



Kubernetes

Utopios® Tous droits réservés

Sommaire

1. Introduction Docker et les containers

- Révolution des containers
- Création et utilisation de containers

2. Kubernetes et l'orchestration de containers

- Pourquoi un orchestrateur ?
- Avantage de Kubernetes
- Mise en place de Kubernetes

3. Architecture de Kubernetes

- Principes de fonctionnement
- Composants de kubernetes
- Masters/workers
- Couche réseau

4. Créer un cluster Kubernetes

- Solutions possibles (clouds publiques, distributions tierces, minikube)
- Créer son cluster avec minikube
- Introduction à kubectl
- Commandes et options de base
- Obtenir les informations du cluster
- Contextes et namespaces

5. Déployer sa première application

- Introduction aux Pods
- Que mettre dans un pod ?
- Créer un pod avec kubectl
- Introduction aux fichiers de manifestes
- Metadata, labels, annotations, specs

6. Suivi et diagnostic d'un pod en fonctionnement

- Lister les pods et afficher les détails
- Accès aux logs
- Exécuter des commandes à l'intérieur du pod
- Échanger des fichiers avec le podIGPDE
- Sondes de disponibilité et de bonne santé (liveness et readiness probes)

7. Accéder à ses applications

- Comprendre le réseau virtuel de Kubernetes
- Rendre son service accessible sur le réseau virtuel
- Introduction aux Services et leurs différents types
- Déployer un service NodePort et comprendre son fonctionnement

Sommaire

8. RéPLICATION DES PODS

- Objectifs de la réPLICATION (redondance, passage à l'échelle, sharding)
- Principe de boucle de réconciliation
- Créer et manipuler des ReplicaSet
- Principe de passage à l'échelle horizontal (vs vertical)
- Mettre en place l'auto-scaling horizontal
- Introduction aux DaemonSet

9. GÉRER LE CYCLE DE VIE D'UNE APPLICATION

- Principe de Rolling update
- Créer et gérer des Deployments
- Stratégies de déploiement

10. DÉPLOIEMENT ET PARTAGE DES ÉLÉMENTS DE CONFIGURATION

- Principe de généricité et d'indépendance de l'application et de la plateforme
- Créer des ConfigMaps
- Différents moyens de consommation des ConfigMaps
- Créer et consommer des Secrets

11. GÉRER LES DONNÉES PERSISTANTES ET LES APPLICATIONS STATEFUL

- Problématique du stockage persistant dans le cloud
- Introduction aux volumes, tour d'horizon des différents types de volumes
- Créer une application en exploitant un volume emptyDir
- Monter un ConfigMap en tant que volume
- Utiliser du stockage distant

12. SÉCURITÉ DANS KUBERNETES

- Introduction et manipulation de l'api RBAC
- Introduction aux NetworkPolicies
- Cloisonner une application avec une NetworkPolicy
- Introduction aux PodSecurityPolicies

13. KUBERNETES ET SON ÉCOSYSTÈME

- Déployer des solutions avec Helm
- Etcd, gestion de configuration distribuée



Introduction Docker et les containers

Introduction Docker et les containers

1. Docker et les containers:

- Docker est une plateforme qui permet de créer, déployer et exécuter des applications dans des conteneurs. Un conteneur est une unité standardisée d'exécution du logiciel qui regroupe le code de l'application et toutes ses dépendances, ce qui permet de garantir qu'elle s'exécute de la même manière, quels que soient l'environnement et le système d'exploitation.

2. Révolution des containers:

1. **Isolation:** Avec Docker, chaque application est isolée dans son propre conteneur, ce qui signifie qu'elle fonctionne de manière constante dans différents environnements.
2. **Portabilité:** Les conteneurs peuvent être déplacés entre différents environnements (développement, test, production) sans modification.
3. **Efficacité:** Les conteneurs sont légers et démarrent rapidement, ce qui les rend idéaux pour la mise à l'échelle et le déploiement continu.
4. **Microservices:** Docker facilite la transition vers une architecture de microservices, où chaque service est empaqueté dans son propre conteneur.

Introduction Docker et les containers

3. Crédation et utilisation de containers:

1. **Dockerfile:** Un Dockerfile est un script qui contient les instructions pour construire une image Docker. Il spécifie la base OS, les logiciels à installer, les ports à ouvrir, etc.

```
FROM ubuntu:18.04
RUN apt-get update && apt-get install -y nginx
CMD ["nginx", "-g", "daemon off;"]
```

2. Construire une image:

```
docker build -t monimage:latest .
```

3. Exécuter un conteneur:

```
docker run -d -p 80:80 monimage:latest
```

4. Gestion des conteneurs:

- Lister les conteneurs: docker ps
- Stopper un conteneur: docker stop [CONTAINER_ID]
- Supprimer un conteneur: docker rm [CONTAINER_ID]



Kubernetes et l'orchestration de containers

Kubernetes et l'orchestration de containers

1. Pourquoi un orchestrateur?

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

Kubernetes et l'Orchestration de Containers

2. Avantages de Kubernetes

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

Kubernetes et l'Orchestration de Containers

3. Mise en place de Kubernetes

1. Prérequis:

- Avoir une connaissance de base de Docker.
- Avoir un cluster de machines (physiques ou virtuelles).

2. Installation:

- Vous pouvez installer Kubernetes en utilisant des outils comme [kubeadm](#), kind, k3s.
- Pour une solution clé en main, envisagez des services comme Google Kubernetes Engine (GKE) ou Azure Kubernetes Service (AKS).

3. Configuration initiale:

- Configurer le réseau pour les pods.
- Configurer le stockage pour la persistance des données.

