Ou, plus couramment, créer un ConfigMap où le contenu entier du fichier est la valeur d'une clé unique :

**Code Block 5.4: Créer un ConfigMap à partir d'un fichier (contenu entier)**

```
$
1      kubectl create configmap app-settings-file --from-file=app↩
      ↪ -settings.properties
2      # ou pour une cl  sp cifique:
3      # kubectl create configmap app-settings-customkey --from-↩
      ↪ file=customkey.props=app-settings.properties
4
```

**Listing 5.4 –** kubectl create configmap –from-file

Dans le premier cas ci-dessus, le ConfigMap `app-settings-file` aura une clé nommée `app-settings.properties` (le nom du fichier) dont la valeur est le contenu entier du fichier.

3. **À partir d'un manifeste YAML :** C'est l'approche déclarative, recommandée pour la gestion en production.

**Code Block 5.5: Définition d'un ConfigMap en YAML**

```
$
1      apiVersion: v1
2      kind: ConfigMap
3      metadata:
4        name: my-app-config-yaml
5        namespace: default
6      data:
7        # Paires cl -valeur
8        database.host: "mysql-service"
9        database.port: "3306"
10       # Contenu de fichier embarqu
11       nginx.conf: |
12         server {
13           listen       80;
14           server_name  localhost;
15
16           location / {
17             root   /usr/share/nginx/html;
18             index  index.html index.htm;
```

**Code Block 5.5: Définition d'un ConfigMap en YAML**

```
$           }
          }
      binaryData: {} # Pour les donn es binaires (encod es en ↩
    ↪ base64)

```

**Listing 5.5 –** configmap-app-config.yaml

Vous appliquez ensuite ce fichier avec `kubectl apply -f configmap-app-config.yaml`.

### 5.1.2 Utilisation des ConfigMaps dans les Pods

Les données d'un ConfigMap peuvent être injectées dans un Pod de plusieurs manières :

1. **Comme variables d'environnement pour un conteneur :**

**Code Block 5.6: Utiliser un ConfigMap comme variables d'environnement**

```
$
      apiVersion: v1
      kind: Pod
      metadata:
        name: pod-configmap-env
      spec:
        containers:
        - name: my-app-container
          image: busybox
          command: [ "/bin/sh", "-c", "env && sleep 3600" ]
          env:
            # D finir une variable d'environnement    partir d'↩
    ↪ une  c l   sp cifique d'un ConfigMap
          - name: DB_HOST
            valueFrom:
              configMapKeyRef:
                name: my-app-config-yaml # Nom du ConfigMap
                key: database.host       # C l   dans le ↩
    ↪ ConfigMap
          - name: DB_PORT
            valueFrom:
              configMapKeyRef:
                name: my-app-config-yaml
                key: database.port
        # Ou injecter toutes les  c l  s  d'un ConfigMap comme ↩
    ↪ variables d'environnement
        # envFrom:
        # - configMapRef:
        #     name: my-app-config-yaml

```

**Listing 5.6 –** pod-with-configmap-env.yaml

2. **Comme arguments de ligne de commande pour un conteneur :** Moins courant, mais possible en utilisant des variables d'environnement (définies à partir d'un ConfigMap) dans la section `command` ou `args` du conteneur.

3. **Comme fichiers dans un volume monté :** C'est la méthode la plus flexible, surtout pour injecter des fichiers de configuration entiers.

**Code Block 5.7: Monter un ConfigMap comme volume**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: pod-configmap-volume
spec:
  containers:
  - name: nginx-container
    image: nginx:1.25.3
    ports:
    - containerPort: 80
    volumeMounts: # O  monter le volume dans le conteneur
    - name: nginx-config-volume
      mountPath: /etc/nginx/conf.d # Le contenu de 'nginx.conf' du ConfigMap sera ici
      readOnly: true
  volumes: # D finition du volume
  - name: nginx-config-volume
    configMap:
      name: my-app-config-yaml # Nom du ConfigMap    utiliser
      items: # Permet de s lectionner quelles c l s monter et sous quel nom de fichier
      - key: nginx.conf # C l   du ConfigMap
        path: custom-nginx.conf # Nom du fichier dans le mountPath
                                # Si 'items' est omis, toutes les c l s sont mont es
                                # avec leurs noms comme noms de fichiers.

```

**Listing 5.7 –** pod-with-configmap-volume.yaml

Dans cet exemple (Listing 5.7), la clé `nginx.conf` du ConfigMap `my-app-config-yaml` (défini dans Listing 5.5) sera montée comme un fichier nommé `custom-nginx.conf` dans le répertoire `/etc/nginx/conf.d/` du conteneur Nginx.

**Note Importante 5.1: Mise à Jour des ConfigMaps et Pods**

Lorsque vous mettez à jour un ConfigMap, les Pods qui le montent en tant que volume verront généralement les fichiers mis à jour dynamiquement (avec un certain délai). Ce-

pendant, les Pods qui utilisent un ConfigMap pour des variables d'environnement ne verront pas les mises à jour à moins d'être redémarrés. Pour des mises à jour propagées de manière fiable, il est souvent nécessaire de déclencher un redémarrage des Pods (par exemple, via un 'kubectl rollout restart deployment/...').

## §5.2 Secrets : Gestion des Données Sensibles

**Définition 5.2: Secret**

Un Secret Kubernetes est un objet utilisé pour stocker et gérer des informations sensibles, telles que des mots de passe, des jetons OAuth, des clés SSH, des certificats TLS, etc. Les stocker dans un Secret est plus sûr que de les mettre en clair dans une définition de Pod ou dans une image de conteneur. Les Secrets peuvent être consommés par les Pods de manière similaire aux ConfigMaps (variables d'environnement ou fichiers dans un volume) [37].

Bien que les données dans les Secrets soient encodées en Base64 par défaut lorsqu'elles sont stockées dans `etcd`, il est crucial de comprendre que **Base64 n'est PAS un mécanisme de chiffrement**. C'est un encodage facilement réversible.

**Attention 5.1: Sécurité des Secrets**

Par défaut, les Secrets sont stockés en Base64 dans `etcd`. Toute personne ayant accès à l'API Kubernetes ou à `etcd` peut potentiellement lire ces Secrets. Pour une sécurité renforcée en production, il est recommandé de :

- Activer le chiffrement au repos (encryption at rest) pour les Secrets dans `etcd`.
- Utiliser des politiques RBAC strictes pour limiter l'accès aux Secrets.
- Intégrer des solutions de gestion de secrets externes (comme HashiCorp Vault, Azure Key Vault, AWS Secrets Manager) via des opérateurs ou des outils comme External Secrets Operator.

Poulton [9] discute de certaines de ces considérations de sécurité.

### 5.2.1 Types de Secrets

Kubernetes propose plusieurs types de Secrets intégrés pour des cas d'usage courants :

- `Opaque` (par défaut) : Utilisé pour stocker des données arbitraires encodées en Base64.
- `kubernetes.io/service-account-token` : Utilisé pour stocker un jeton qui identifie

un ServiceAccount. Géré automatiquement par Kubernetes.

- `kubernetes.io/dockercfg` ou `kubernetes.io/dockerconfigjson` : Utilisé pour stocker les identifiants d'un registre Docker privé afin que `kubelet` puisse télécharger des images privées.
- `kubernetes.io/basic-auth` : Pour les identifiants d'authentification basique.
- `kubernetes.io/ssh-auth` : Pour les données d'authentification SSH.
- `kubernetes.io/tls` : Pour stocker un certificat TLS et sa clé privée, typiquement utilisé par les Ingress pour la terminaison SSL/TLS. Contient les clés `tls.crt` et `tls.key`.
- `bootstrap.kubernetes.io/token` : Pour les jetons de bootstrap lors de la jonction de nouveaux nœuds au cluster.

### 5.2.2 Création de Secrets

Similaire aux ConfigMaps, vous pouvez créer des Secrets de plusieurs manières :

1. **À partir de valeurs littérales (via `kubectl`) :**

**Code Block 5.8: Créer un Secret Opaque à partir de littéraux**

```
$
# Les valeurs seront automatiquement encodées en Base64
kubectl create secret generic db-credentials \
  --from-literal=username=admin \
  --from-literal=password='S3cr3tP@sswOrd!'

```

**Listing 5.8 –** kubectl create secret generic –from-literal

2. **À partir de fichiers (via `kubectl`) :** Supposez un fichier `user.txt` contenant `myuser` et `pass.txt` contenant `mypassword`.

**Code Block 5.9: Créer un Secret Opaque à partir de fichiers**

```
$
kubectl create secret generic app-auth \
  --from-file=./user.txt \
  --from-file=./pass.txt
# Cela créera des clés 'user.txt' et 'pass.txt' dans le Secret.
# Pour des noms de clés personnalisés :
# kubectl create secret generic app-auth-custom \
#   --from-file=USERNAME=./user.txt \
#   --from-file=PASSWORD=./pass.txt

```

**Listing 5.9 –** kubectl create secret generic –from-file

Pour un Secret TLS :

**Code Block 5.10: Créer un Secret TLS à partir de fichiers certificat/clé**

```
$
1       kubectl create secret tls my-app-tls \
2         --cert=/path/to/tls.crt \
3         --key=/path/to/tls.key
4
```

**Listing 5.10 –** kubectl create secret tls

3. **À partir d'un manifeste YAML :** Les valeurs dans le champ `data` doivent être encodées en Base64.

**Code Block 5.11: Définition d'un Secret Opaque en YAML**

```
$
1       apiVersion: v1
2       kind: Secret
3       metadata:
4         name: db-user-pass-yaml
5       type: Opaque # Type de Secret
6       data:
7         # Les valeurs DOIVENT  tre  encod es en Base64
8         # 'echo -n "yourusername" | base64' et 'echo -n "↩
      ↪ yourpassword" | base64'
9         username: eW91cnVzZXJuYW1l # "yourusername" en Base64
10        password: eW91cnBhc3N3b3Jk # "yourpassword" en Base64
11      # Pour des donn es string non-encod es, utilisez '↩
      ↪ stringData' (Kubernetes les encodera pour vous)
12      # stringData:
13      #   username: yourusername
14      #   password: yourpassword
15
```

**Listing 5.11 –** secret-db-creds.yaml

**Pro Tips 5.2: Utilisation de `stringData` dans les manifestes Secret**

Lors de la définition de Secrets en YAML, vous pouvez utiliser le champ `stringData` pour fournir des valeurs en clair. Kubernetes les encodera automatiquement en Base64 lors de la création du Secret. C'est plus pratique et moins sujet aux erreurs que l'encodage manuel. Le champ `stringData` est en écriture seule ; lorsque vous lisez un Secret via l'API, vous verrez toujours le champ `data` avec les valeurs encodées.

### 5.2.3 Utilisation des Secrets dans les Pods

Les Secrets sont consommés par les Pods de la même manière que les ConfigMaps :

1. **Comme variables d'environnement pour un conteneur :**

**Code Block 5.12: Utiliser un Secret comme variables d'environnement**

```
$
      apiVersion: v1
      kind: Pod
      metadata:
        name: pod-secret-env
      spec:
        containers:
        - name: my-secure-app
          image: busybox
          command: [ "/bin/sh", "-c", "env | grep DB_ && sleep ↩
    ↪ 3600" ]
          env:
            - name: DB_USERNAME
              valueFrom:
                secretKeyRef:
                  name: db-user-pass-yaml # Nom du Secret
                  key: username           # Cl  dans le Secret
            - name: DB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: db-user-pass-yaml
                  key: password
          # Ou injecter toutes les cl s d'un Secret comme ↩
    ↪ variables d'environnement
          # envFrom:
          # - secretRef:
          #     name: db-user-pass-yaml

```

**Listing 5.12 –** pod-with-secret-env.yaml

2. **Comme fichiers dans un volume monté (lecture seule) :** C'est souvent la méthode préférée pour les certificats ou les configurations sensibles. Les Secrets montés en volume sont automatiquement mis à jour si le Secret change.

**Code Block 5.13: Monter un Secret comme volume**

```
$
      apiVersion: v1
      kind: Pod
      metadata:
        name: pod-secret-volume
      spec:
        containers:
        - name: my-app-with-tls
```

**Code Block 5.13: Monter un Secret comme volume**

```
$         image: nginx:1.25.3
          volumeMounts:
          - name: tls-certs-volume
            mountPath: "/etc/tls-certs" # Les fichiers du Secret↩
    ↪  seront ici
            readOnly: true
        volumes:
        - name: tls-certs-volume
          secret:
            secretName: my-app-tls # Nom du Secret de type ↩
    ↪ kubernetes.io/tls
            # items: # Optionnel pour s lectionner des c l s ↩
    ↪ sp cifiques
            # - key: tls.crt
            #   path: server.crt
            # - key: tls.key
            #   path: server.key

```

**Listing 5.13 –** pod-with-secret-volume.yaml

Les fichiers dans le volume auront les noms des clés du Secret (par exemple, `tls.crt` et `tls.key` pour un Secret TLS).

3. **Pour télécharger des images depuis des registres privés (`imagePullSecrets`) :** Déclaré au niveau de la `spec` du Pod ou du ServiceAccount.

**Code Block 5.14: Utiliser imagePullSecrets**