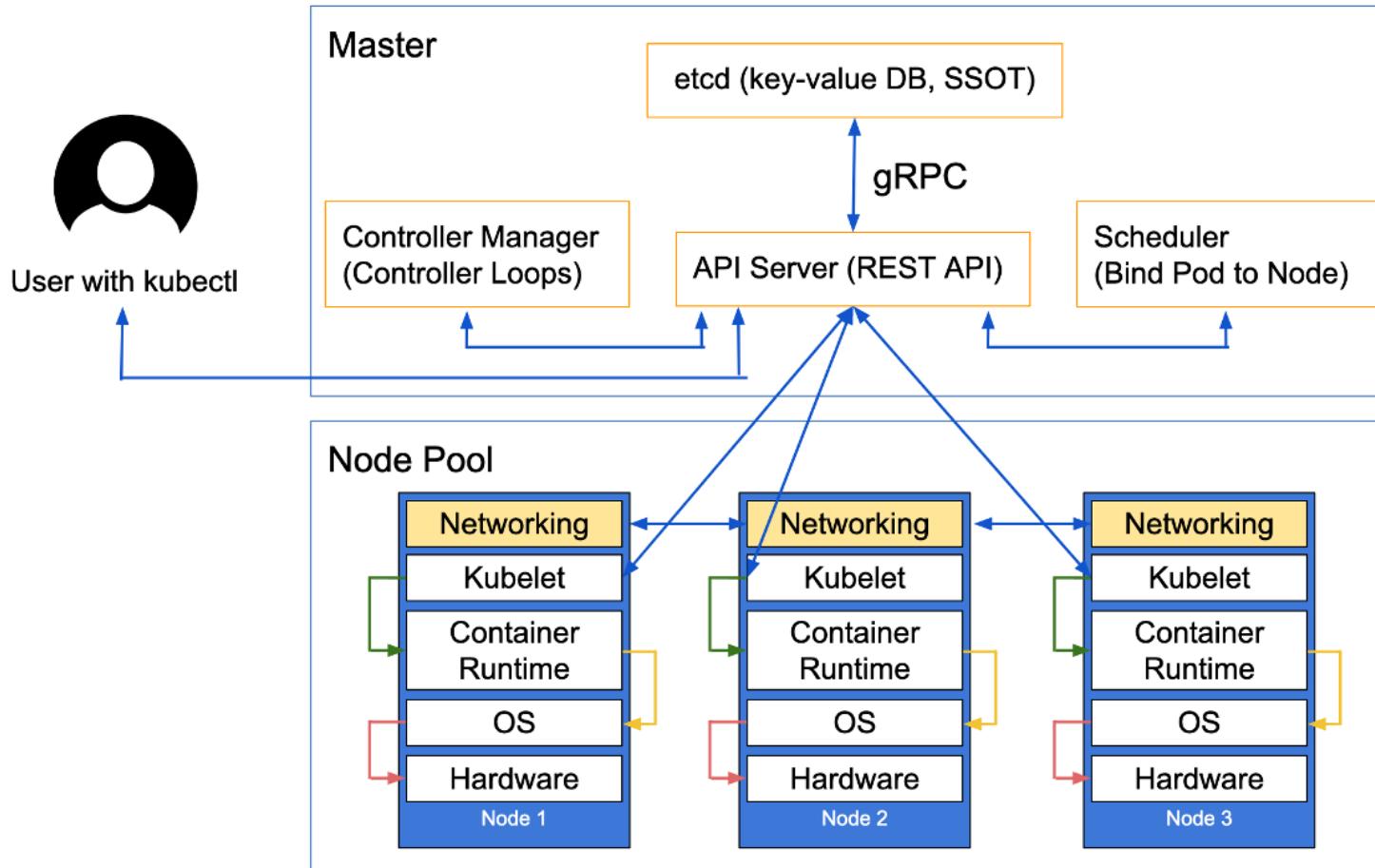
4. Déployer des applications:

- Utiliser des fichiers de configuration YAML pour décrire les ressources nécessaires (comme les déploiements, les services et les volumes).



Architecture de Kubernetes

Architecture de Kubernetes



Architecture de Kubernetes

1. Principes de fonctionnement

- Kubernetes est basé sur une architecture de type maître-esclave (ou master-worker). Le maître (ou master) prend des décisions concernant le cluster, telles que la planification, et répond aux demandes d'API, tandis que les esclaves (ou workers) exécutent les conteneurs.
- 1. État désiré vs état actuel:** L'une des idées fondamentales de Kubernetes est la notion d'état désiré. Vous définissez ce que vous souhaitez voir s'exécuter (par exemple, je veux 3 instances de mon application), et Kubernetes s'efforce de s'assurer que la réalité correspond à cet état.
 - 2. Autoguérison:** Si un conteneur tombe en panne, Kubernetes le redémarre pour maintenir l'état désiré. De même, si une machine entière tombe en panne, les conteneurs qui s'y exécutaient sont redistribués.

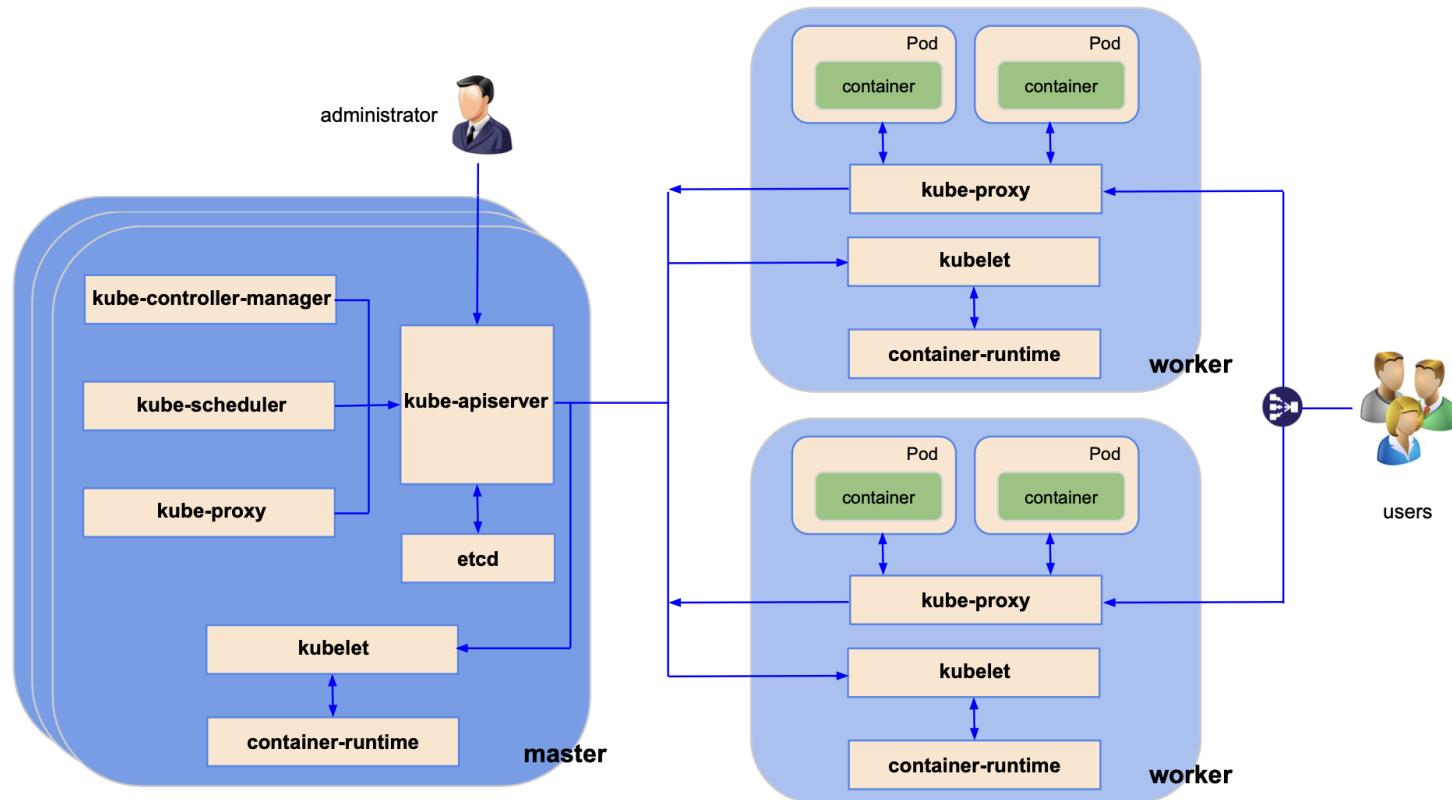
Architecture de Kubernetes

2. Composants de Kubernetes

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

Architecture de Kubernetes

Les processus : vision d'ensemble



Architecture de Kubernetes

3. Masters vs Workers

1. Master (Maître):

- Gère le cluster.
- Prend des décisions globales (par exemple, la planification).
- Déetecte les événements du cluster (par exemple, un conteneur qui a échoué).