```
$
     apiVersion: v1
     kind: Pod
     metadata:
       name: private-reg-pod
     spec:
       containers:
       - name: my-private-container
         image: myprivateregistry.example.com/my-app:1.0
       imagePullSecrets:
       - name: my-registry-key # Nom du Secret de type ↩
    ↪ kubernetes.io/dockerconfigjson

```

**Listing 5.14 –** pod-with-imagepullsecret.yaml

### Question Ouverte 5.1: Gestion des Secrets en Pratique

Quelles sont les données sensibles que vous anticipez devoir gérer pour vos applications sur Kubernetes ? Quelles stratégies (utilisation de types de Secrets spécifiques, chiffrement au repos, intégration avec des coffres-forts externes) considéreriez-vous pour sécuriser ces données en environnement de production ?

# 6 Stockage Persistant dans Kubernetes

Les conteneurs et les Pods Kubernetes sont par nature éphémères. Lorsqu'un Pod est supprimé ou redémarré, toutes les données écrites sur le système de fichiers du conteneur sont perdues, à moins qu'elles ne soient stockées dans un emplacement persistant. Pour les applications stateful, telles que les bases de données, les files d'attente de messages ou tout service nécessitant de conserver son état au-delà du cycle de vie d'un Pod, une solution de stockage persistant est indispensable. Ce chapitre explore les concepts de Volumes, PersistentVolumes, PersistentVolumeClaims et StorageClasses.

## §6.1 Volumes : Stockage dans les Pods

**Définition 6.1: Volume Kubernetes**

Un Volume Kubernetes est essentiellement un répertoire, potentiellement avec des données, qui est accessible aux conteneurs d'un Pod. La manière dont ce répertoire est créé, le support de stockage qui le sous-tend, et son contenu sont déterminés par le type de volume utilisé. Un volume est défini au niveau du Pod et est monté dans des chemins spécifiques à l'intérieur de ses conteneurs. Un volume survit aux redémarrages des conteneurs au sein du Pod, mais sa durée de vie est généralement liée à celle du Pod lui-même (sauf pour les volumes persistants) [44].

Kubernetes supporte de nombreux types de volumes, chacun avec ses propres cas d'usage :

- **`emptyDir` :** Un volume `emptyDir` est créé lorsque le Pod est assigné à un nœud, et existe tant que le Pod s'exécute sur ce nœud. Il est initialement vide. Tous les conteneurs du Pod peuvent lire et écrire les mêmes fichiers dans le volume `emptyDir`, bien qu'il puisse être monté sur des chemins différents dans chaque conteneur. Lorsque le Pod est supprimé du nœud (pour quelque raison que ce soit), les données du volume `emptyDir` sont perdues définitivement. *Cas d'usage :* Espace temporaire pour des opérations de tri, point de contrôle pour une longue tâche de calcul, partage de fichiers entre conteneurs d'un même Pod.
- **`hostPath` :** Un volume `hostPath` monte un fichier ou un répertoire du système de fichiers du nœud hôte directement dans votre Pod.

**⚠ Attention 6.1: Utilisation de `hostPath`**

L'utilisation de `hostPath` doit être faite avec une extrême prudence. Elle introduit des dépendances fortes vis-à-vis de l'environnement du nœud et peut poser des risques de sécurité importants si le Pod a accès à des répertoires sensibles de l'hôte. Son utilisation est généralement limitée à des cas très spécifiques, comme l'accès à des pilotes de périphériques ou à des composants système du nœud (par exemple, par des agents de monitoring ou des DaemonSets système).

- **`configMap` et `secret` :** Ces types de volumes permettent d'exposer les données des ConfigMaps et des Secrets comme des fichiers dans le système de fichiers du conteneur. Nous les avons abordés dans le Chapitre 5.
- **Volumes de stockage réseau (NFS, iSCSI, CephFS, GlusterFS, etc.) :** Kubernetes supporte l'intégration avec divers systèmes de stockage réseau. Ces volumes permettent aux données de persister indépendamment du cycle de vie du Pod et d'être partagées entre plusieurs Pods (selon les capacités du système de stockage).
- **Volumes spécifiques aux fournisseurs de cloud (awsElasticBlockStore, azureDisk, gcePersistentDisk, etc.) :** Ces types de volumes s'intègrent avec les solutions de stockage en mode bloc des fournisseurs de cloud. Les données persistent au-delà du cycle de vie du Pod.

**Exemple Pratique 6.1: Utilisation d'un Volume `emptyDir`**

Pour illustrer comment les conteneurs au sein d'un même Pod peuvent partager des données via un volume `emptyDir`, considérons un scénario avec un conteneur "producteur" qui écrit des données et un conteneur "consommateur" qui les lit.

**Code Block 6.1: Pod avec Volume emptyDir partagé entre deux conteneurs**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: emptydir-example-pod
spec:
  volumes:
  - name: shared-data # Nom du volume d fini au niveau du Pod
    emptyDir: {}    # Type de volume
  containers:
  - name: producer-container
    image: busybox
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          echo "$(date) - Message from producer" >> /data/shared-↩
    ↪ log.txt;
```

**Code Block 6.1: Pod avec Volume emptyDir partagé entre deux conteneurs**

```
$           sleep 5;
          done
      volumeMounts:
      - name: shared-data # R f rence au volume d fini ci-dessus
        mountPath: /data    # Chemin de montage dans ce conteneur
    - name: consumer-container
      image: busybox
      command: ["/bin/sh", "-c", "tail -f /app/shared-log.txt"]
      volumeMounts:
      - name: shared-data
        mountPath: /app     # Chemin de montage diff rent dans ce ↩
    ↪ conteneur
```

**Listing 6.1 –** pod-emptydir-example.yaml

Dans cet exemple, les deux conteneurs partagent le même volume `emptyDir` nommé `shared-data`. Le conteneur `producer-container` écrit dans `/data/shared-log.txt`, et le conteneur `consumer-container` lit depuis `/app/shared-log.txt` (qui pointent vers le même fichier grâce au volume partagé).

## §6.2 PersistentVolumes (PV) et PersistentVolumeClaims (PVC)

Pour gérer le stockage persistant de manière découplée des Pods, Kubernetes introduit deux objets API : PersistentVolume (PV) et PersistentVolumeClaim (PVC). Cette abstraction permet de séparer les préoccupations de l'administrateur du cluster (qui provisionne le stockage) de celles du développeur/utilisateur (qui consomme le stockage) [33].

**Définition 6.2: PersistentVolume (PV)**

Un PersistentVolume (PV) est une unité de stockage dans le cluster qui a été provisionnée par un administrateur ou dynamiquement provisionnée en utilisant des StorageClasses. Il s'agit d'une ressource du cluster, tout comme un nœud est une ressource du cluster. Les PVs ont un cycle de vie indépendant de tout Pod individuel qui utilise le PV. Ils capturent les détails de l'implémentation du stockage (NFS, iSCSI, stockage spécifique au cloud, etc.).

**Définition 6.3: PersistentVolumeClaim (PVC)**

Une PersistentVolumeClaim (PVC) est une demande de stockage faite par un utilisateur. Elle est similaire à un Pod. Les Pods consomment des ressources de nœud et les PVCs consomment des ressources de PV. Les Pods peuvent demander des niveaux spécifiques

de ressources (CPU et mémoire). De même, les PVCs peuvent demander une taille spécifique et des modes d'accès (par exemple, lecture/écriture unique, lecture seule multiple, lecture/écriture multiple).

Le flux typique est le suivant :

1. Un administrateur de cluster crée un certain nombre de PVs avec différentes caractéristiques (taille, performance, type de stockage). Ces PVs sont disponibles pour être consommés.
2. Un utilisateur/développeur crée une PVC demandant une certaine quantité de stockage avec des modes d'accès spécifiques.
3. Le Control Plane de Kubernetes tente de lier (bind) la PVC à un PV disponible qui correspond aux exigences de la PVC.
4. Une fois la PVC liée à un PV, le Pod de l'utilisateur peut monter cette PVC comme un volume, et ainsi accéder au stockage persistant fourni par le PV.

### 6.2.1 Attributs Clés des PVs et PVCs

- **Capacity (Capacité) :** Chaque PV a une capacité de stockage spécifique (par exemple, `5Gi`, `100Mi`). Une PVC demande une capacité.
- **Access Modes (Modes d'Accès) :** Définissent comment le volume peut être monté sur un hôte. Les modes courants sont :
  - `ReadWriteOnce` (`RWO`) : Le volume peut être monté en lecture-écriture par un seul nœud.
  - `ReadOnlyMany` (`ROX`) : Le volume peut être monté en lecture seule par de nombreux nœuds.
  - `ReadWriteMany` (`RWX`) : Le volume peut être monté en lecture-écriture par de nombreux nœuds.
  - `ReadWriteOncePod` (`RWOP`) : (Fonctionnalité Alpha/Beta) Le volume peut être monté en lecture-écriture par un seul Pod.

**⚠ Attention 6.2: Support des Modes d'Accès**

Tous les types de stockage ne supportent pas tous les modes d'accès. Par exemple, un disque EBS d'AWS ne supporte généralement que RWO. Un partage NFS supporte souvent RWX.

- **Volume Modes (Modes de Volume) :** Indique si le volume doit être formaté avec un système de fichiers (`Filesystem`, par défaut) ou utilisé comme un périphérique bloc brut (`Block`).
- **StorageClassName :** Un PV peut appartenir à une "classe" de stockage (StorageClass), qui peut être utilisée pour le provisionnement dynamique et pour aider à lier les PVCs aux PVs appropriés.

- **Reclaim Policy (Politique de Récupération) :** Définit ce qui arrive à un PV après que la PVC à laquelle il était lié a été supprimée. Les options sont :
  - `Retain` : Le PV est conservé (ainsi que ses données). L'administrateur doit manuellement nettoyer le volume et le rendre disponible à nouveau. C'est souvent le choix par défaut pour les données critiques.
  - `Delete` : Le volume sous-jacent (par exemple, le disque EBS, le volume GCE PD) est supprimé.
  - `Recycle` (Déprécié) : Effectue un nettoyage basique du volume (`rm -rf /thevolume/*`) et le rend disponible pour une nouvelle PVC. Cette option est dépréciée en faveur du provisionnement dynamique.

**Exemple Pratique 6.2: Manifestes YAML pour PV et PVC (Provisionnement Statique)**

Voici un exemple de création manuelle d'un PV (par un administrateur) et d'une PVC (par un utilisateur) qui s'y lie.

**Code Block 6.2: PersistentVolume (exemple avec hostPath pour démo locale)**

```
$
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-manual-pv
  labels:
    type: local-storage # Label pour aider au matching
spec:
  storageClassName: manual # Important pour le matching avec la PVC si spécifié
  capacity:
    storage: 2Gi # Taille du volume
  accessModes:
    - ReadWriteOnce # Peut être monté par un seul nœud en R/W
  hostPath: # Pour la démo locale, NE PAS utiliser en production pour des données critiques
    path: "/mnt/data/my-manual-pv-data" # Doit exister sur le nœud
  persistentVolumeReclaimPolicy: Retain # Conserver les données après suppression de la PVC
```

**Listing 6.2 –** pv-hostpath-example.yaml

**Code Block 6.3: PersistentVolumeClaim**

```
$
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-app-pvc
spec:
  storageClassName: manual # Doit correspondre    celui du PV pour un matching statique
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi # Demande de stockage (doit  tre  <=  la  capacit  du PV)
# Si vous voulez un PV sp cifique, vous pouvez ajouter (mais moins flexible):
# volumeName: my-manual-pv
```

**Listing 6.3 –** pvc-example.yaml

Une fois ces deux objets créés, Kubernetes tentera de lier `my-app-pvc` à `my-manual-pv` si les conditions (StorageClass, accessModes, capacity) sont remplies. Ensuite, un Pod peut utiliser `my-app-pvc` :

**Code Block 6.4: Pod utilisant une PVC**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: my-app-with-persistent-storage
spec:
  containers:
  - name: my-frontend
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: my-app-storage
      mountPath: "/usr/share/nginx/html" # Nginx servira les fichiers depuis ce volume
  volumes:
  - name: my-app-storage
    persistentVolumeClaim:
      claimName: my-app-pvc # Référence la PVC créée précédemment
```

**Listing 6.4 –** pod-with-pvc.yaml

## §6.3 StorageClasses : Provisionnement Dynamique

Le provisionnement manuel de PVs peut être fastidieux, surtout dans des environnements cloud où le stockage peut être créé à la demande. Les StorageClasses permettent aux administrateurs de définir différentes "classes" de stockage (par exemple, "rapide-ssd", "lent-hdd-archive", "premium-io") et de permettre le provisionnement dynamique des PVs [40].

**Définition 6.4: StorageClass**

Un objet StorageClass fournit un moyen aux administrateurs de décrire les "classes" de stockage qu'ils offrent. Différentes classes peuvent correspondre à des niveaux de qualité de service, des politiques de sauvegarde, ou des politiques arbitraires déterminées par l'administrateur du cluster. Chaque StorageClass contient les champs `provisioner`, `parameters`, et `reclaimPolicy`, qui sont utilisés lorsque qu'un PersistentVolume appartenant à la classe doit être dynamiquement provisionné.

Lorsqu'une PVC est créée et qu'elle spécifie un `storageClassName` (ou si une StorageClass par défaut est configurée dans le cluster), le provisionneur associé à cette StorageClass créera automatiquement un PV correspondant et le liera à la PVC.

### Exemple Pratique 6.3: Exemple de StorageClass (pour GCE Persistent Disk)

Pour illustrer le provisionnement dynamique, considérons la création d'une StorageClass qui utilise les disques persistants SSD de Google Compute Engine (GCE).

**Code Block 6.5: StorageClass pour GCE PD SSD**

```
$
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-gce-pd # Nom de la classe de stockage
provisioner: kubernetes.io/gce-pd # Provisionneur sp cifique    ↩
    ↪ GCE
parameters:
  type: pd-ssd      # Type de disque: SSD Persistent Disk
  # replication-type: regional-pd # Pour des disques r gionaux (↩
    ↪ plus haute dispo)
reclaimPolicy: Delete # Supprimer le disque GCE PD lorsque la PVC ↩
    ↪ est supprim e
allowVolumeExpansion: true # Permet d'augmenter la taille du volume↩
    ↪  apr s  cr ation
mountOptions:
  - debug
volumeBindingMode: Immediate # Provisionne  d  s  que la PVC est ↩
    ↪  c r   e
# volumeBindingMode: WaitForFirstConsumer # Attend qu'un Pod ↩
    ↪ utilise la PVC pour provisionner (optimise le choix de zone)
```

**Listing 6.5 –** storageclass-gce-ssd.yaml

Une fois cette StorageClass `fast-gce-pd` définie, un utilisateur peut simplement créer une PersistentVolumeClaim (PVC) qui la référence pour demander un stockage SSD provisionné dynamiquement. Par exemple :

**Code Block 6.6: PVC demandant une StorageClass spécifique**

```
$
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-dynamic-ssd-pvc
spec:
  storageClassName: fast-gce-pd # R f rence    la StorageClass
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
```

**Code Block 6.6: PVC demandant une StorageClass spécifique**

```
$       storage: 50Gi # Demande de 50Go de SSD
```

**Listing 6.6 –** pvc-dynamic-ssd.yaml

Lorsqu'un utilisateur applique le manifeste de cette PVC, Kubernetes, via le provisionneur GCE PD spécifié dans la StorageClass, créera automatiquement un disque persistant SSD de 50Go sur Google Cloud. Ensuite, un PersistentVolume (PV) sera créé pour représenter ce disque dans le cluster, et ce PV sera automatiquement lié à la PVC `my-dynamic-ssd-pvc`, la rendant prête à être utilisée par un Pod.

**Pro Tips 6.1: StorageClass par Défaut**

Un cluster Kubernetes peut avoir une StorageClass marquée comme "par défaut". Si une PVC est créée sans spécifier de `storageClassName`, la StorageClass par défaut sera utilisée pour le provisionnement dynamique (si elle existe et supporte le provisionnement dynamique). C'est souvent le cas dans les clusters Kubernetes managés par les fournisseurs de cloud.

**Question Ouverte 6.1: Choix de Stratégie de Stockage**

Pour une base de données PostgreSQL que vous souhaitez déployer en production sur Kubernetes, quelle combinaison de PV, PVC, et StorageClass utiliseriez-vous ? Quelle politique de récupération (‘reclaimPolicy‘) choisiriez-vous pour les PVs et pourquoi ? Quels modes d'accès seraient appropriés ?

# 7 Déploiements Avancés et Gestion du Cycle de Vie

Au-delà du simple déploiement d'applications avec des Deployments et des Services, Kubernetes offre des mécanismes sophistiqués pour gérer la santé de vos applications, optimiser l'utilisation des ressources, et exécuter différents types de charges de travail. Ce chapitre explore les sondes de vivacité et de préparation (health probes), la gestion des requêtes et limites de ressources, ainsi que les Jobs, CronJobs et DaemonSets.

## §7.1 Health Probes : Vérification de Santé des Applications

Pour s'assurer que les applications s'exécutent correctement et sont prêtes à recevoir du trafic, Kubernetes utilise des sondes (probes) configurées au niveau des conteneurs. Ces sondes permettent au `kubelet` de vérifier régulièrement l'état d'un conteneur [16].

**Définition 7.1: Health Probe (Sonde de Santé)**

Une Health Probe est un diagnostic périodique effectué par le `kubelet` sur un conteneur. Kubernetes utilise ces sondes pour déterminer quand redémarrer un conteneur (liveness probe), quand un conteneur est prêt à accepter du trafic (readiness probe), et quand une application a démarré (startup probe).

Il existe trois types principaux de sondes :

- **Liveness Probe (Sonde de Vivacité) :** Indique si le conteneur est en cours d'exécution. Si la sonde de vivacité échoue (par exemple, si l'application est bloquée et ne répond plus), le `kubelet` tue le conteneur, et le conteneur est soumis à sa politique de redémarrage (`restartPolicy`). Si un conteneur n'a pas de sonde de vivacité, l'état par défaut est `Success`. *Objectif :* Redémarrer les conteneurs qui sont dans un état irrécupérable.
- **Readiness Probe (Sonde de Préparation) :** Indique si le conteneur est prêt à servir des requêtes. Si la sonde de préparation échoue, les points de terminaison (endpoints) de ce Pod

sont retirés des Services qui le ciblent. Ainsi, aucun trafic ne lui sera envoyé tant qu'il n'est pas prêt. *Objectif :* Empêcher le trafic d'atteindre des Pods qui ne sont pas encore prêts à le traiter (par exemple, pendant le démarrage, ou lors du chargement de grosses données initiales).

- **Startup Probe (Sonde de Démarrage) :** Indique si l'application à l'intérieur du conteneur a démarré. Toutes les autres sondes sont désactivées si une sonde de démarrage est fournie, jusqu'à ce qu'elle réussisse. Si la sonde de démarrage échoue, le `kubelet` tue le conteneur, et le conteneur est soumis à sa politique de redémarrage. *Objectif :* Utile pour les applications qui ont un temps de démarrage long, pour éviter que les sondes de vivacité ne les tuent prématurément.

Chaque sonde peut être configurée de trois manières pour vérifier l'état :

- **`exec` :** Exécute une commande spécifiée à l'intérieur du conteneur. Le diagnostic est considéré comme réussi si la commande se termine avec un code de sortie 0.
- **`httpGet` :** Effectue une requête HTTP GET contre l'adresse IP du Pod sur un port et un chemin spécifiés. Si la réponse a un code de statut entre 200 et 399 (inclus), le diagnostic est considéré comme réussi.
- **`tcpSocket` :** Tente d'ouvrir un socket TCP sur le port spécifié de l'adresse IP du Pod. Si la connexion peut être établie, le diagnostic est considéré comme réussi.
- **`grpc` :** (Fonctionnalité Beta) Effectue une vérification de santé gRPC.

**Exemple Pratique 7.1: Configuration de Liveness et Readiness Probes**

Pour configurer des sondes de vivacité et de préparation, vous les définissez dans la spécification du conteneur à l'intérieur de votre manifeste de Pod. L'exemple suivant illustre comment configurer des sondes HTTP GET.

**Code Block 7.1: Pod avec Liveness et Readiness Probes HTTP**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: my-app-probed
  labels:
    app: health-check-demo
spec:
  containers:
  - name: my-app-container
    image: my-custom-app:1.0 # Suppose une image avec des endpoints↩
    ↪  /healthz et /readyz
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /healthz # Endpoint pour la vivacit
        port: 8080
      initialDelaySeconds: 15 # Attendre 15s avant la premi re ↩
```

**Code Block 7.1: Pod avec Liveness et Readiness Probes HTTP**

```
$    ↪ sonde
18       periodSeconds: 20      # Sonder toutes les 20s
19       timeoutSeconds: 5      # D lai d'attente pour la r ponse
20       failureThreshold: 3    # Nombre d' checs  avant de consid rer↩
     ↪  la sonde comme   choue
21       successThreshold: 1    # Nombre de succ s pour consid rer la↩
     ↪  sonde comme r ussie apr s un  chec
22     readinessProbe:
23       httpGet:
24         path: /readyz # Endpoint pour la pr paration
25         port: 8080
26       initialDelaySeconds: 5
27       periodSeconds: 10
28       timeoutSeconds: 3
29       failureThreshold: 2
```

**Listing 7.1 –** pod-with-probes.yaml

Dans cet exemple (Listing 7.1), le `kubelet` est configuré pour effectuer des requêtes HTTP GET périodiques. Pour la sonde de vivacité (`livenessProbe`), il interrogera l'endpoint `/healthz` sur le port `8080` du conteneur. Pour la sonde de préparation (`readinessProbe`), c'est l'endpoint `/readyz` qui sera vérifié. Les divers paramètres tels que `initialDelaySeconds` (délai avant la première sonde après le démarrage du conteneur), `periodSeconds` (fréquence des sondes), `timeoutSeconds` (délai d'attente pour une réponse), `failureThreshold` (nombre d'échecs consécutifs pour considérer la sonde comme échouée), et `successThreshold` (nombre de succès consécutifs pour considérer la sonde comme réussie après un échec) permettent d'ajuster finement le comportement de ces vérifications de santé en fonction des caractéristiques de l'application.

**Pro Tips 7.1: Bonnes Pratiques pour les Health Probes**

- Les endpoints des sondes doivent être légers et rapides à répondre.
- La sonde de vivacité doit vérifier l'état interne de l'application, pas seulement si le processus est en cours.
- La sonde de préparation doit vérifier si l'application est prête à traiter des requêtes, incluant la disponibilité des dépendances (bases de données, autres services).
- Ajustez les paramètres (`initialDelaySeconds`, `periodSeconds`, etc.) en fonction des caractéristiques de démarrage et de réponse de votre application.

## §7.2 Gestion des Ressources : Requests et Limits

Lorsque vous définissez un Pod, vous pouvez optionnellement spécifier la quantité de CPU et de mémoire (RAM) que chaque conteneur nécessite. Ces spécifications sont appelées *requests* (demandes) et *limits* (limites) [36].

**Définition 7.2: Requests et Limits**

- **Requests (Demandes) :** La quantité de ressources (CPU, mémoire) qui est garantie d'être allouée au conteneur. Si un conteneur spécifie une demande de CPU, Kubernetes garantira cette quantité de CPU au conteneur. De même pour la mémoire. Le scheduler utilise les demandes pour décider sur quel nœud placer le Pod. Un Pod ne sera planifié que si un nœud a suffisamment de ressources disponibles pour satisfaire les demandes de tous ses conteneurs.
- **Limits (Limites) :** La quantité maximale de ressources qu'un conteneur est autorisé à consommer. Si un conteneur dépasse sa limite de CPU, il peut être étranglé (throttled). S'il dépasse sa limite de mémoire, il sera généralement tué par le système (Out Of Memory - OOM killed).

Les unités pour les ressources sont :

- **CPU :** Mesurée en unités de "CPU Kubernetes". 1 CPU équivaut à 1 vCPU/Core pour les fournisseurs cloud, ou 1 hyperthread sur des processeurs bare-metal. Vous pouvez spécifier des fractions, par exemple `0.5` (un demi CPU) ou `100m` (100 millicpu/millicores, soit 0.1 CPU).
- **Mémoire :** Mesurée en octets. Vous pouvez utiliser des suffixes comme K, M, G, T (kilo, mega, giga, tera) ou leurs équivalents en puissance de deux Ki, Mi, Gi, Ti (kibioctet, mébioctet, gibioctet, tébioctet). Par exemple, `128Mi`, `1Gi`.

**Exemple Pratique 7.2: Configuration des Requests et Limits**

Pour définir les demandes et limites de ressources pour un conteneur, vous utilisez la section `resources` dans la spécification du conteneur. Voici un exemple de Pod configurant ces paramètres.

**Code Block 7.2: Pod avec Requests et Limits de ressources**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: my-app-resources
spec:
  containers:
  - name: my-app-main
    image: my-application:latest
    resources:
      requests: # Ressources garanties
```

**Code Block 7.2: Pod avec Requests et Limits de ressources**

```
$          memory: "64Mi"   # 64 Mbioctets de mmoire
           cpu: "250m"      # 0.25 CPU (250 millicores)
         limits:    # Ressources maximales autorises
           memory: "128Mi" # 128 Mbioctets de mmoire
           cpu: "500m"      # 0.5 CPU (500 millicores)
```

**Listing 7.2 –** pod-with-resources.yaml

Dans ce manifeste YAML, le conteneur `my-app-main` demande (`requests`) 64 Mébioctets de mémoire et 0.25 unité de CPU. Ces ressources lui sont garanties par Kubernetes lors de la planification du Pod. De plus, le conteneur a une limite (`limits`) de 128 Mébioctets de mémoire et de 0.5 unité de CPU, qu'il ne pourra pas dépasser. Si la limite de mémoire est atteinte, le conteneur risque d'être terminé (OOMKilled). Si la limite de CPU est atteinte, son utilisation du CPU sera limitée (throttled).

### 7.2.1 Qualité de Service (QoS) des Pods

En fonction des requests et limits spécifiés pour ses conteneurs, Kubernetes assigne une classe de Qualité de Service (QoS) à chaque Pod. Ces classes influencent la manière dont Kubernetes gère les Pods en cas de pénurie de ressources sur un nœud :

- **Guaranteed :**
  - Tous les conteneurs du Pod doivent avoir des limits de CPU et de mémoire spécifiées.
  - Pour chaque ressource (CPU, mémoire), la request doit être égale à la limit pour tous les conteneurs.

  Ces Pods ont la plus haute priorité et sont les moins susceptibles d'être tués en cas de contention de ressources.

- **Burstable :**
  - Au moins un conteneur du Pod a une request de CPU ou de mémoire, mais ne remplit pas les critères de la classe Guaranteed (par exemple, request < limit, ou certains conteneurs n'ont pas de limits).

  Ces Pods peuvent utiliser plus de ressources que leurs requests si disponibles, jusqu'à leurs limits. Ils sont plus susceptibles d'être tués que les Pods Guaranteed mais moins que les BestEffort.

- **BestEffort :**
  - Aucun conteneur du Pod n'a de requests ou de limits de CPU ou de mémoire spécifiées.

  Ces Pods ont la plus basse priorité et sont les premiers à être tués si le nœud manque de ressources.

**Attention 7.1: Importance de Spécifier Requests et Limits**

Il est fortement recommandé de spécifier les requests et limits pour vos applications en production. Cela aide le scheduler à prendre de meilleures décisions de placement, assure une meilleure stabilité du cluster en évitant qu'un Pod ne consomme toutes les ressources d'un nœud, et permet de définir des priorités via les classes QoS. Omettre ces valeurs conduit à la classe QoS BestEffort, qui est risquée pour les charges de travail critiques.

## §7.3 Jobs et CronJobs : Tâches Batch et Planifiées

Jusqu'à présent, nous avons principalement considéré des applications à longue durée de vie (comme les serveurs web) gérées par des Deployments. Kubernetes supporte également des charges de travail qui s'exécutent jusqu'à leur complétion.

### 7.3.1 Jobs

**Définition 7.3: Job**

Un Job crée un ou plusieurs Pods et s'assure qu'un nombre spécifié d'entre eux se terminent avec succès. Lorsqu'un Job est complété (tous ses Pods se sont terminés avec succès), il enregistre cet état. La suppression d'un Job nettoie les Pods qu'il a créés. Les Jobs sont utiles pour les tâches batch, les opérations ponctuelles, ou les travaux qui doivent s'exécuter jusqu'à leur complétion [22].

**Exemple Pratique 7.3: Manifeste YAML d'un Job**

Pour illustrer la création d'un Job, considérons un exemple simple qui utilise une image Perl pour calculer les décimales de Pi et s'arrêter une fois la tâche accomplie.

**Code Block 7.3: Job simple calculant Pi**

```
$
apiVersion: batch/v1 # Notez l'apiVersion pour les Jobs
kind: Job
metadata:
  name: pi-calculator
spec:
  template: # Template du Pod    ex cuter
    spec:
      containers:
```

**Code Block 7.3: Job simple calculant Pi**

```
$       - name: pi
          image: perl:5.34 # Image Perl
          command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"] # Calcule Pi avec 2000 d cimales
        restartPolicy: Never # Ou OnFailure. 'Always' n'est pas permis pour les Jobs.
    # backoffLimit: 4 # Nombre de tentatives avant de marquer le Job comme   chou   (optionnel)
    # completions: 5 # Ex cuter 5 Pods jusqu'   compl tion (pour des Jobs parall les)
    # parallelism: 2 # Ex cuter 2 Pods en parall le (si completions  > 1)
```

**Listing 7.3 –** job-calculate-pi.yaml

Ce manifeste définit un Job nommé `pi-calculator`. La section `spec.template` décrit le Pod qui sera exécuté par ce Job. Ce Pod contient un unique conteneur nommé `pi`, basé sur l'image `perl:5.34`, qui exécute une commande Perl pour calculer Pi avec 2000 décimales. La politique de redémarrage (`restartPolicy`) est fixée à `Never`, ce qui signifie que si le conteneur échoue, le Pod lui-même ne redémarrera pas le conteneur (le Job pourrait cependant créer un nouveau Pod selon sa configuration de `backoffLimit`). Pour les Jobs, `restartPolicy` ne peut être que `Never` ou `OnFailure`. Des options commentées comme `backoffLimit`, `completions` et `parallelism` permettent de contrôler le nombre de tentatives, le nombre total de Pods à exécuter jusqu'à complétion, et le nombre de Pods à exécuter en parallèle, respectivement.

### 7.3.2 CronJobs

**Définition 7.4: CronJob**

Un CronJob crée des Jobs sur une base de temps récurrente, suivant le format Cron. C'est utile pour des tâches planifiées comme les sauvegardes, la génération de rapports, l'envoi d'e-mails, etc. [18].

Un CronJob est défini par :

- Une `schedule` au format Cron (par exemple, `"0 5 * * *"` pour tous les jours à 5h00).
- Un `jobTemplate` qui spécifie le Job à créer.
- Des politiques optionnelles pour la gestion des exécutions (par exemple, `concurrencyPolicy`, `successfulJobsHistoryLimit`, `failedJobsHistoryLimit`).

**Exemple Pratique 7.4: Manifeste YAML d'un CronJob**

Pour illustrer comment définir une tâche planifiée avec un CronJob, considérons un exemple qui exécute un simple message "Hello" toutes les minutes.

**Code Block 7.4: CronJob exécutant une tâche toutes les minutes**