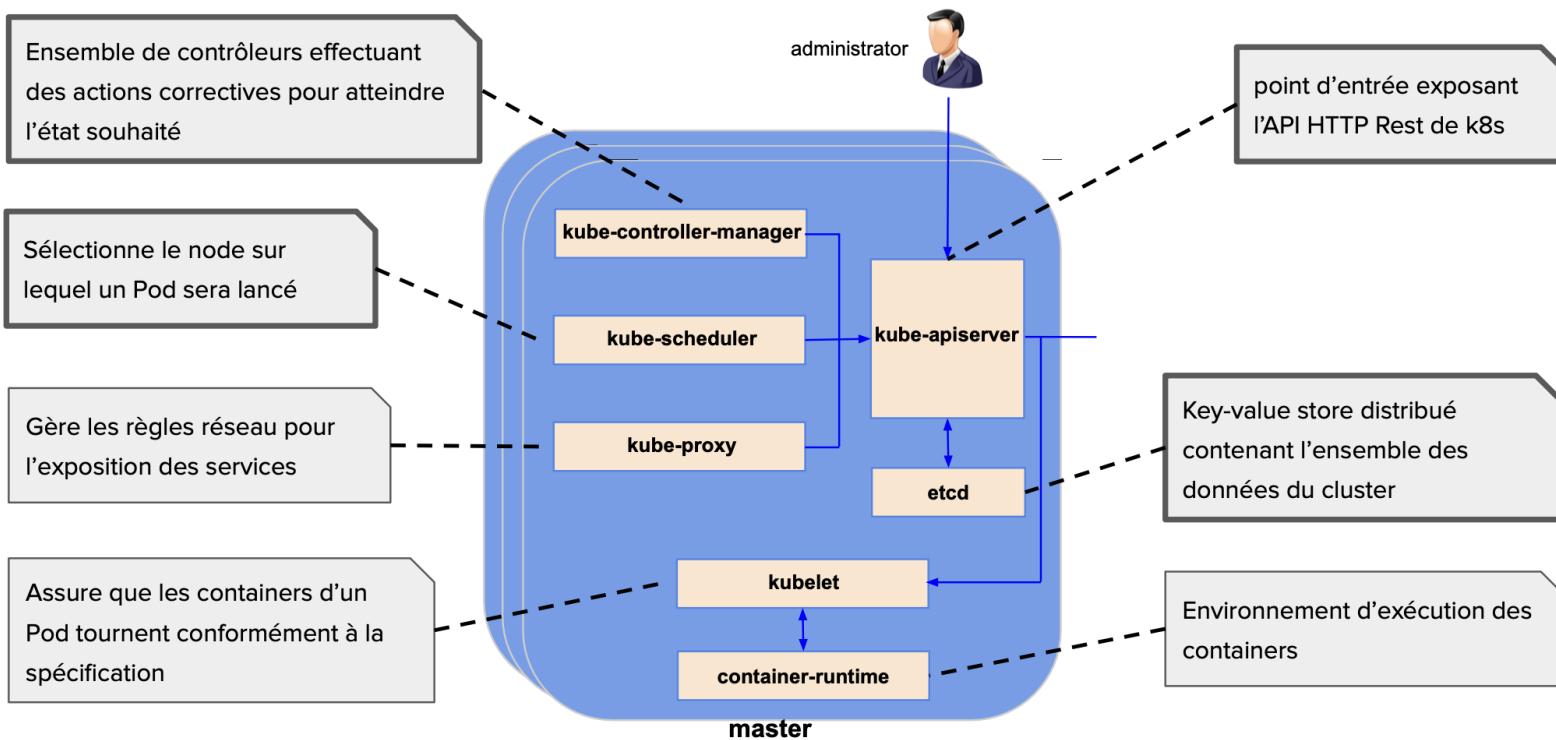
○ *Composants typiques d'un noeud master: API Server, etcd, Scheduler, et autres composants de contrôle.*

2. Worker (Esclave):

- Exécute les conteneurs.
- Rapporte à master.
- Chaque worker est équipé de Docker (ou une autre solution conteneur), kubelet, et kube-proxy.

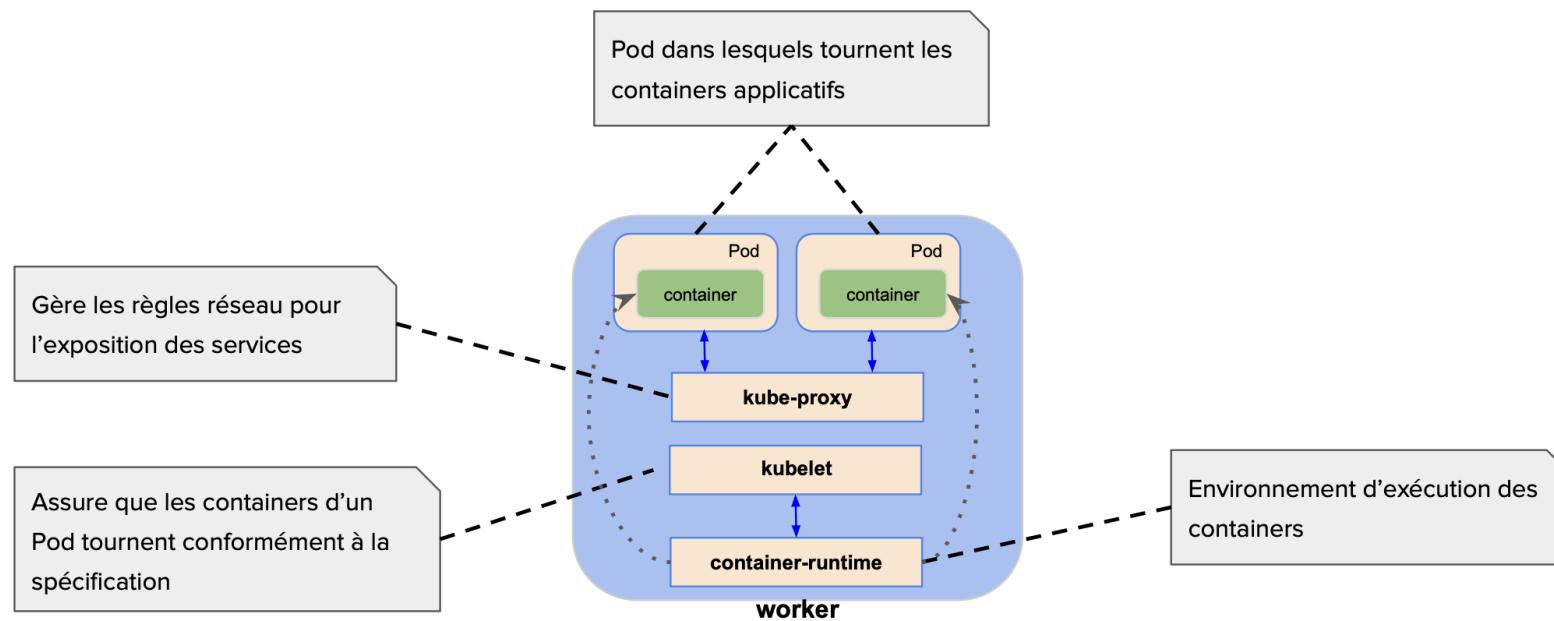
Architecture de Kubernetes

Les processus : côté Master



Architecture de Kubernetes

Les processus : côté Worker



Architecture de Kubernetes

4. Couche réseau

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



Créer un cluster Kubernetes

Créer un cluster Kubernetes

Solutions pour Créer un Cluster Kubernetes :