```
$
apiVersion: batch/v1 # batch/v1beta1 pour les versions plus ↩
    ↪ anciennes de K8s
kind: CronJob
metadata:
  name: hello-cronjob
spec:
  schedule: "*/1 * * * *" # Toutes les minutes
  jobTemplate: # Sp cification du Job    cr er
    spec:
      template:
        spec:
          containers:
          - name: hello-container
            image: busybox:1.28
            args:
            - /bin/sh
            - -c
            - date; echo "Hello from the Kubernetes CronJob"
          restartPolicy: OnFailure
  successfulJobsHistoryLimit: 3 # Conserver l'historique des 3 ↩
    ↪ derniers Jobs r ussis
  failedJobsHistoryLimit: 1     # Conserver l'historique du dernier↩
    ↪  Job   chou
```

**Listing 7.4 –** cronjob-hello.yaml

Ce manifeste définit un CronJob nommé `hello-cronjob`. La clé `spec.schedule` utilise la syntaxe Cron standard et est ici configurée pour exécuter la tâche toutes les minutes (`"*/1 * * * *"`). La section `spec.jobTemplate` contient la spécification du Job qui sera créé à chaque exécution planifiée. Dans cet exemple, le Job exécutera un Pod basé sur l'image `busybox:1.28`, qui affichera la date actuelle suivie d'un message. Les champs `successfulJobsHistoryLimit` et `failedJobsHistoryLimit` contrôlent combien d'instances de Jobs terminés (réussis ou échoués) seront conservées pour l'historique.

## §7.4 DaemonSets : Pods sur Chaque Nœud (ou Certains Nœuds)

**Définition 7.5: DaemonSet**

Un DaemonSet s'assure qu'une copie d'un Pod s'exécute sur tous (ou un sous-ensemble spécifié) les nœuds d'un cluster. Lorsqu'un nœud est ajouté au cluster, un Pod du DaemonSet y est démarré. Lorsqu'un nœud est retiré, le Pod est nettoyé. La suppression d'un DaemonSet nettoie les Pods qu'il a créés [19].

Les DaemonSets sont typiquement utilisés pour :

- Exécuter des agents de collecte de logs sur chaque nœud (par exemple, Fluentd, Logstash).
- Exécuter des agents de monitoring sur chaque nœud (par exemple, Prometheus Node Exporter, Datadog Agent).
- Exécuter des démons de stockage spécifiques au cluster (par exemple, `glusterd`, `ceph`).
- Exécuter des pilotes de périphériques ou des logiciels réseau spécifiques au nœud.

**Code Block 7.5: DaemonSet pour un agent de logging factice**

```
$
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: dummy-log-agent
  labels:
    app: log-agent
spec:
  selector:
    matchLabels:
      name: dummy-log-agent-pod
  template: # Template du Pod    ex cuter sur chaque n ud
    metadata:
      labels:
        name: dummy-log-agent-pod
    spec:
      # tolerations: # Pour s'assurer qu'il s'ex cute  m me sur ↩
    ↪ les n uds Control Plane si besoin
      # - key: node-role.kubernetes.io/control-plane
      #   operator: Exists
      #   effect: NoSchedule
      containers:
      - name: log-collector
        image: busybox:1.28
        args:
        - /bin/sh
        - -c
        - >
          while true; do
            echo "Simulating log collection on $(hostname) at $(↩
    ↪ date)";
            sleep 60;
          done
        # resources: # Il est bon de sp cifier des ressources pour↩
    ↪  les DaemonSets
          #   requests:
          #     cpu: "50m"
          #     memory: "50Mi"
          #   limits:
          #     cpu: "100m"
          #     memory: "100Mi"
```

**Listing 7.5 –** daemonset-log-agent.yaml

Ce DaemonSet déploiera un Pod `dummy-log-agent-pod` sur chaque nœud éligible du clus-

ter. Vous pouvez utiliser un `nodeSelector` ou des affinités/anti-affinités de nœud pour restreindre les DaemonSets à un sous-ensemble de nœuds.

**Question Ouverte 7.1: Choix du Type de Charge de Travail**

Pour les scénarios suivants, quel contrôleur Kubernetes (Deployment, Job, CronJob, DaemonSet, ou peut-être un StatefulSet que nous n'avons pas encore vu) serait le plus approprié et pourquoi ?

- Un service API REST stateless qui doit être hautement disponible.
- Une tâche de migration de base de données à exécuter une seule fois.
- Un processus de nettoyage de fichiers temporaires à exécuter tous les soirs à 2h du matin.
- Un agent de sécurité qui doit s'exécuter sur chaque machine du cluster pour surveiller l'activité.

# 8 Observabilité et Maintenance dans Kubernetes

Déployer des applications sur Kubernetes n'est que la première étape. Pour assurer leur bon fonctionnement, leur performance et leur fiabilité, il est essentiel de mettre en place des stratégies d'observabilité (logging, monitoring, tracing) et de comprendre les opérations de maintenance du cluster. Ce chapitre aborde les bases du logging et du monitoring dans Kubernetes, les techniques de débogage courantes, et la maintenance des nœuds.

## §8.1 Logging dans Kubernetes

**Définition 8.1: Logging Applicatif**

Le logging applicatif consiste à enregistrer les événements et les messages générés par une application pendant son exécution. Ces logs sont cruciaux pour le débogage, l'audit, et la compréhension du comportement de l'application. Dans un environnement conteneurisé comme Kubernetes, les applications écrivent généralement leurs logs sur la sortie standard (`stdout`) et l'erreur standard (`stderr`).

Le runtime de conteneur (par exemple, Docker, containerd) capture ces flux et les redirige vers un pilote de logging. Kubernetes fournit ensuite des mécanismes pour y accéder.

### 8.1.1 Accès aux Logs des Pods

La commande la plus simple pour accéder aux logs d'un conteneur dans un Pod est `kubectl logs` :

**Code Block 8.1: Accéder aux logs d'un Pod**

```
$
# Afficher les logs du conteneur 'my-container' dans le Pod 'my-pod'
kubectl logs my-pod -c my-container

# Si le Pod n'a qu'un seul conteneur, -c est optionnel
kubectl logs my-pod

# Afficher les logs en continu (comme tail -f)
kubectl logs -f my-pod -c my-container

# Afficher les logs des N derni res lignes
kubectl logs --tail=50 my-pod

# Afficher les logs du conteneur pr c dent (s'il a   t   red marr )
kubectl logs --previous my-pod -c my-container
```

**Listing 8.1 –** Commandes kubectl logs

Ces logs sont stockés sur le nœud où s'exécute le Pod. Si le Pod est supprimé ou si le nœud tombe en panne, ces logs peuvent être perdus à moins qu'une solution de logging centralisée ne soit en place.

### 8.1.2 Architecture de Logging Centralisée

Pour une gestion robuste des logs en production, en particulier pour l'agrégation, la recherche, l'analyse et l'archivage à long terme, une solution de logging centralisée est indispensable. L'architecture typique implique [27] :

1. **Un agent de logging sur chaque nœud :** Un conteneur (souvent déployé via un DaemonSet) s'exécute sur chaque nœud. Cet agent collecte les logs des conteneurs s'exécutant sur ce nœud (généralement en lisant les fichiers de logs écrits par le runtime de conteneur ou en capturant `stdout`/`stderr`). Des agents populaires incluent Fluentd, Fluent Bit, et Logstash.
2. **Un backend de logging :** Les agents envoient les logs collectés vers un backend centralisé pour stockage et indexation. Des backends courants sont Elasticsearch, Loki, Splunk, ou des services cloud comme Google Cloud Logging, AWS CloudWatch Logs.
3. **Un outil de visualisation et de requête :** Une interface utilisateur (comme Kibana pour Elasticsearch, Grafana pour Loki) permet de rechercher, visualiser et analyser les logs agrégés.

**Pro Tips 8.1: La Stack EFK/ELK ou PLG**

Des combinaisons populaires pour le logging centralisé sont :

- **EFK Stack :** Elasticsearch (stockage/indexation), Fluentd (collecte/expédition), Kibana (visualisation).
- **ELK Stack :** Similaire à EFK, mais utilise Logstash au lieu de Fluentd (Logstash est

plus gourmand en ressources mais offre plus de capacités de transformation).

- **PLG Stack :** Promtail (collecte/expédition, spécifique à Loki), Loki (stockage/indexation optimisé pour les logs et les labels Prometheus), Grafana (visualisation). Cette stack est souvent appréciée pour sa simplicité et son intégration avec Prometheus.

Des tutoriels comme celui de DigitalOcean [3] expliquent comment mettre en place de telles stacks.

## §8.2 Monitoring dans Kubernetes

**Définition 8.2: Monitoring**

Le monitoring consiste à collecter, traiter, agréger et afficher des données quantitatives en temps réel sur un système, telles que l'utilisation du CPU, de la mémoire, la latence des requêtes, les taux d'erreur, etc. Le monitoring est essentiel pour comprendre la performance, la disponibilité, et la santé globale du cluster et des applications.

### 8.2.1 Pipeline de Métriques de Base

Kubernetes fournit un pipeline de métriques de base qui permet de collecter des informations sur l'utilisation des ressources des Pods et des Nœuds [42] :

- **cAdvisor :** Un agent intégré au `kubelet` sur chaque nœud, qui collecte des métriques sur les conteneurs.
- **Metrics Server :** Un agrégateur de métriques à l'échelle du cluster. Il collecte les métriques de cAdvisor via l'API Summary du `kubelet` et les expose via l'API de Métriques de Kubernetes (`metrics.k8s.io`). Il est nécessaire pour des commandes comme `kubectl top node` et `kubectl top pod`, ainsi que pour le Horizontal Pod Autoscaler (HPA).

Metrics Server n'est pas toujours installé par défaut sur tous les clusters, mais il est facile à déployer.

**Code Block 8.2: Utiliser kubectl top (nécessite Metrics Server)**

```
$
# Afficher l'utilisation des ressources des n uds
kubectl top nodes

# Afficher l'utilisation des ressources des Pods dans le namespace courant
kubectl top pods

# Afficher l'utilisation des ressources des Pods dans tous les namespaces
```

**Code Block 8.2: Utiliser kubectl top (nécessite Metrics Server)**

```
$kubectl top pods --all-namespaces
```

**Listing 8.2 –** Commandes kubectl top

### 8.2.2 Monitoring Avancé avec Prometheus et Grafana

Pour un monitoring plus complet et personnalisable, la combinaison de Prometheus et Grafana est devenue un standard de facto dans l'écosystème Kubernetes.

- **Prometheus [47] :** Un système de monitoring et d'alerte open-source. Il collecte des métriques à partir de cibles configurées (scraping d'endpoints HTTP exposant des métriques au format Prometheus), les stocke dans une base de données de séries temporelles, et permet des requêtes puissantes avec son langage PromQL. De nombreux composants Kubernetes et applications exposent nativement des métriques au format Prometheus.
- **Grafana [6] :** Une plateforme open-source pour la visualisation et l'analyse de métriques. Elle permet de créer des tableaux de bord interactifs et personnalisables en se connectant à diverses sources de données, y compris Prometheus.

**Note Importante 8.1: Déploiement de Prometheus et Grafana**

Prometheus et Grafana peuvent être déployés sur Kubernetes en utilisant des manifestes YAML, des opérateurs (comme le Prometheus Operator), ou des Charts Helm. Ils offrent une visibilité détaillée sur la santé du cluster, des nœuds, des Pods, des services Kubernetes, ainsi que des métriques applicatives personnalisées si vos applications les exposent.

## §8.3 Debugging des Applications et du Cluster

Malgré tous nos efforts, des problèmes surviendront. Kubernetes fournit plusieurs outils et techniques pour le débogage.

- **`kubectl describe <type> <nom>` :** C'est souvent la première commande à utiliser lorsqu'un objet ne se comporte pas comme prévu. Elle fournit une vue détaillée de l'objet, y compris sa configuration, son statut, les événements récents qui lui sont associés, et les conditions (par exemple, pourquoi un Pod est en attente).

**Code Block 8.3: Utiliser kubectl describe**

```
$
# D crire un Pod pour voir ses   vnements   et son  tat
kubectl describe pod my-problematic-pod

# D crire un Deployment pour comprendre pourquoi les ↩
↪ r pliques ne sont pas pr tes
```

**Code Block 8.3: Utiliser kubectl describe**

```
$ kubectl describe deployment my-app-deployment
```

**Listing 8.3 –** Exemple de kubectl describe

- **`kubectl get events -sort-by=.metadata.creationTimestamp` :** Affiche les événements du cluster (ou d'un namespace spécifique si −n est utilisé) triés par heure. Très utile pour voir ce qui s'est passé récemment.
- **`kubectl exec -it <nom-du-pod> - <commande>` :** Permet d'exécuter une commande à l'intérieur d'un conteneur en cours d'exécution. Utile pour inspecter l'environnement du conteneur, vérifier des fichiers, ou exécuter des outils de diagnostic.

**Code Block 8.4: Utiliser kubectl exec**

```
$
# Ouvrir un shell interactif dans un conteneur
kubectl exec -it my-pod -c my-container -- /bin/sh

# Ex cuter une commande non interactive
kubectl exec my-pod -c my-container -- ls /app
```

**Listing 8.4 –** Exemple de kubectl exec

- **`kubectl port-forward <type>/<nom> <port-local>:<port-distant>` :** Permet de transférer le trafic d'un port local de votre machine vers un port d'un Pod ou d'un Service. Utile pour accéder directement à une application en cours de débogage sans l'exposer via un Service de type NodePort ou LoadBalancer.

**Code Block 8.5: Utiliser kubectl port-forward**

```
$
# Transf rer le port local 8080 vers le port 80 du Pod '↩
↪ my-web-pod'
kubectl port-forward pod/my-web-pod 8080:80

# Transf rer vers un Service
kubectl port-forward service/my-service 9000:http
# ( o  'http' est le nom du port du Service)
```

**Listing 8.5 –** Exemple de kubectl port-forward

- **Vérification des sondes (Probes) :** Si un Pod est redémarré en boucle ou n'accepte pas de trafic, vérifiez la configuration et le fonctionnement de ses Liveness et Readiness Probes.

Les messages d'échec des sondes apparaissent dans les événements du Pod (`kubectl describe pod ...`).

- **Vérification des ressources :** Si un Pod est tué de manière inattendue (OOMKilled) ou si les performances sont médiocres, vérifiez ses requests et limits de CPU/mémoire, et l'utilisation réelle des ressources sur le nœud (`kubectl top pod`, `kubectl top node`).

Poulton [9] propose également un chapitre dédié au troubleshooting.

## §8.4 Maintenance des Nœuds

Parfois, il est nécessaire d'effectuer des opérations de maintenance sur les nœuds d'un cluster Kubernetes (par exemple, mise à jour du noyau, maintenance matérielle). Kubernetes fournit des mécanismes pour gérer cela de manière contrôlée afin de minimiser l'impact sur les applications.

- **`kubectl cordon <nom-du-noeud>` :** Marque un nœud comme "non planifiable" (`unschedulable`). Cela signifie que le scheduler n'y placera plus de nouveaux Pods. Les Pods existants sur le nœud continuent de s'exécuter. C'est la première étape avant une maintenance.
- **`kubectl drain <nom-du-noeud> [options]` :** Évacue en toute sécurité tous les Pods (ou une sélection) d'un nœud. `drain` respecte les PodDisruptionBudgets (que nous n'avons pas encore couverts) et s'assure que les Pods gérés par des contrôleurs (Deployments, StatefulSets) sont proprement terminés et recréés sur d'autres nœuds. Options courantes :
  - `--ignore-daemonsets` : Ne pas essayer de supprimer les Pods gérés par des DaemonSets (car ils s'exécutent sur tous les nœuds).
  - `--delete-emptydir-data` (ou `--delete-local-data` dans les anciennes versions) : Supprimer les Pods qui utilisent des volumes `emptyDir` (car leurs données seraient perdues de toute façon).
  - `--force` : Forcer la suppression des Pods non gérés par un contrôleur de réplication.

  Après un `drain` réussi (et une fois la maintenance terminée sur le nœud), vous pouvez le réintégrer dans le cluster.
- **`kubectl uncordon <nom-du-noeud>` :** Marque à nouveau un nœud comme "planifiable", permettant au scheduler d'y placer de nouveaux Pods. C'est l'étape pour réintroduire un nœud après maintenance.

**Code Block 8.6: Commandes pour la maintenance d'un nœud**

```
$
# 1. Marquer le n ud comme non planifiable
kubectl cordon my-node-to-maintain

# 2.  vacuer  les Pods du n ud
kubectl drain my-node-to-maintain --ignore-daemonsets

# --- Effectuer la maintenance sur my-node-to-maintain ---
# (Mise    jour, red marrage, etc.)

# 3. R int grer le n ud dans le cluster
```

**Code Block 8.6: Commandes pour la maintenance d'un nœud**

```
$kubectl uncordon my-node-to-maintain
```

**Listing 8.6 –** Maintenance d'un nœud

**Question Ouverte 8.1: Votre Stratégie d'Observabilité**

Pour une application web critique que vous gérez, quels types de métriques (système, applicatives) collecteriez-vous ? Quels types de logs seraient essentiels ? Comment configureriez-vous les alertes pour être notifié des problèmes importants ?

# 9 Sécurité dans Kubernetes

La sécurité est une préoccupation majeure dans tout système distribué, et Kubernetes ne fait pas exception. Un cluster Kubernetes mal sécurisé peut exposer des données sensibles, permettre des accès non autorisés, ou être utilisé pour des activités malveillantes. Ce chapitre introduit les concepts fondamentaux de la sécurité dans Kubernetes, couvrant l'authentification, l'autorisation (RBAC), les ServiceAccounts, les contextes de sécurité et l'admission de la sécurité des Pods. Un aperçu général de la sécurité cloud native peut être trouvé dans la documentation officielle [32].

## §9.1 Authentification et Autorisation : Les Piliers de l'Accès

L'accès à l'API Kubernetes est contrôlé par une série d'étapes :

1. **Authentification (AuthN) :** Qui fait la requête ? Kubernetes doit vérifier l'identité de l'utilisateur (humain ou processus) qui tente d'effectuer une action.
2. **Autorisation (AuthZ) :** La requête est-elle permise ? Une fois l'identité vérifiée, Kubernetes détermine si l'utilisateur authentifié a le droit d'effectuer l'action demandée sur la ressource spécifiée.
3. **Contrôle d'Admission (Admission Control) :** La requête est-elle valide et conforme aux politiques du cluster ? Avant de persister un objet dans `etcd`, les contrôleurs d'admission peuvent intercepter la requête pour la valider, la modifier, ou la rejeter.

### 9.1.1 Authentification (AuthN)

Kubernetes ne gère pas directement les comptes utilisateurs. Il s'appuie sur des mécanismes externes pour l'authentification. Plusieurs stratégies d'authentification peuvent être configurées pour l'API server [17] :

- **Certificats Clients X.509 :** Les utilisateurs peuvent être authentifiés via des certificats clients signés par une autorité de certification (CA) de confiance du cluster. C'est une méthode courante pour les administrateurs et les composants du système.
- **Jetons Statiques (Static Token File) :** Un fichier CSV contenant des jetons et des informations utilisateur peut être fourni à l'API server. Peu flexible et déconseillé pour les environnements de production importants.

- **Jetons de Compte de Service (Service Account Tokens) :** Les ServiceAccounts (discutés plus loin) sont des identités pour les processus s'exécutant dans les Pods. Ils utilisent des jetons JWT (JSON Web Tokens) pour s'authentifier auprès de l'API server.
- **OpenID Connect (OIDC) Tokens :** Permet d'intégrer Kubernetes avec des fournisseurs d'identité externes (comme Google, Azure AD, Keycloak, Okta) qui supportent OIDC. C'est une méthode populaire pour gérer l'authentification des utilisateurs humains.
- **Webhook Token Authentication :** Permet de déléguer l'authentification à un service externe via un webhook.
- **Authenticating Proxy :** L'API server peut être configuré pour faire confiance à l'identité fournie par un proxy d'authentification placé devant lui.

Une fois authentifié, l'utilisateur se voit attribuer un nom d'utilisateur (string) et optionnellement une liste de groupes. Ces informations sont ensuite utilisées par le module d'autorisation.

### 9.1.2 Autorisation (AuthZ)

Une fois l'utilisateur authentifié, le module d'autorisation détermine s'il est autorisé à effectuer l'action demandée. Kubernetes supporte plusieurs modules d'autorisation, le plus courant et recommandé étant RBAC (Role-Based Access Control) [43]. D'autres incluent ABAC (Attribute-Based Access Control, moins utilisé), Node, et Webhook.

## §9.2 RBAC : Contrôle d'Accès Basé sur les Rôles

**Définition 9.1: RBAC (Role-Based Access Control)**

RBAC est une méthode de régulation de l'accès aux ressources informatiques ou réseau basée sur les rôles des utilisateurs individuels au sein d'une organisation. Dans Kubernetes, RBAC permet aux administrateurs de définir de manière granulaire qui peut effectuer quelles actions (verbes) sur quelles ressources (types d'objets API) dans quels namespaces (ou à l'échelle du cluster).

RBAC repose sur quatre types d'objets principaux :

- **Role :** Définit un ensemble de permissions (règles) au sein d'un namespace spécifique. Un Role contient des règles qui représentent un ensemble de permissions. Les permissions sont purement additives (il n'y a pas de règles de "refus").
- **ClusterRole :** Similaire à un Role, mais les permissions définies sont valides à l'échelle du cluster (non-namespacées). Peut être utilisé pour accorder des permissions sur des ressources de cluster (comme les Nodes), des ressources namespacées dans tous les namespaces, ou des endpoints non-ressources (comme `/healthz`).
- **RoleBinding :** Lie un Role à un sujet (utilisateur, groupe, ou ServiceAccount), lui accordant les permissions définies dans ce Role au sein du namespace du RoleBinding.
- **ClusterRoleBinding :** Lie un ClusterRole à un sujet, lui accordant les permissions définies dans ce ClusterRole à l'échelle du cluster.

Une règle dans un Role ou ClusterRole spécifie :

- `apiGroups` : Le groupe d'API de la ressource (par exemple, `""` pour le core API group, `apps`, `batch`).
- `resources` : Le type de ressource (par exemple, `pods`, `deployments`, `services`).
- `verbs` : Les actions autorisées (par exemple, `get`, `list`, `watch`, `create`, `update`, `patch`, `delete`).

**Exemple Pratique 9.1: Exemple de Role et RoleBinding**

Illustrons avec un exemple concret. Le Role suivant accorde des permissions de lecture sur les Pods dans le namespace "default".

**Code Block 9.1: Role pour lire les Pods**

```
$
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indique le core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

**Listing 9.1 –** pod-reader-role.yaml

Ensuite, ce RoleBinding accorde le Role `pod-reader` à l'utilisateur `jane` dans le namespace "default".

**Code Block 9.2: RoleBinding pour l'utilisateur Jane**

```
$
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-jane
  namespace: default
subjects: # La liste des sujets auxquels lier le r le
- kind: User # Peut  tre  User, Group, ou ServiceAccount
  name: jane # Le nom de l'utilisateur (sensible    la casse)
  apiGroup: rbac.authorization.k8s.io
roleRef: # R f rence au Role (ou ClusterRole)    lier
  kind: Role # Doit  tre  Role pour un RoleBinding
  name: pod-reader # Nom du Role
  apiGroup: rbac.authorization.k8s.io
```

**Listing 9.2 –** jane-pod-reader-binding.yaml

Avec cette configuration, l'utilisateur "jane" pourra lister et lire les Pods dans le namespace "default", mais ne pourra pas les créer ou les supprimer.

**Pro Tips 9.1: Principe du Moindre Privilège avec RBAC**

Lors de la configuration de RBAC, appliquez toujours le principe du moindre privilège : n'accordez que les permissions strictement nécessaires à un utilisateur ou à un processus pour accomplir ses tâches. Évitez d'utiliser des ClusterRoleBindings avec des permissions étendues (comme `cluster-admin`) pour les utilisateurs ou ServiceAccounts quotidiens.

## §9.3 ServiceAccounts : Identités pour les Processus dans les Pods

**Définition 9.2: ServiceAccount**

Un ServiceAccount fournit une identité pour les processus qui s'exécutent à l'intérieur d'un Pod. Lorsqu'un Pod est créé, s'il ne spécifie pas de ServiceAccount, il se voit attribuer le ServiceAccount `default` du namespace. Un jeton d'authentification (JWT) associé au ServiceAccount est automatiquement monté dans le Pod (par défaut à `/var/run/secrets/kubernetes.io/serviceaccount/token`), permettant aux processus du Pod de s'authentifier auprès de l'API server [39].

Les ServiceAccounts sont des sujets valides pour les RoleBindings et ClusterRoleBindings. C'est ainsi que vous accordez des permissions spécifiques à vos applications pour interagir avec l'API Kubernetes.

**Exemple Pratique 9.2: Utilisation d'un ServiceAccount Personnalisé**

Supposons qu'une application dans un Pod ait besoin de lister les ConfigMaps de son namespace. Les étapes seraient les suivantes :

1. **Créer un ServiceAccount :**

**Code Block 9.3: Créer un ServiceAccount**

```
$
1       kubectl create serviceaccount my-app-sa -n my-namespace
2
```

**Listing 9.3 –** Création d'un ServiceAccount

2. **Créer un Role** qui accorde la permission de lister les ConfigMaps :

**Code Block 9.4: Role pour lister les ConfigMaps**

```
$
1       apiVersion: rbac.authorization.k8s.io/v1
2       kind: Role
3       metadata:
4         namespace: my-namespace
5         name: configmap-lister
6       rules:
7       - apiGroups: [""]
8         resources: ["configmaps"]
9         verbs: ["get", "list"]
10
```

**Listing 9.4 –** configmap-lister-role.yaml

3. **Lier le Role au ServiceAccount** avec un RoleBinding :

**Code Block 9.5: RoleBinding pour le ServiceAccount**

```
$
1       apiVersion: rbac.authorization.k8s.io/v1
2       kind: RoleBinding
3       metadata:
4         name: my-app-sa-can-list-configmaps
5         namespace: my-namespace
6       subjects:
7       - kind: ServiceAccount
8         name: my-app-sa # Nom du ServiceAccount
9         namespace: my-namespace
10      roleRef:
11        kind: Role
12        name: configmap-lister
13        apiGroup: rbac.authorization.k8s.io
14
```

**Listing 9.5 –** configmap-lister-binding.yaml

4. **Configurer le Pod** pour utiliser ce ServiceAccount :

**Code Block 9.6: Pod utilisant un ServiceAccount spécifique**

```
$
1       apiVersion: v1
2       kind: Pod
3       metadata:
4         name: my-app-pod
```

**Code Block 9.6: Pod utilisant un ServiceAccount spécifique**

```
$       namespace: my-namespace
      spec:
        serviceAccountName: my-app-sa # Sp cifier le ↩
    ↪ ServiceAccount
        containers:
        - name: my-app-container
          image: my-app-image
          # ...

```

**Listing 9.6 –** pod-with-custom-sa.yaml

L'application s'exécutant dans `my-app-pod` pourra maintenant utiliser son jeton ServiceAccount pour lister les ConfigMaps dans le namespace `my-namespace`.

**Attention 9.1: Désactiver l'Automontage du Jeton ServiceAccount**

Si un Pod n'a pas besoin d'accéder à l'API Kubernetes, il est recommandé de désactiver l'automontage du jeton ServiceAccount pour réduire la surface d'attaque. Cela peut être fait en mettant `automountServiceAccountToken: false` dans la `spec` du Pod ou dans la définition du ServiceAccount lui-même.

## §9.4 Security Contexts : Contrôle des Privilèges des Pods et Conteneurs

**Définition 9.3: Security Context (Contexte de Sécurité)**

Un Security Context définit les privilèges et les contrôles d'accès pour un Pod ou un Conteneur. Il permet de spécifier des paramètres tels que l'UID/GID sous lequel le processus s'exécute, les capabilities Linux, le mode SELinux, AppArmor, seccomp, et si le conteneur peut s'exécuter en mode privilégié [15].

Les contextes de sécurité peuvent être définis à deux niveaux :

- **Au niveau du Pod (`spec.securityContext`) :** S'applique à tous les conteneurs du Pod.
- **Au niveau du Conteneur (`spec.containers[].securityContext`) :** S'applique uniquement à ce conteneur spécifique et peut surcharger les paramètres définis au niveau du Pod.

Quelques paramètres courants de Security Context :

- `runAsUser` / `runAsGroup` : Spécifie l'UID / GID sous lequel les processus du conteneur s'exécutent.
- `runAsNonRoot: true` : Exige que le conteneur s'exécute avec un utilisateur non-root. Si non spécifié ou `false`, le conteneur peut s'exécuter en tant que root si l'image est configurée ainsi.
- `fsGroup` : Spécifie un GID supplémentaire qui sera propriétaire des volumes montés et auquel tous les processus du conteneur appartiendront.
- `privileged: true` : Exécute le conteneur en mode privilégié, lui donnant accès à tous les périphériques de l'hôte. À utiliser avec une extrême prudence.
- `capabilities` : Permet d'ajouter ou de supprimer des capabilities Linux spécifiques (par exemple, `NET_ADMIN`, `SYS_TIME`).
- `readOnlyRootFilesystem: true` : Monte le système de fichiers racine du conteneur en lecture seule.
- `allowPrivilegeEscalation: false` : Empêche un processus d'acquérir plus de privilèges que son processus parent.