Clouds Publics :

- AWS EKS
- Azure AKS
- Google GKE

Distributions Tiers :

- OpenShift (Red Hat)
- Rancher

Environnement de développement :

- Minikube
- Kind

Créer un cluster Kubernetes

1. Clouds Publics

AWS EKS (Elastic Kubernetes Service)

- **Présentation :** AWS EKS est un service géré qui permet d'exécuter Kubernetes sur AWS sans nécessiter l'installation et la maintenance du plan de contrôle de Kubernetes.
- **Avantages pour les environnements de développement :**
 - **Intégration avec AWS :** Accès facile aux autres services AWS (S3, RDS, etc.).
 - **Gestion simplifiée :** AWS gère la maintenance du plan de contrôle Kubernetes.
 - **Évolutivité et sécurité :** Haute disponibilité, sécurité et conformité intégrées.

Azure AKS (Azure Kubernetes Service)

- **Présentation :** Azure AKS est un service Kubernetes géré qui simplifie le déploiement, la gestion et les opérations de Kubernetes sur Azure.
- **Avantages pour les environnements de développement :**
 - **Intégration avec l'écosystème Azure :** Accès aux services Azure comme Azure Functions et Azure Logic Apps.
 - **Outils de développement :** Support de Visual Studio Code pour Kubernetes.
 - **Monitoring et gestion :** Outils intégrés pour la surveillance et la gestion du cluster.

Créer un cluster Kubernetes

1. Clouds Publics

Google GKE (Google Kubernetes Engine)

- **Présentation :** Google GKE est un service Kubernetes géré et sécurisé qui automatise la gestion et la maintenance de vos clusters Kubernetes.
- **Avantages pour les environnements de développement :**
 - **Expertise Kubernetes de Google :** Google est le créateur initial de Kubernetes, donc GKE bénéficie de cette expertise.
 - **Intégration avec Google Cloud :** Accès aux services de Google Cloud Platform.
 - **Auto-scaling et Load Balancing :** Gestion automatisée de la charge et de la taille du cluster.

Créer un cluster Kubernetes

2. Distributions Tiers

OpenShift (Red Hat)

- **Présentation :** OpenShift est une plateforme de conteneurs basée sur Kubernetes qui offre une solution d'entreprise pour le développement d'applications cloud-native.
- **Avantages pour les environnements de développement :**
 - **Sécurité renforcée :** Sécurité et conformité pour les entreprises.
 - **CI/CD intégré :** Intégration continue et déploiement continu intégrés.
 - **Support multi-cloud et hybride :** Peut être déployé sur divers environnements cloud ou on-premise.

Rancher

- **Présentation :** Rancher est une plateforme complète de gestion de conteneurs qui facilite la gestion de Kubernetes.
- **Avantages pour les environnements de développement :**
 - **Gestion multi-cluster :** Facilite la gestion de multiples clusters Kubernetes.
 - **Interface utilisateur intuitive :** Interface graphique pour une gestion simplifiée.
 - **Large compatibilité :** Supporte plusieurs distributions Kubernetes.

Créer un cluster Kubernetes

3. Environnement de développement

Minikube :

- **Présentation :** Minikube est un outil qui permet de lancer un cluster Kubernetes local sur votre machine, idéal pour le développement et les tests.
- **Avantages pour les environnements de développement :**
 - **Facilité de mise en place :** Simple à installer et à configurer.
 - **Idéal pour l'apprentissage :** Excellent pour expérimenter et apprendre Kubernetes.
 - **Environnement local :** Permet de tester des applications Kubernetes localement sans coûts de cloud.

Kind (*Kubernetes IN Docker*):

- **Présentation :** Kind est un outil qui lance des clusters Kubernetes locaux en utilisant Docker, conçu principalement pour le test de Kubernetes, mais également utile pour le développement local et l'intégration continue.
- **Avantages pour les environnements de développement :**
 - **Facile à Installer et Configurer :** Permet un démarrage rapide et facile de clusters Kubernetes.
Idéal pour l'Apprentissage : Excellent pour ceux qui débutent avec Kubernetes, offrant un environnement pratique pour l'expérimentation.
 - **Environnement Local Sans Coût Cloud :** Exécute Kubernetes localement dans des conteneurs Docker, évitant les coûts du cloud.

Créer un cluster Kubernetes

Commandes de Base pour Minikube

1. Installer Minikube

- Sur les systèmes basés sur Linux, Windows ou macOS, suivez les instructions spécifiques à votre système d'exploitation sur le [site officiel de Minikube](#).

2. Démarrer Minikube

```
minikube start
```

Cette commande démarre un cluster Kubernetes local. Vous pouvez spécifier le pilote (comme VirtualBox, Hyperkit, etc.) avec l'option `--driver=<driver_name>`.

3. Vérifier l'État de Minikube

```
minikube status
```

4. Accéder au Dashboard Kubernetes

```
minikube dashboard
```

Cette commande ouvre le tableau de bord Kubernetes dans votre navigateur par défaut, offrant une interface graphique pour gérer vos clusters.

1. Arrêter Minikube

```
minikube stop
```

Utilisez cette commande pour arrêter le cluster Minikube.

2. Supprimer le Cluster Minikube

```
minikube delete
```

Cette commande supprime le cluster Minikube et libère toutes les ressources associées.

3. Exécuter des Commandes Kubernetes

Vous pouvez utiliser `kubectl` pour interagir avec votre cluster Minikube. Si `kubectl` n'est pas installé, Minikube peut le configurer pour vous avec :

```
minikube kubectl -- <command>
```

Par exemple, pour obtenir des informations sur les nœuds du cluster :

```
minikube kubectl -- get nodes
```

Créer un cluster Kubernetes

Commandes et options de base

1. Concepts de base de Kubernetes

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

2. Kubernetes API

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

Créer un cluster Kubernetes

Introduction à kubectl

3. Outil `kubectl`

- `kubectl` est l'outil en ligne de commande pour interagir avec le cluster Kubernetes. Il utilise l'API Kubernetes pour communiquer avec le cluster.

- **Commandes de base:**

- `kubectl get`: Affiche une ou plusieurs ressources.
- `kubectl describe`: Montre les détails d'une ressource spécifique.
- `kubectl create`: Crée une ressource.
- `kubectl delete`: Supprime des ressources.
- `kubectl apply`: Applique une configuration à une ressource.

Créer un cluster Kubernetes

Contextes et namespaces

Namespace

1. Définition :

- Un namespace dans Kubernetes est une sorte de segmentation logique du cluster. Cela permet de diviser les ressources du cluster entre plusieurs utilisateurs ou groupes d'utilisateurs.

2. Utilisations :

- **Isolation des ressources** : Chaque namespace peut contenir un ensemble de ressources (comme les pods, les services, les déploiements, etc.), et ces ressources sont isolées des autres namespaces.
- **Gestion des quotas** : Les administrateurs peuvent définir des quotas de ressources pour chaque namespace, limitant ainsi l'utilisation des ressources du cluster.
- **Contrôle d'accès** : Les namespaces permettent d'appliquer des politiques de contrôle d'accès plus fines en attribuant des rôles et des autorisations spécifiques au niveau du namespace.

Contexte

1. Définition :

- Dans Kubernetes, un contexte est une combinaison de trois éléments : le cluster, l'espace de noms (namespace) et les informations d'identification de l'utilisateur. Il est utilisé par `kubectl`, l'outil en ligne de commande pour Kubernetes, pour communiquer avec un cluster spécifique.

2. Utilisations :

- **Sélection du Cluster** : Permet de basculer facilement entre différents clusters. C'est utile si vous gérez plusieurs clusters Kubernetes.
- **Sélection du Namespace** : Définit le namespace par défaut pour les opérations en cours.

3. Configuration :

- Gérée dans le fichier de configuration `kubectl`, souvent `~/.kube/config`.
- Vous pouvez avoir plusieurs contextes dans un seul fichier de configuration et basculer entre eux en utilisant `kubectl config use-context <nom-du-contexte>`.



Déployer sa première application

Déployer sa première application

1. Pod

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

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

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

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

Déployer sa première application

2. Deployment

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

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

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

Déployer sa première application

Introduction aux fichiers de manifestes

```
# Définition de la version de l'API Kubernetes utilisée
apiVersion: v1

# Type de ressource à créer
kind: Pod

# Métagdonnées du pod
metadata:
  # Nom du pod
  name: mon-pod-nginx

# Spécifications du pod
spec:
  # Définition des conteneurs à exécuter dans le pod
  containers:
    # Liste des conteneurs (ici, un seul conteneur)
    - name: nginx
      # Image à utiliser pour le conteneur
      image: nginx

      # (Optionnel) Ports à exposer sur le conteneur
      # ports:
      # - containerPort: 80
```

1. Labels :

- Ce sont des paires clé-valeur attachées aux objets pour organiser et sélectionner des sous-ensembles de ressources. Les labels sont utilisés pour grouper et filtrer des ressources selon des critères spécifiques.

2. Annotations :

- Semblables aux labels, mais généralement utilisées pour stocker des informations supplémentaires qui peuvent aider à décrire ou configurer une ressource. Les annotations ne sont pas destinées à identifier et sélectionner des objets.

3. Specs (Spécifications) :

- Il s'agit de la description détaillée de la configuration souhaitée pour la ressource. Les **specs** définissent ce que vous voulez que votre ressource soit, y compris ses comportements, états et autres détails spécifiques.



Suivi et diagnostic d'un pod en fonctionnement

Suivi et diagnostic d'un pod en fonctionnement

1. Lister les pods et afficher les détails:

- **Commande Kubernetes:** `kubectl get pods`
 - Cette commande affiche une liste de tous les pods dans le namespace courant.
 - Pour voir plus de détails sur un pod spécifique, utilisez `kubectl describe pod <nom_du_pod>`.

2. Accès aux logs:

- **Commande Kubernetes:** `kubectl logs <nom_du_pod>`
 - Cette commande récupère les logs standard d'un conteneur dans le pod. Si le pod a plusieurs conteneurs, spécifiez le conteneur avec l'option `-c <nom_du_conteneur>`.

3. Exécuter des commandes à l'intérieur du pod:

- **Commande Kubernetes:** `kubectl exec`
 - Pour exécuter une commande dans un conteneur spécifique d'un pod, utilisez `kubectl exec <nom_du_pod> -- <commande>`. Par exemple, `kubectl exec <nom_du_pod> -- ls /` pour lister le contenu du répertoire racine.
 - Pour obtenir un shell interactif, utilisez `kubectl exec -it <nom_du_pod> -- /bin/bash`.

4. Échanger des fichiers avec le pod:

- **Commande Kubernetes:** `kubectl cp`
 - Pour copier des fichiers du pod vers votre système local, utilisez



Accéder à ses applications

Accéder à ses applications

Accéder à ses applications dans Kubernetes :

- Dans un cluster Kubernetes, les applications sont généralement déployées en tant que "pods".
- Pour y accéder, on utilise des "**services**" qui agissent comme des points d'entrée stables pour l'accès réseau.
- Un "**Ingress**" peut également être utilisé pour gérer l'accès externe, permettant de définir des règles de routage plus avancées et de fournir des fonctionnalités telles que SSL/TLS.

Le réseau virtuel de Kubernetes :

- Chaque pod dans un cluster **Kubernetes** reçoit sa propre adresse IP, créant un réseau plat où tous les pods peuvent communiquer entre eux.
- Cette configuration simplifie la communication inter-pods et le découplage des applications des adresses IP physiques.
- Le réseau est implémenté à l'aide de plugins CNI, qui définissent comment les différents éléments du réseau Kubernetes communiquent.

Accéder à ses applications

Service

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

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

Accéder à ses applications

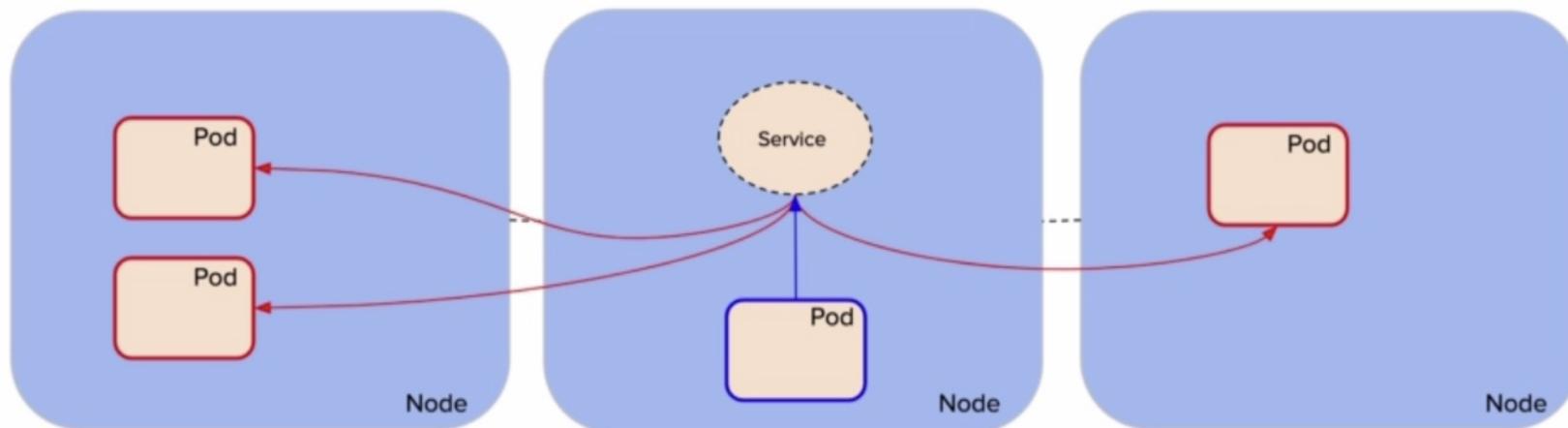
ClusterIP: Expose le service sur une adresse IP interne dans le cluster, le rendant accessible uniquement de l'intérieur du cluster.