**Exemple Pratique 9.3: Pod avec Security Context**

Le manifeste YAML suivant montre un exemple de configuration de Security Context.

**Code Block 9.7: Configuration d'un Security Context pour un Pod et un Conteneur**

```
$
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod-example
spec:
  securityContext: # Contexte de s curit  au niveau du Pod
    runAsUser: 1000      # Tous les conteneurs s'ex cuteront avec ↩
    ↪ UID 1000
    runAsGroup: 3000     # Tous les conteneurs s'ex cuteront avec ↩
    ↪ GID 3000
    fsGroup: 2000        # Les volumes mont s appartiendront au ↩
    ↪ GID 2000
  containers:
  - name: my-secure-container
    image: alpine:latest
    command: [ "sh", "-c", "id && sleep 3600" ]
    securityContext: # Contexte de s curit  sp cifique au ↩
    ↪ conteneur (peut surcharger)
      runAsNonRoot: true # Doit s'ex cuter en non-root
      allowPrivilegeEscalation: false
      capabilities:
        drop: # Supprimer toutes les capabilities par d faut
        - "ALL"
        add: # Ajouter uniquement les capabilities n cessaires
```

**Code Block 9.7: Configuration d'un Security Context pour un Pod et un Conteneur**

```
21 $          - "NET_BIND_SERVICE" # Permet de lier des ports < 1024 en ↩
       ↪ tant que non-root
22          readOnlyRootFilesystem: true
```

**Listing 9.7 –** pod-with-security-context.yaml

## §9.5 Admission de la Sécurité des Pods (Pod Security Admission)

Pour renforcer la sécurité des Pods à l'échelle du cluster, Kubernetes a introduit (et stabilisé depuis la version 1.25) le contrôleur d'admission *Pod Security Admission* (PSA). Il remplace l'ancienne fonctionnalité *PodSecurityPolicy* (PSP), qui a été dépréciée.

**Définition 9.4: Pod Security Admission (PSA)**

Pod Security Admission est un contrôleur d'admission intégré qui applique les Standards de Sécurité des Pods (Pod Security Standards) au niveau du namespace. Ces standards sont des politiques prédéfinies (`Privileged`, `Baseline`, `Restricted`) qui définissent différents niveaux de sécurité pour les Pods [34].

Les administrateurs peuvent configurer des labels sur les namespaces pour indiquer quel standard de sécurité appliquer et quelle action entreprendre si un Pod ne respecte pas ce standard (par exemple, `enforce` pour rejeter, `audit` pour journaliser, `warn` pour avertir l'utilisateur).

- **Privileged :** Politique la plus permissive, essentiellement aucune restriction. Destinée aux charges de travail système de confiance.
- **Baseline :** Politique minimalement restrictive, empêchant les escalades de privilèges connues. Elle est conçue pour une adoption facile pour les charges de travail courantes.
- **Restricted :** Politique fortement restrictive, suivant les meilleures pratiques actuelles de renforcement des Pods. Destinée aux charges de travail sensibles à la sécurité.

**Exemple Pratique 9.4: Configuration de Pod Security Admission sur un Namespace**

Pour appliquer le standard `Baseline` et auditer les violations du standard `Restricted` sur un namespace, on utilise les commandes suivantes.

**Code Block 9.8: Labels de Namespace pour Pod Security Admission**

```
$
# Appliquer la politique Baseline en mode 'enforce'
kubectl label --overwrite ns my-secure-namespace \
  pod-security.kubernetes.io/enforce=baseline

# Optionnel: auditer pour la politique Restricted
kubectl label --overwrite ns my-secure-namespace \
  pod-security.kubernetes.io/audit=restricted

# Optionnel: avertir pour la politique Restricted
kubectl label --overwrite ns my-secure-namespace \
  pod-security.kubernetes.io/warn=restricted
```

**Listing 9.8 –** Appliquer Pod Security Admission

Si un Pod est ensuite créé dans `my-secure-namespace` et qu'il ne respecte pas le standard `Baseline` (par exemple, s'il demande à s'exécuter en mode privilégié), il sera rejeté.

**Pro Tips 9.2: Passer de PodSecurityPolicy à Pod Security Admission**

Si vous migrez depuis des clusters plus anciens utilisant PodSecurityPolicy, une planification est nécessaire pour adopter Pod Security Admission. Des outils et des guides sont disponibles pour faciliter cette transition.

**Question Ouverte 9.1: Votre Stratégie de Sécurité pour les Pods**

Quelles configurations de Security Context considéreriez-vous comme essentielles pour la plupart de vos applications ? Quel niveau de Pod Security Standard (Privileged, Baseline, Restricted) viseriez-vous par défaut pour les namespaces de vos applications, et pourquoi ?

# 10 Écosystème et Outils Avancés

Kubernetes lui-même est une plateforme puissante, mais sa véritable force réside également dans le vaste écosystème d'outils et de concepts qui se sont développés autour de lui. Ces outils simplifient le déploiement, la gestion de la configuration, l'extensibilité et l'exploitation de Kubernetes dans divers environnements. Ce chapitre présente certains des outils et concepts les plus influents : Helm pour la gestion de paquets, Kustomize pour la personnalisation déclarative, les Opérateurs pour l'automatisation d'applications complexes, et un aperçu des services Kubernetes managés.

## §10.1 Helm : Le Gestionnaire de Paquets pour Kubernetes

**Définition 10.1: Helm**

Helm se décrit comme "le gestionnaire de paquets pour Kubernetes". Il aide à définir, installer et mettre à niveau même les applications Kubernetes les plus complexes. Helm utilise un format de paquetage appelé *Charts*. Un Chart est une collection de fichiers qui décrivent un ensemble de ressources Kubernetes associées [13].

Pensez à Helm comme à `apt`, `yum`, ou `brew` pour Kubernetes. Au lieu d'appliquer manuellement de multiples fichiers YAML pour une application (par exemple, un Deployment, un Service, un ConfigMap, un Secret), vous pouvez utiliser un Chart Helm qui les regroupe et gère leurs dépendances et configurations.

### 10.1.1 Concepts Clés de Helm

- **Chart :** Un paquet Helm. Il contient tous les modèles de ressources, les valeurs par défaut, et les métadonnées nécessaires pour déployer une application ou un service sur un cluster Kubernetes.
- **Repository (Dépôt) :** Un emplacement où les Charts sont collectés et partagés. Artifact Hub [12] est un registre public populaire pour trouver des Charts Helm.

- **Release :** Une instance d'un Chart déployée sur un cluster Kubernetes. Un même Chart peut être déployé plusieurs fois sur le même cluster, chaque déploiement créant une nouvelle Release avec sa propre configuration.
- **Values (Valeurs) :** Permettent de personnaliser un Chart Helm. Chaque Chart est livré avec un fichier `values.yaml` qui expose des options de configuration que vous pouvez surcharger au moment du déploiement (par exemple, la version de l'image, le nombre de répliques, les noms de domaine).
- **Templates :** Les fichiers YAML à l'intérieur d'un Chart sont en réalité des modèles (utilisant le moteur de template Go). Helm combine ces modèles avec les valeurs fournies pour générer les manifestes Kubernetes finaux qui sont appliqués au cluster.

### 10.1.2 Utilisation de Helm

Après avoir installé le client Helm, les commandes courantes incluent :

**Code Block 10.1: Commandes Helm de base**

```
$
# Ajouter un dépôt de Charts (exemple: le dépôt stable officiel, aujourd'hui dépréci mais bon pour l'exemple)
helm repo add stable https://charts.helm.sh/stable
# (Note: de nombreux charts sont maintenant sur Artifact Hub et ont leurs propres dépôts)
# Exemple avec le dépôt Bitnami, très populaire:
helm repo add bitnami https://charts.bitnami.com/bitnami

# Mettre à jour la liste des Charts des dépôts configurés
helm repo update

# Rechercher un Chart dans les dépôts ou sur Artifact Hub
helm search repo wordpress
helm search hub prometheus # Recherche sur Artifact Hub

# Installer un Chart pour créer une Release
# Cela déploiera WordPress avec le nom de release 'my-blog' dans le namespace 'default'
helm install my-blog bitnami/wordpress

# Lister les Releases déployées
helm list
# ou helm ls

# Afficher le statut d'une Release
helm status my-blog

# Mettre à niveau une Release avec une nouvelle version du Chart ou de nouvelles valeurs
# (par exemple, en passant des valeurs via un fichier YAML personnalisé values-custom.yaml)
helm upgrade my-blog bitnami/wordpress -f values-custom.yaml

# Revenir à une version précédente d'une Release (Rollback)
helm rollback my-blog 1 # Revenir à la révision 1 de my-blog
```

**Code Block 10.1: Commandes Helm de base**

```
$
# Désinstaller une Release (supprime toutes les ressources associées)
helm uninstall my-blog
```

**Listing 10.1 –** Exemples de commandes Helm

**Pro Tips 10.1: Avantages de Helm**

- **Gestion de la Complexité :** Simplifie le déploiement d'applications complexes avec de nombreuses ressources interdépendantes.
- **Réutilisabilité :** Les Charts peuvent être partagés et réutilisés.
- **Personnalisation Facile :** Les valeurs permettent d'adapter les déploiements à différents environnements.
- **Gestion du Cycle de Vie :** Facilite les mises à jour, les rollbacks, et la suppression des applications.
- **Gestion des Dépendances :** Un Chart peut dépendre d'autres Charts.

## §10.2 Kustomize : Gestion Déclarative de la Configuration

**Définition 10.2: Kustomize**

Kustomize est un outil autonome pour personnaliser les configurations d'applications Kubernetes de manière déclarative, sans utiliser de "templating". Il vous permet de superposer des "patches" sur un ensemble de fichiers YAML de base pour générer des configurations spécifiques à différents environnements ou variantes, tout en gardant les bases communes. Kustomize est intégré nativement dans `kubectl` depuis la version 1.14 (`kubectl apply -k <répertoire>`) [45].

Kustomize fonctionne en prenant un ensemble de fichiers YAML de "base" et un fichier `kustomization.yaml` qui décrit comment modifier ces bases.

### 10.2.1 Principes de Kustomize

- **Bases et Overlays (Superpositions) :** Vous avez un ensemble de manifestes de base (par exemple, pour un environnement de développement). Pour un autre environnement (par exemple, production), vous créez un "overlay" qui ne contient que les différences (patches, modifications de labels, d'images, de nombre de répliques).

- **Pas de Templating :** Contrairement à Helm, Kustomize n'utilise pas de langage de template. Il manipule directement les structures YAML.
- **Fichier `kustomization.yaml` :** Ce fichier au cœur de Kustomize liste les ressources de base, les patches à appliquer, les générateurs de ConfigMaps/Secrets, les modifications de noms, de labels, d'images, etc.

**Pro Tips 10.2: Helm vs Kustomize**

Helm et Kustomize sont parfois vus comme des alternatives, mais ils peuvent aussi être complémentaires. Helm est excellent pour packager et distribuer des applications tierces ou complexes. Kustomize excelle dans la personnalisation fine et déclarative de vos propres applications pour différents environnements sans la complexité du templating. Certains utilisent Kustomize pour personnaliser les manifestes générés par Helm.

## §10.3 Opérateurs Kubernetes : Automatisation d'Applications Stateful

Gérer des applications stateless avec des Deployments est relativement simple. Cependant, les applications stateful (comme les bases de données, les systèmes de cache distribué, les systèmes de monitoring complexes) nécessitent souvent des connaissances opérationnelles spécifiques pour leur déploiement, leur mise à l'échelle, leurs sauvegardes, leurs mises à jour, et leur gestion des pannes.

**Définition 10.3: Opérateur Kubernetes**

Un Opérateur est une méthode de packaging, déploiement et gestion d'une application Kubernetes. Un Opérateur est un contrôleur spécifique à une application qui étend l'API Kubernetes pour créer, configurer et gérer des instances d'applications complexes et stateful pour le compte d'un utilisateur Kubernetes. Il s'appuie sur les concepts de base de Kubernetes, notamment les contrôleurs et les Custom Resource Definitions (CRDs) [31].

Un Opérateur encode l'expertise humaine (opérationnelle) dans un logiciel qui s'exécute à l'intérieur de votre cluster Kubernetes.

### 10.3.1 Custom Resource Definitions (CRDs)

Les CRDs permettent d'étendre l'API Kubernetes avec vos propres types de ressources personnalisées. Par exemple, un opérateur pour une base de données PostgreSQL pourrait définir une CRD appelée `PostgresqlCluster`. Vous pourriez alors créer un objet de type `PostgresqlCluster` avec un simple YAML, et l'opérateur se chargerait de créer tous les StatefulSets, Services, ConfigMaps, Secrets, PVs/PVCs, et d'orchestrer les opérations complexes (comme la création de répliques, les sauvegardes, les basculements) pour ce cluster PostgreSQL.

### 10.3.2 Fonctionnement d'un Opérateur

1. **Définition de la CRD :** L'opérateur installe une CRD qui définit le nouveau type de ressource (par exemple, `EtcdCluster`, `Prometheus`, `CassandraDatacenter`).
2. **Déploiement du Contrôleur de l'Opérateur :** Le code de l'opérateur (un contrôleur personnalisé) est déployé dans le cluster (souvent comme un Deployment).
3. **Création d'une Ressource Personnalisée :** L'utilisateur crée une instance de la ressource personnalisée (par exemple, un objet YAML de `kind: PostgresqlCluster`).
4. **Boucle de Contrôle de l'Opérateur :** Le contrôleur de l'opérateur surveille les objets de sa CRD. Lorsqu'il détecte une nouvelle ressource personnalisée (ou une modification), il effectue les actions nécessaires pour atteindre l'état désiré (par exemple, provisionner des disques, configurer des StatefulSets, initialiser la base de données, configurer le monitoring). Il peut aussi gérer des opérations de "jour 2" comme les mises à jour, les sauvegardes, la mise à l'échelle, et la récupération après incident.

**Note Importante 10.1: Trouver des Opérateurs**

OperatorHub.io [10] est un registre communautaire où vous pouvez trouver des Opérateurs pour de nombreuses applications populaires. Des frameworks comme Operator SDK, Kubebuilder, ou Metacontroller aident au développement d'Opérateurs.