NodePort: Expose le service sur chaque nœud du cluster à un port statique (NodePort). Accessible de l'extérieur du cluster en utilisant :.

LoadBalancer: Expose le service en utilisant l'équilibrer de charge de votre fournisseur de cloud, permettant un accès externe direct.

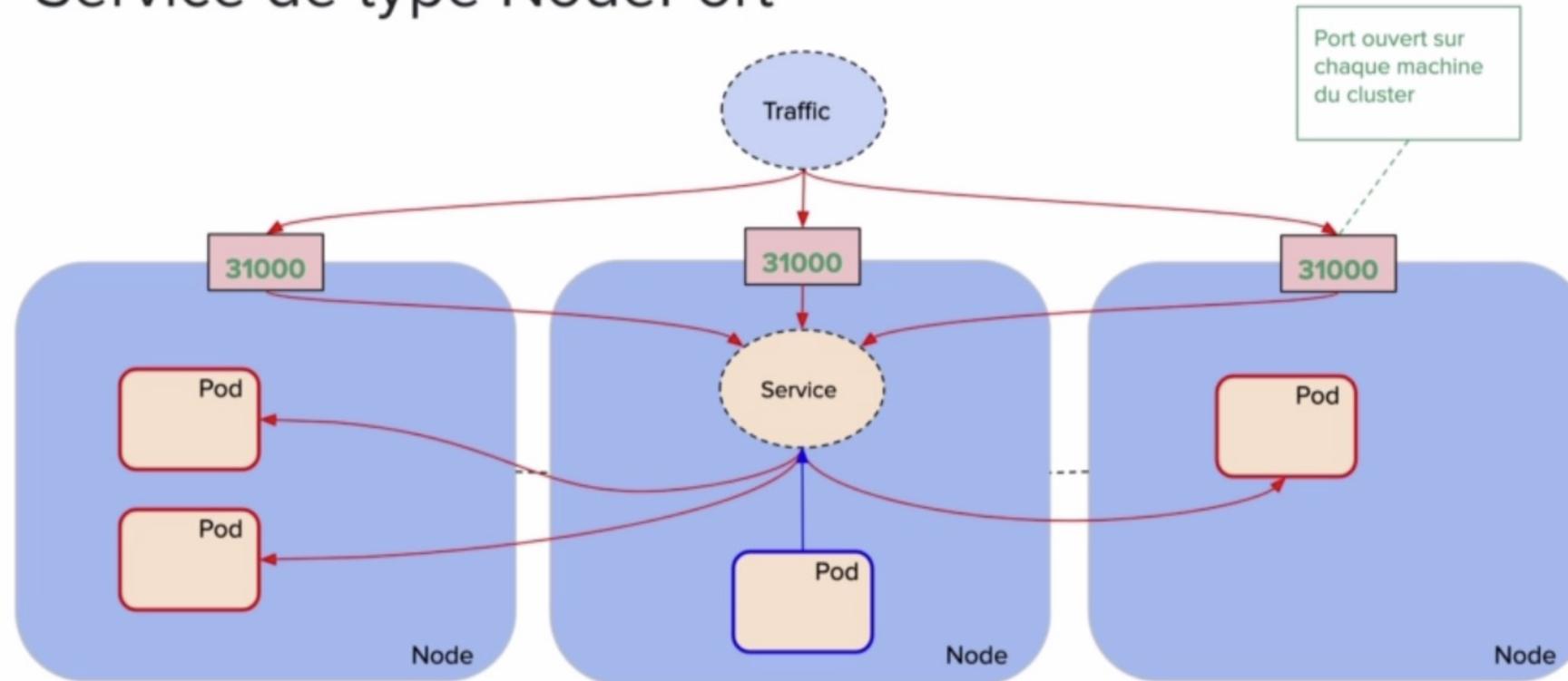
Accéder à ses applications

Service de type ClusterIP



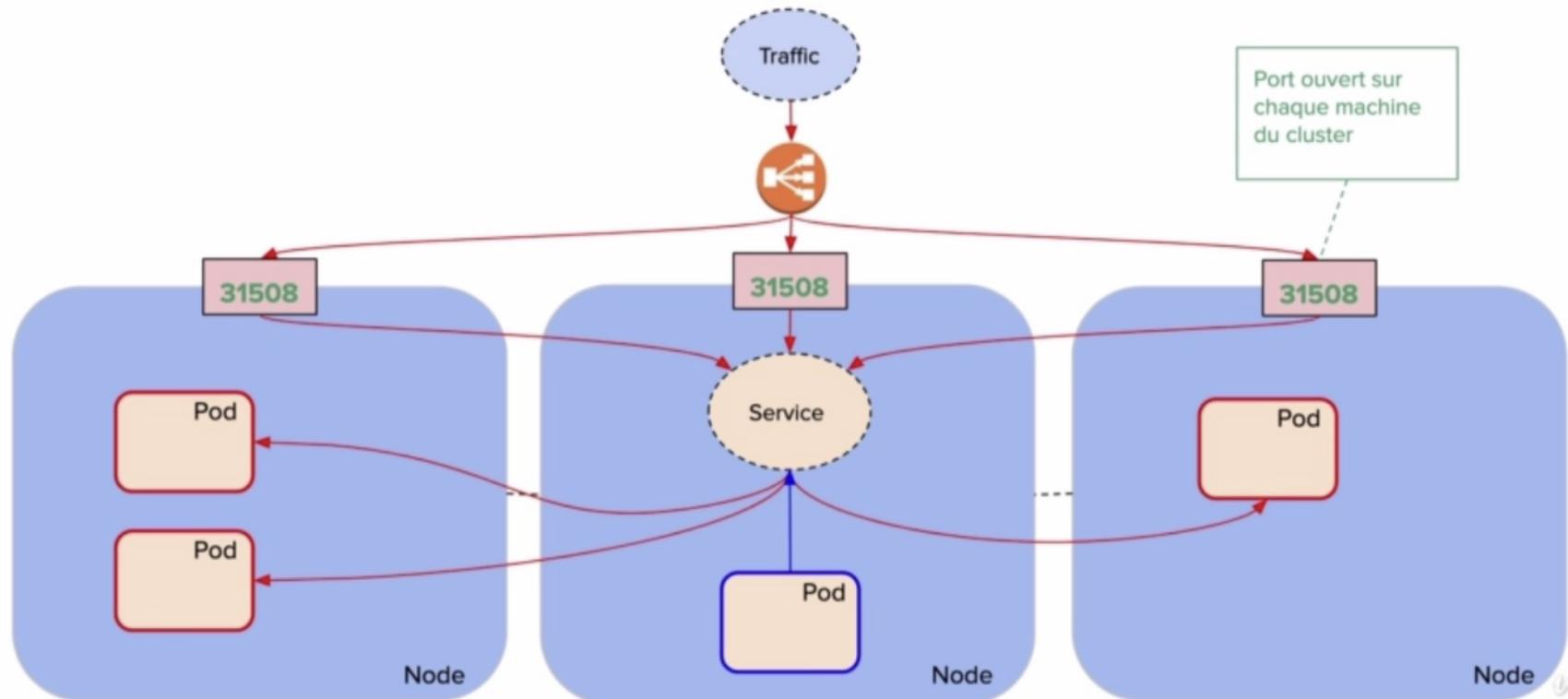
Accéder à ses applications

Service de type NodePort



Accéder à ses applications

Service de type LoadBalancer



Accéder à ses applications

Principe de CNI (Container Network Interface)

Le Container Network Interface (CNI) est une spécification standardisée pour la gestion du réseau dans les conteneurs. Elle est particulièrement pertinente dans le contexte de Kubernetes pour plusieurs raisons :

- **Standardisation** : CNI définit une interface commune entre différents plugins de réseau, permettant ainsi une grande variété de solutions de réseautage qui peuvent être utilisées avec Kubernetes ou d'autres orchestrateurs de conteneurs.
- **Flexibilité** : Grâce à cette standardisation, les utilisateurs de Kubernetes peuvent choisir parmi de nombreux plugins CNI selon leurs besoins spécifiques, sans être liés à une implémentation spécifique.
- **Simplicité d'intégration** : CNI est conçu pour être simple à intégrer et à utiliser. Il se concentre uniquement sur la connexion des interfaces réseau aux conteneurs.