## §10.4 Kubernetes Managés dans le Cloud (AKS, EKS, GKE)

Bien qu'il soit possible de construire et de gérer son propre cluster Kubernetes à partir de zéro, cela représente une charge opérationnelle significative (mise en place du control plane, gestion des mises à jour, sécurité, etc.). Les principaux fournisseurs de cloud public offrent des services Kubernetes managés qui simplifient grandement cela :

- **Azure Kubernetes Service (AKS)** par Microsoft Azure.
- **Amazon Elastic Kubernetes Service (EKS)** par Amazon Web Services.
- **Google Kubernetes Engine (GKE)** par Google Cloud Platform.

D'autres fournisseurs comme DigitalOcean, Linode, IBM Cloud, Oracle Cloud proposent également des solutions Kubernetes managées.

### 10.4.1 Avantages des Services Managés

- **Control Plane Géré :** Le fournisseur de cloud gère la disponibilité, la scalabilité, et les mises à jour du control plane Kubernetes. Vous ne payez généralement que pour les nœuds workers.
- **Intégration Facilitée :** Intégration native avec d'autres services du cloud (stockage, réseau, bases de données, identité, monitoring, logging).
- **Scalabilité Simplifiée :** Facilité pour ajouter/supprimer des nœuds workers, souvent avec des options d'autoscaling des nœuds.

- **Sécurité et Conformité :** Les fournisseurs appliquent des configurations de sécurité par défaut et aident à répondre à certaines exigences de conformité.
- **Mises à Jour Facilitées :** Processus de mise à jour du cluster (control plane et nœuds workers) plus simples et souvent automatisés.

### 10.4.2 Inconvénients Potentiels

- **Coût :** Peut être plus cher que de gérer soi-même, surtout à grande échelle, bien que cela doive être mis en balance avec le coût opérationnel économisé.
- **Moins de Flexibilité/Contrôle :** Certaines configurations très avancées du control plane peuvent ne pas être accessibles.
- **Dépendance au Fournisseur (Vendor Lock-in) :** Bien que Kubernetes soit portable, l'utilisation intensive des intégrations spécifiques au cloud peut rendre la migration plus complexe.
- **Latence des Mises à Jour :** Vous dépendez du rythme du fournisseur pour la disponibilité des nouvelles versions de Kubernetes.

TechTarget [11] est un exemple d'article qui compare ces services, mais il est toujours bon de consulter la documentation la plus récente de chaque fournisseur.

**Question Ouverte 10.1: Votre Boîte à Outils Kubernetes**

Parmi les outils présentés (Helm, Kustomize, Opérateurs), lesquels vous semblent les plus pertinents pour vos projets ou votre organisation, et pourquoi ? Si vous deviez démarrer un nouveau projet sur Kubernetes, opteriez-vous pour un service managé ou une installation auto-gérée, et quels seraient vos critères de décision ?

# 11

# Bonnes Pratiques, Conclusion et Prochaines Étapes

Félicitations ! Vous avez parcouru un long chemin depuis les concepts de base jusqu'aux aspects plus avancés de Kubernetes. Ce dernier chapitre vise à synthétiser certaines des bonnes pratiques essentielles pour travailler efficacement avec Kubernetes, à conclure notre parcours d'apprentissage, et à vous indiquer des pistes pour continuer à développer votre expertise dans cet écosystème dynamique.

## §11.1 Bonnes Pratiques Générales pour Kubernetes

Adopter de bonnes pratiques dès le début facilitera grandement la gestion, la maintenabilité, la sécurité et la scalabilité de vos applications sur Kubernetes. La documentation officielle et des ouvrages comme celui de Burns et al. [1] regorgent de conseils, mais voici quelques points essentiels :

- **Utiliser des Manifestes Déclaratifs (YAML) :** Privilégiez toujours l'approche déclarative (`kubectl apply -f <fichier.yaml>`) à l'approche impérative (`kubectl run`, `kubectl create`) pour les déploiements en production. Stockez vos manifestes YAML dans un système de contrôle de version (comme Git) pour suivre les changements, faciliter la collaboration et permettre le GitOps.
- **Organisation des Fichiers YAML :** Pour les applications complexes, organisez vos manifestes en répertoires logiques. Utilisez des outils comme Kustomize (Chapitre 10.2) pour gérer les variations entre environnements (développement, staging, production) de manière propre.
- **Labels et Annotations Cohérents :** Définissez une stratégie de nommage et d'utilisation cohérente pour les labels afin d'organiser et de sélectionner efficacement vos ressources. Utilisez les annotations pour les métadonnées non identifiantes.
- **Spécifier les Versions d'Image :** Utilisez toujours des tags d'image spécifiques (par exemple, `mon-app:1.2.3`) plutôt que `:latest` dans vos définitions de Pods pour assurer des déploiements reproductibles et prévisibles.
- **Gestion des Ressources (Requests et Limits) :** Définissez systématiquement des requests et limits de CPU et de mémoire pour vos conteneurs (Chapitre 7.2). Cela améliore la stabilité

du cluster, aide le scheduler, et permet d'assigner des classes de Qualité de Service (QoS) appropriées.

- **Health Probes (Sondes de Santé) :** Configurez des Liveness, Readiness, et Startup Probes pour tous vos conteneurs critiques afin d'assurer que Kubernetes puisse gérer correctement leur cycle de vie et le routage du trafic (Chapitre 7.1).
- **Principe du Moindre Privilège (Sécurité) :**
  - Configurez RBAC (Chapitre 9.2) en accordant uniquement les permissions nécessaires aux utilisateurs et aux ServiceAccounts.
  - Utilisez des Security Contexts (Chapitre 9.4) pour restreindre les capacités des Pods et des conteneurs (par exemple, `runAsNonRoot: true`, `readOnlyRootFilesystem: true`, suppression des capabilities inutiles).
  - Utilisez Pod Security Admission (Chapitre 9.5) pour appliquer des standards de sécurité à l'échelle du namespace.
  - Ne montez pas de jetons ServiceAccount dans les Pods qui n'en ont pas besoin (`automountServiceAccountToken: false`).
  - Sécurisez l'accès à l'API server et à `etcd`.
- **Gestion de la Configuration et des Secrets :** Utilisez des ConfigMaps et des Secrets (Chapitre 5) pour externaliser la configuration et les données sensibles de vos images de conteneurs. Pour les Secrets en production, envisagez le chiffrement au repos et l'intégration avec des solutions de gestion de secrets externes.
- **Stratégies de Déploiement Appropriées :** Utilisez la stratégie `RollingUpdate` (par défaut pour les Deployments) pour les mises à jour sans interruption de service. Explorez d'autres stratégies (Blue/Green, Canary) si nécessaire, souvent facilitées par des outils de Service Mesh ou des Ingress Controllers avancés.
- **Logging et Monitoring Centralisés :** Mettez en place des solutions de logging et de monitoring centralisées (Chapitre 8.1 et 8.2) pour l'observabilité de votre cluster et de vos applications.
- **Minimiser la Taille des Images de Conteneurs :** Utilisez des images de base minimales (comme Alpine Linux), des builds multi-étapes dans vos Dockerfiles, et supprimez les dépendances inutiles pour réduire la taille des images. Cela accélère les téléchargements, réduit la surface d'attaque, et économise de l'espace disque.
- **Réseau Sécurisé :** Utilisez des Network Policies (Chapitre 4.4) pour segmenter le trafic entre les Pods et appliquer le principe du moindre privilège au niveau réseau.
- **Haute Disponibilité du Control Plane :** En production, assurez-vous que votre control plane est configuré en mode haute disponibilité (plusieurs instances des composants du control plane, etcd clusterisé). Ceci est généralement pris en charge par les services Kubernetes managés.

De nombreuses autres bonnes pratiques existent, souvent spécifiques à des cas d'usage ou à des composants de l'écosystème. Le blog officiel de Kubernetes [23] est une bonne source d'articles sur ce sujet.

## §11.2 Récapitulatif des Concepts Clés

Au cours de ce guide, nous avons exploré les concepts fondamentaux et avancés de Kubernetes. Revoyons brièvement les points essentiels :

- **Architecture :** La distinction entre le Control Plane (API server, etcd, scheduler, controller manager) et les Nœuds Workers (kubelet, kube-proxy, container runtime).
- **Objets Fondamentaux :** Les Pods comme unité de base, les Labels et Sélecteurs pour l'organisation, les ReplicaSets pour la disponibilité, les Deployments pour les mises à jour déclaratives, et les Namespaces pour l'isolation logique.
- **Réseau :** Les Services pour l'exposition et la découverte de Pods (ClusterIP, NodePort, LoadBalancer), les Ingress pour le routage HTTP/S avancé, et les Network Policies pour la sécurité réseau.
- **Configuration et Secrets :** Les ConfigMaps pour la configuration non sensible et les Secrets pour les données confidentielles, découplés des images.
- **Stockage Persistant :** Les Volumes pour le stockage lié au Pod, et le triptyque PersistentVolume (PV), PersistentVolumeClaim (PVC), et StorageClass pour le stockage persistant découplé et dynamique.
- **Cycle de Vie et Déploiements Avancés :** Les Health Probes pour la santé des applications, la gestion des Requests et Limits de ressources, les Jobs et CronJobs pour les tâches batch/planifiées, et les DaemonSets pour les Pods sur chaque nœud.
- **Observabilité et Maintenance :** Le logging (local et centralisé), le monitoring (de base et avancé avec Prometheus/Grafana), le débogage, et la maintenance des nœuds.
- **Sécurité :** L'authentification, l'autorisation (RBAC), les ServiceAccounts, les Security Contexts, et Pod Security Admission.
- **Écosystème :** Des outils comme Helm, Kustomize, et le concept des Opérateurs pour étendre et simplifier l'utilisation de Kubernetes.

**Pro Tips 11.1: Le Modèle Déclaratif : La Clé de Voûte**

Si vous ne deviez retenir qu'un seul principe de Kubernetes, ce serait son modèle déclaratif. En décrivant l'état souhaité de votre système, vous laissez Kubernetes se charger de la complexité de l'atteindre et de le maintenir. Cette approche est fondamentale pour l'automatisation, la résilience et la scalabilité qu'offre la plateforme.

## §11.3 Ressources pour Aller Plus Loin

Kubernetes est un projet vaste et en constante évolution. Votre apprentissage ne s'arrête pas ici ! Voici quelques ressources pour continuer à progresser :

- **Documentation Officielle de Kubernetes (`kubernetes.io/docs`) [25] :** C'est LA source de vérité. Elle est complète, bien organisée, et régulièrement mise à jour. Explorez les sections Concepts, Tâches, Tutoriels, et Référence.
- **Blog Kubernetes (`kubernetes.io/blog`) :** Pour les annonces de nouvelles versions, les articles de fond sur des fonctionnalités spécifiques, et les retours d'expérience de la communauté.
- **Communauté Kubernetes :**
  - **Forums et Groupes de Discussion :** Stack Overflow (tag `kubernetes`), forum Discuss Kubernetes (`discuss.kubernetes.io`).
  - **Slack Kubernetes (`slack.k8s.io`) :** De nombreux canaux dédiés à des sujets spécifiques, où vous pouvez poser des questions et interagir avec d'autres utilisateurs et contributeurs.
  - **Meetups et Conférences :** Les KubeCons (organisées par la CNCF) sont les événements majeurs, mais de nombreux meetups locaux existent.
  - **Groupes d'Intérêt Spéciaux (SIGs) :** Si vous souhaitez contribuer ou suivre de près le développement d'une partie spécifique de Kubernetes (par exemple, SIG-Network, SIG-Storage, SIG-Security).
- **Livres :** Outre ceux déjà cités (Hightower, Burns et Beda [7], Poulton [9], Burns et al. [1]), de nombreux autres ouvrages explorent des aspects spécifiques de Kubernetes et de l'écosystème cloud native.
- **Tutoriels et Cours en Ligne :** De nombreuses plateformes (Katacoda, KodeKloud, Udemy, Coursera, A Cloud Guru, etc.) offrent des environnements interactifs et des cours vidéo pour apprendre Kubernetes.
- **CNCF Cloud Native Interactive Landscape (`landscape.cncf.io`) [2] :** Pour explorer l'immense écosystème de projets et de produits cloud native gravitant autour de Kubernetes.
- **GitHub (`github.com/kubernetes`) :** Le code source de Kubernetes et de nombreux projets de l'écosystème sont open source. Explorer le code peut être une excellente façon d'apprendre.

## §11.4 Certifications Kubernetes

Pour valider et démontrer vos compétences Kubernetes, la Cloud Native Computing Foundation (CNCF), en partenariat avec la Linux Foundation, propose plusieurs certifications reconnues par l'industrie [46] :

- **Certified Kubernetes Administrator (CKA) :** Destinée aux administrateurs Kubernetes, cette certification teste la capacité à effectuer les tâches d'administration d'un cluster Kubernetes (installation, configuration, dépannage, sécurisation). Elle est très axée sur la pratique en ligne de commande.
- **Certified Kubernetes Application Developer (CKAD) :** Cible les développeurs qui construisent, déploient et configurent des applications sur Kubernetes. Elle se concentre sur la définition des ressources applicatives, l'utilisation des objets API de base, et le déploiement d'applications.

- **Certified Kubernetes Security Specialist (CKS) :** Une certification plus avancée qui nécessite d'être CKA au préalable. Elle valide les compétences en matière de sécurisation des clusters Kubernetes et des applications qui y sont déployées (sécurité du build, du déploiement, et de l'exécution).

Préparer ces certifications est un excellent moyen d'approfondir vos connaissances pratiques de Kubernetes.

## §11.5 Conclusion

Kubernetes est une technologie transformatrice qui a redéfini la manière dont nous déployons et gérons les applications à l'ère du cloud native. Sa complexité initiale peut sembler intimidante, mais sa puissance, sa flexibilité et la force de sa communauté en font un investissement précieux pour tout professionnel de l'informatique moderne.

Nous espérons que ce guide vous a fourni une base solide pour comprendre les concepts clés, interagir avec un cluster, déployer vos premières applications, et appréhender les aspects plus avancés de la plateforme. Le voyage dans le monde de Kubernetes est continu ; la curiosité, la pratique régulière, et l'engagement avec la communauté seront vos meilleurs alliés pour maîtriser cet outil fascinant.

**Question Ouverte 11.1: Votre Parcours Kubernetes**

Quels sont les aspects de Kubernetes que vous trouvez les plus intéressants ou les plus pertinents pour vos objectifs professionnels ou personnels ? Quels sujets aimeriez-vous explorer plus en profondeur après ce cours ?

# 12

# Projet Pratique Global : Déploiement d'une Application Web Multi-tiers

Ce projet final vise à consolider vos connaissances en vous guidant à travers le déploiement d'une application web multi-tiers sur Kubernetes. Nous allons simuler le déploiement d'une application de vote simple, composée d'un frontend, d'une API backend, et d'une base de données.

**Note Importante 12.1: Choix de l'Application**

Pour ce projet, vous pouvez utiliser une application de démonstration existante comme "Example Voting App" (souvent utilisée pour les démos Docker et Kubernetes) ou créer une application simple vous-même si vous êtes à l'aise avec le développement. L'important est d'avoir au moins trois composants distincts à déployer et à interconnecter. Si vous utilisez "Example Voting App", vous trouverez ses Dockerfiles et son code source en ligne (par exemple, sur GitHub chez des utilisateurs comme `dockersamples`).

## §12.1 Objectifs du Projet

À la fin de ce projet, vous aurez :

- Conteneurisé (si nécessaire) les différents composants de l'application.
- Écrit des manifestes Kubernetes (Deployments, Services, ConfigMaps, Secrets, PersistentVolumeClaims) pour chaque composant.
- Déployé l'application sur votre cluster Kubernetes.
- Configuré le réseau pour permettre la communication entre les composants et l'accès externe au frontend.
- Mis en place des configurations et des secrets de manière sécurisée.
- Assuré la persistance des données pour la base de données.
- (Optionnel) Mis en œuvre des health probes et des stratégies de mise à jour.

## §12.2 Composants de l'Application (Exemple "Voting App")

Notre application de vote exemple se compose typiquement de :

1. **Frontend (Vote) :** Une application web (par exemple, Python/Flask, Node.js/React) qui permet aux utilisateurs de voter pour une option (ex : "Chats" vs "Chiens").
2. **API Backend (Worker) :** Une API (par exemple, .NET Core, Java/Spring Boot) qui reçoit les votes du frontend et les stocke dans une file d'attente de messages (ex : Redis). Un worker consomme ensuite les votes de la file et les persiste dans une base de données. *Alternative simplifiée :* L'API écrit directement dans la base de données.
3. **Base de Données (DB) :** Une base de données relationnelle (par exemple, PostgreSQL) pour stocker les résultats des votes.
4. **(Optionnel) File d'Attente de Messages (Queue) :** Un service comme Redis, utilisé par l'API pour mettre les votes en attente avant traitement par un worker.
5. **(Optionnel) Résultat (Result) :** Une application web qui se connecte à la base de données et affiche les résultats du vote en temps réel.

## §12.3 Étapes du Projet

1. **Préparation et Conteneurisation :**
   - Si vous n'utilisez pas d'images préexistantes, écrivez des Dockerfiles pour chaque composant de l'application.
   - Construisez les images Docker et poussez-les vers un registre de conteneurs accessible par votre cluster Kubernetes (Docker Hub, GHCR, ou un registre privé).

**Pro Tips 12.1: Utiliser des Images Publiques pour Simplifier**

Pour vous concentrer sur Kubernetes, vous pouvez utiliser des images publiques existantes pour chaque composant si vous trouvez une version de "Example Voting App" avec des images prêtes à l'emploi.

2. **Définition des Ressources Kubernetes (Manifestes YAML) :** Pour chaque composant (frontend, API/worker, DB, queue, result), créez les manifestes YAML nécessaires :
   - **Namespaces :** Créez un namespace dédié pour cette application (par exemple, `voting-app`).
   - **Base de Données (DB) :**
     - `PersistentVolumeClaim` pour le stockage des données.
     - `Secret` pour les identifiants de la base de données (utilisateur, mot de passe).

  - `Deployment` ou `StatefulSet` (un StatefulSet est plus approprié pour les bases de données, mais un Deployment peut suffire pour une démo si vous ne gérez pas la réplication complexe).
  - `Service` (ClusterIP) pour exposer la base de données aux autres composants.
- **File d'Attente de Messages (Queue - ex : Redis) :**
  - `Deployment`.
  - `Service` (ClusterIP).
  - (Optionnel) `ConfigMap` si Redis nécessite une configuration spécifique.
- **API Backend / Worker :**
  - `Deployment`.
  - `ConfigMap` pour les URLs de la base de données et de la queue (injectées via des variables d'environnement faisant référence aux Services correspondants).
  - `Secret` si l'API a besoin de s'authentifier auprès de la base de données (injecté via des variables d'environnement).
  - (Pas de Service externe nécessaire si le frontend communique directement avec la queue ou si l'API est un worker interne).
- **Frontend (Vote) :**
  - `Deployment`.
  - `ConfigMap` pour l'URL de l'API backend ou de la queue.
  - `Service` (NodePort ou LoadBalancer, ou utiliser un Ingress) pour exposer le frontend à l'extérieur.
- **(Optionnel) Application Résultat :**
  - `Deployment`.
  - `ConfigMap` pour l'URL de la base de données.
  - `Service` (NodePort ou LoadBalancer, ou utiliser un Ingress).

**⚠ Attention 12.1: Ordre de Déploiement**

Pensez à l'ordre dans lequel vous déployez les composants. En général, les dépendances (comme la base de données et la queue) doivent être prêtes avant les applications qui les utilisent.

3. **Déploiement sur Kubernetes :**
   - Utilisez `kubectl apply -f <répertoire_des_manifestes> -n voting-app` pour déployer tous vos manifestes.
   - Vérifiez le statut de tous vos Pods, Deployments, Services, PVCs.
   - Déboguez les problèmes en utilisant `kubectl logs`, `kubectl describe`.
4. **Configuration du Réseau et Accès Externe :**
   - Assurez-vous que les composants peuvent communiquer entre eux via les noms de Service DNS internes.
   - Configurez un Service de type `LoadBalancer` ou `NodePort` pour le frontend (et l'application résultat si vous l'avez).

- (Avancé) Mettez en place un `Ingress` pour router le trafic vers le frontend et l'application résultat en utilisant des chemins ou des noms d'hôtes distincts.

5. **Tests et Validation :**
   - Accédez à l'application frontend via votre navigateur.
   - Effectuez quelques votes.
   - Vérifiez que les votes sont correctement enregistrés dans la base de données (vous pouvez utiliser `kubectl exec` pour vous connecter à la base de données si nécessaire, ou via l'application résultat).
6. **Améliorations (Optionnel) :**
   - Ajoutez des `livenessProbe` et `readinessProbe` à vos Deployments.
   - Définissez des `requests` et `limits` de ressources pour vos conteneurs.
   - Implémentez des `NetworkPolicies` pour restreindre la communication (par exemple, seul l'API backend peut accéder à la base de données).
   - Utilisez Helm pour packager votre application.
   - Utilisez Kustomize pour gérer une variante "production" avec plus de répliques et des ressources différentes.

## §12.4 Points d'Attention et Défis

- **Configuration des Noms de Service DNS :** Assurez-vous que vos applications utilisent correctement les noms DNS des Services pour se connecter les unes aux autres (par exemple, `http://db-service.voting-app.svc.cluster.local:5432`).
- **Persistance des Données :** La configuration correcte des PVCs et des montages de volume pour la base de données est cruciale.
- **Ordre de Démarrage des Dépendances :** Les applications peuvent échouer au démarrage si leurs dépendances (comme la base de données) ne sont pas encore prêtes. Les Readiness Probes et les mécanismes de nouvelle tentative dans les applications peuvent aider.
- **Sécurité des Secrets :** Manipulez les identifiants de base de données avec soin.

**Pro Tips 12.2: Commencez Simplement et Itérez**

N'essayez pas de tout faire parfaitement du premier coup. Commencez par déployer un composant, assurez-vous qu'il fonctionne, puis ajoutez les suivants. Déboguez au fur et à mesure. Une fois que la version de base fonctionne, vous pourrez ajouter les améliorations optionnelles.

### Question Ouverte 12.1: Votre Implémentation du Projet

Quelles ont été les parties les plus difficiles de ce projet ? Quelles décisions de conception avez-vous prises et pourquoi ? Quelles améliorations supplémentaires apporteriez-vous si vous aviez plus de temps ?

Ce projet global est une excellente occasion de mettre en pratique tout ce que vous avez appris. Bonne chance et amusez-vous bien en construisant votre application sur Kubernetes !

# Références Bibliographiques


[1] Brendan Burns et al. *Kubernetes Best Practices : Blueprints for Building Successful Applications on Kubernetes*. O'Reilly Media, 2019. ISBN : 978-1492056478.

[2] Cloud Native Computing Foundation. *CNCF Cloud Native Interactive Landscape*. 2023. URL : https://landscape.cncf.io/ (visité le 30/10/2023).

[3] DigitalOcean. *How To Set Up an Elasticsearch, Fluentd, and Kibana (EFK) Logging Stack on Kubernetes*. 2020. URL : https://www.digitalocean.com/community/tutorials/how-to-set-up-an-elasticsearch-fluentd-and-kibana-efk-logging-stack-on-kubernetes (visité le 30/10/2023).

[4] Docker Inc. *Docker Documentation*. 2023. URL : https://docs.docker.com/ (visité le 28/10/2023).

[5] etcd Authors. *etcd Documentation*. 2023. URL : https://etcd.io/docs/ (visité le 28/10/2023).

[6] Grafana Labs. *Grafana - The open and composable observability platform*. 2023. URL : https://grafana.com/ (visité le 30/10/2023).

[7] Kelsey Hightower, Brendan Burns et Joe Beda. *Kubernetes : Up and Running, 2nd Edition*. O'Reilly Media, 2019. ISBN : 978-1492046530.

[8] Kubernetes Community. *NGINX Ingress Controller - Documentation*. 2023. URL : https://kubernetes.github.io/ingress-nginx/ (visité le 29/10/2023).

[9] Nigel Poulton. *The Kubernetes Book*. Version : March 2021. Nigel Poulton, 2021. ISBN : 979-8700262102.

[10] Red Hat and the Operator Framework Community. *OperatorHub.io - Find and Install Kubernetes Operators*. 2023. URL : https://operatorhub.io/ (visité le 30/10/2023).

[11] TechTarget. *EKS vs. AKS vs. GKE : A face-off of managed Kubernetes services*. 2023. URL : https://www.techtarget.com/searchcloudcomputing/feature/EKS-vs-AKS-vs-GKE-A-face-off-of-managed-Kubernetes-services (visité le 30/10/2023).

[12] The CNCF Authors. *Artifact Hub*. 2023. URL : https://artifacthub.io/ (visité le 30/10/2023).

[13] The Helm Authors. *Helm | Documentation*. 2023. URL : https://helm.sh/docs/ (visité le 30/10/2023).

[14] The Kubernetes Authors. *ConfigMaps - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/configuration/configmap/ (visité le 29/10/2023).

[15] The Kubernetes Authors. *Configure a Security Context for a Pod or Container - Kubernetes*. 2023. URL : https://kubernetes.io/docs/tasks/configure-pod-container/security-context/ (visité le 30/10/2023).

[16] The Kubernetes Authors. *Configure Liveness, Readiness and Startup Probes - Kubernetes*. 2023. URL : https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/ (visité le 30/10/2023).

[17] The Kubernetes Authors. *Controlling Access to the Kubernetes API - Authentication*. 2023. URL : https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (visité le 30/10/2023).

[18] The Kubernetes Authors. *CronJob - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/workloads/controllers/cron-job/ (visité le 30/10/2023).

[19] The Kubernetes Authors. *DaemonSet - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (visité le 30/10/2023).

[20] The Kubernetes Authors. *Deployments - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visité le 28/10/2023).

[21] The Kubernetes Authors. *Ingress - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/services-networking/ingress/ (visité le 29/10/2023).

[22] The Kubernetes Authors. *Jobs - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/workloads/controllers/job/ (visité le 30/10/2023).

[23] The Kubernetes Authors. *Kubernetes Best Practices - Official Blog*. 2023. URL : https://kubernetes.io/blog/tags/best-practices/ (visité le 30/10/2023).

[24] The Kubernetes Authors. *Kubernetes Components*. 2023. URL : https://kubernetes.io/docs/concepts/overview/components/ (visité le 28/10/2023).

[25] The Kubernetes Authors. *Kubernetes Documentation*. 2023. URL : https://kubernetes.io/docs/ (visité le 30/10/2023).

[26] The Kubernetes Authors. *Labels and Selectors - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/ (visité le 28/10/2023).

[27] The Kubernetes Authors. *Logging Architecture - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/cluster-administration/logging/ (visité le 30/10/2023).

[28] The Kubernetes Authors. *Minikube - Local Kubernetes for macOS, Linux, and Windows*. 2023. URL : https://minikube.sigs.k8s.io/docs/ (visité le 28/10/2023).

[29] The Kubernetes Authors. *Namespaces - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visité le 28/10/2023).

[30] The Kubernetes Authors. *Network Policies - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/services-networking/network-policies/ (visité le 29/10/2023).

[31] The Kubernetes Authors. *Operator pattern - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ (visité le 30/10/2023).

[32] The Kubernetes Authors. *Overview of Cloud Native Security - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/security/overview/ (visité le 30/10/2023).

[33] The Kubernetes Authors. *Persistent Volumes - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/storage/persistent-volumes/ (visité le 29/10/2023).

[34] The Kubernetes Authors. *Pod Security Admission - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/security/pod-security-admission/ (visité le 30/10/2023).

[35] The Kubernetes Authors. *Pods - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/workloads/pods/ (visité le 28/10/2023).

[36] The Kubernetes Authors. *Resource Management for Pods and Containers - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ (visité le 30/10/2023).

[37] The Kubernetes Authors. *Secrets - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/configuration/secret/ (visité le 29/10/2023).

[38] The Kubernetes Authors. *Service - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/services-networking/service/ (visité le 29/10/2023).

[39] The Kubernetes Authors. *Service Accounts - Kubernetes*. 2023. URL : https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/ (visité le 30/10/2023).

[40] The Kubernetes Authors. *Storage Classes - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/storage/storage-classes/ (visité le 29/10/2023).

[41] The Kubernetes Authors. *The Kubernetes API*. 2023. URL : https://kubernetes.io/docs/concepts/overview/kubernetes-api/ (visité le 28/10/2023).

[42] The Kubernetes Authors. *Tools for Monitoring Resources - Kubernetes*. 2023. URL : https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/ (visité le 30/10/2023).

[43] The Kubernetes Authors. *Using RBAC Authorization - Kubernetes*. 2023. URL : https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (visité le 30/10/2023).

[44] The Kubernetes Authors. *Volumes - Kubernetes*. 2023. URL : https://kubernetes.io/docs/concepts/storage/volumes/ (visité le 29/10/2023).

[45] The Kustomize Authors. *Kustomize - Declarative Management of Kubernetes Objects*. 2023. URL : https://kustomize.io/ (visité le 30/10/2023).

[46] The Linux Foundation. *Kubernetes Certification (CKA, CKAD, CKS)*. 2023. URL : https://training.linuxfoundation.org/certification/kubernetes-certifications/ (visité le 30/10/2023).

[47] The Prometheus Authors. *Prometheus - Monitoring system  time series database*. 2023. URL : https://prometheus.io/ (visité le 30/10/2023).

[48] Adam Wiggins. *The Twelve-Factor App - III. Config*. 2011. URL : https://12factor.net/config (visité le 29/10/2023).