Accéder à ses applications

Flannel

Flannel est un plugin CNI très populaire dans la communauté Kubernetes, principalement en raison de sa simplicité et de sa facilité de configuration. Voici ses caractéristiques principales :

- **Réseau Overlay** : Flannel crée un réseau overlay qui permet à chaque pod d'avoir une adresse IP unique. Cela signifie que tous les pods peuvent communiquer entre eux comme s'ils étaient sur le même réseau, indépendamment des nœuds sur lesquels ils sont exécutés.
- **Simplicité** : Flannel est conçu pour être simple à déployer et à gérer, ce qui le rend idéal pour des scénarios plus simples ou pour ceux qui débutent avec Kubernetes.
- **Support des backends multiples** : Flannel peut utiliser différents backends (comme VXLAN, UDP) pour créer le réseau overlay, offrant ainsi une certaine flexibilité.

Accéder à ses applications

Calico

Calico, d'autre part, est un plugin CNI offrant des fonctionnalités de réseau et de sécurité avancées :

- **Contrôle de sécurité réseau** : Calico fournit des fonctionnalités de sécurité réseau avancées, y compris le contrôle d'accès au niveau des pods grâce à des politiques réseau fines.
- **Performance et Efficacité** : Contrairement à Flannel, Calico n'utilise pas de réseau overlay par défaut, ce qui peut offrir de meilleures performances et moins de surcharge de réseau.
- **Segmentation réseau** : Calico supporte la segmentation réseau pour améliorer la sécurité et l'efficacité du réseau. Cela permet une isolation plus forte entre différents groupes de pods.
- **Compatibilité étendue** : Calico supporte une gamme étendue de plateformes et d'environnements, ce qui le rend adapté à des déploiements plus complexes ou à grande échelle.



RéPLICATION DES PODS

RéPLICATION DES PODS

La réPLICATION DES PODS dans Kubernetes permet d'assurer la disponibilité, la montée en charge et la résilience des applications.

Objectifs de la RéPLICATION

1. **Redondance** : Avoir plusieurs copies d'un pod assure que l'application reste disponible même si certains pods échouent.
2. **Passage à l'échelle (Scaling)** : La réPLICATION permet d'augmenter ou de diminuer le nombre de pods pour répondre à la demande variable.
3. **Sharding** : Répartir les charges de travail sur plusieurs pods peut améliorer les performances et l'utilisation des ressources.

RéPLICATION DES PODS

1. Utilisation d'un ReplicaSet

Un ReplicaSet garantit qu'un nombre spécifié de répliques de pod est toujours en cours d'exécution.

Voici un exemple de fichier YAML pour créer un ReplicaSet :

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: mon-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mon-appli
  template:
    metadata:
      labels:
        app: mon-appli
    spec:
      containers:
        - name: mon-conteneur
          image: monimage
```

- Un ReplicaSet est un objet Kubernetes qui définit le nombre désiré de répliques (copies) d'un pod.
- Il veille à ce que le nombre spécifié de répliques de pod soit toujours en cours d'exécution.
- Les ReplicaSets peuvent être créés et modifiés via des fichiers YAML ou des commandes kubectl.

RéPLICATION DES PODS

2. Principe de Boucle de Réconciliation

- Dans Kubernetes, la boucle de réconciliation est un processus continu où le contrôleur compare l'état actuel du cluster à l'état désiré (défini par l'utilisateur) et effectue des actions pour aligner les deux. Si un pod tombe en panne, le contrôleur le détecte et en crée un nouveau pour maintenir l'état désiré.
- Ce n'est pas quelque chose que vous implémentez directement, mais plutôt un principe sur lequel Kubernetes fonctionne. Le contrôleur surveille en permanence l'état des pods et prend des mesures pour corriger toute divergence par rapport à l'état désiré.

3. Passage à l'Échelle Horizontal et Vertical

1. Passage à l'échelle horizontal : Augmente ou diminue le nombre de pods pour s'adapter à la charge. Cela n'affecte pas les capacités de chaque pod mais modifie le nombre total de pods.

2. Passage à l'échelle vertical : Implique d'augmenter ou de diminuer les ressources (CPU, mémoire) allouées à chaque pod. Cela ne modifie pas le nombre de pods mais leur capacité individuelle.

- **Horizontal** : Utilisation de l'Horizontal Pod Autoscaler (HPA). Exemple de commande pour créer un HPA :

```
kubectl autoscale rs mon-replicaset --min=2 --max=5 --cpu-percent=80
```

Cela ajuste automatiquement le nombre de pods dans `mon-replicaset` entre 2 et 5 en fonction de l'utilisation du CPU.

- **Vertical** : Mise à l'échelle des ressources d'un pod. Cela se fait généralement en mettant à jour la définition du pod ou du deployment pour changer les limites de CPU/mémoire.

RéPLICATION DES PODS

5. DaemonSet

Un DaemonSet s'assure qu'un pod spécifique s'exécute sur tous les nœuds. Voici un exemple de fichier YAML pour un DaemonSet:

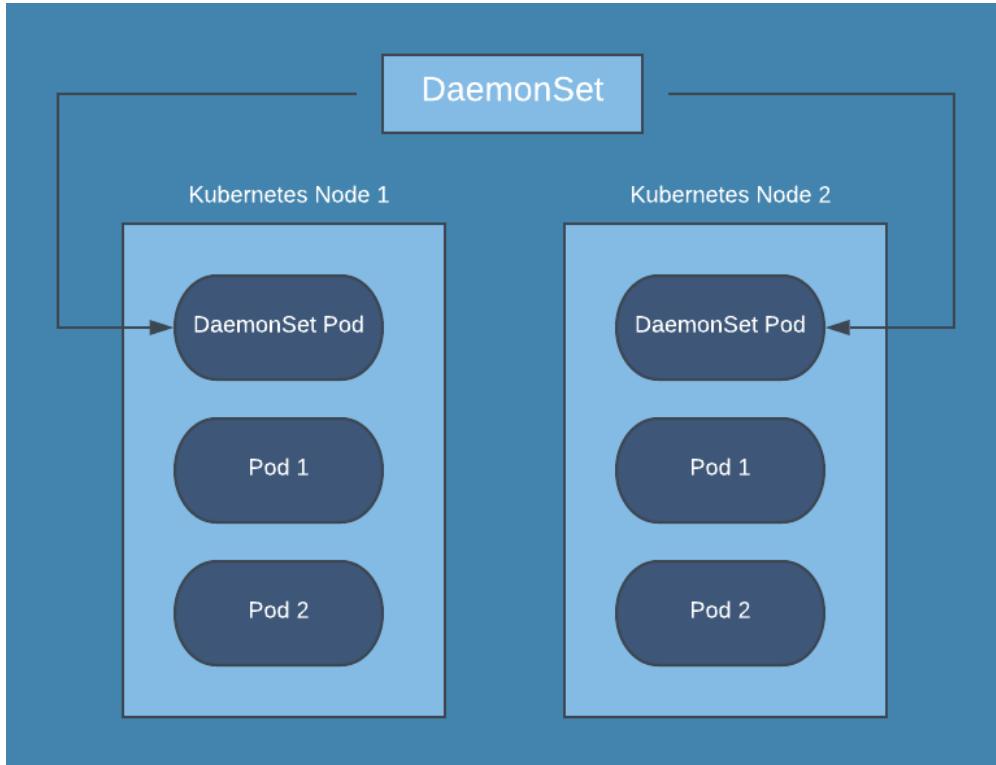
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: mon-daemonset
spec:
  selector:
    matchLabels:
      name: mon-daemon
  template:
    metadata:
      labels:
        name: mon-daemon
    spec:
      containers:
        - name: mon-daemon
          image: monimage-daemon
```

- Un DaemonSet assure qu'une copie de pod spécifique fonctionne sur tous (ou certains) nœuds du cluster.
- C'est utile pour des tâches comme la collecte de logs, la surveillance du réseau, ou d'autres services qui doivent être exécutés sur chaque nœud.

Ce DaemonSet déployera le pod `mon-daemon` sur chaque nœud du cluster.

RéPLICATION DES PODS

- **DaemonSet**



```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```



Gérer le cycle de vie d'une application

Gérer le cycle de vie d'une application

Principe de Rolling Update

Un rolling update est une méthode pour mettre à jour les pods d'un déploiement avec une perturbation minimale. Kubernetes remplace progressivement les anciennes versions des pods par de nouvelles, en s'assurant qu'un certain nombre de pods sont opérationnels à tout moment.

Exemple de mise en œuvre :

1. Vous avez un Deployment avec une certaine version d'une application.
2. Pour mettre à jour, vous modifiez l'image du conteneur dans la spécification du Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: mon-appli
  template:
    metadata:
      labels:
        app: mon-appli
    spec:
      containers:
        - name: mon-conteneur
          image: monimage:v2
```

Dans cet exemple, `maxUnavailable` définit le nombre maximal de pods indisponibles pendant la mise à jour, et `maxSurge` définit le nombre maximal de pods supplémentaires créés.

Gérer le cycle de vie d'une application

Créer et Gérer des Deployments

Un Deployment dans Kubernetes gère des ensembles de pods répliqués. Il permet des mises à jour déclaratives pour les Pods et les ReplicaSets.

Exemple de création d'un Deployment :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mon-appli
  template:
    metadata:
      labels:
        app: mon-appli
    spec:
      containers:
        - name: mon-conteneur
          image: monimage:v1
```

Pour modifier un Deployment, vous pouvez changer la spécification dans le fichier YAML et appliquer la mise à jour avec **kubectl apply -f**.

Gérer le cycle de vie d'une application

Stratégies de Déploiement

Kubernetes supporte plusieurs stratégies de déploiement :

1. **RollingUpdate** (expliqué ci-dessus) : Met à jour les pods de manière incrémentielle.
 2. **Recreate** : Supprime tous les anciens pods avant de créer de nouveaux. Cela entraîne une période d'indisponibilité mais assure que les nouvelles instances ne coexistent pas avec les anciennes.
- Pour consulter l'historique des déploiements :

```
bash kubectl rollout history deployment/<nom-du-déploiement>
```

- Revenir à une révision précédente :
 - Si un déploiement pose problème, il est possible de revenir à une révision précédente :

```
bash kubectl rollout undo deployment/<nom-du-déploiement> --to-revision=<numéro-de-révision>
```

Exemple de stratégie Recreate :

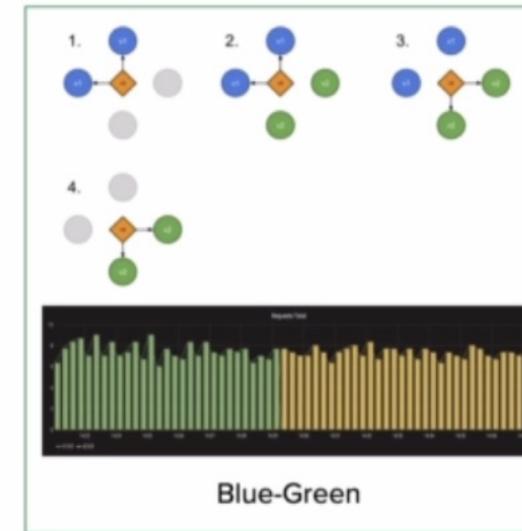
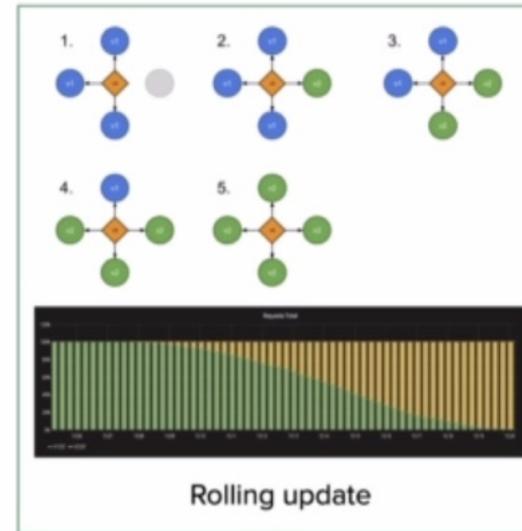
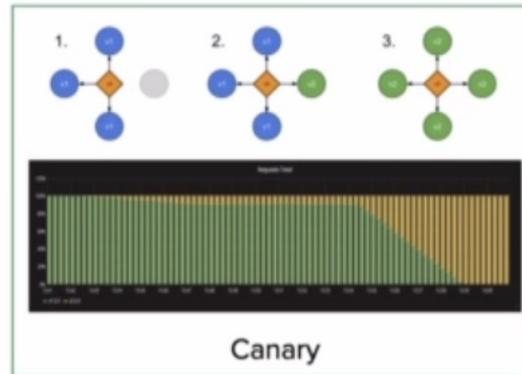
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: mon-appli
  template:
    metadata:
      labels:
        app: mon-appli
    spec:
      containers:
        - name: mon-conteneur
          image: monimage:v2
```

Dans ce cas, le déploiement supprimera tous les anciens pods avant de déployer les nouveaux avec la nouvelle version de l'image.

Gérer le cycle de vie d'une application

Stratégies de Déploiement

Mise à jour d'une application (général)



Options disponibles
dans Kubernetes

Images <https://blog.container-solutions.co>



Lancer des tâches à exécution unique

Lancer des tâches à exécution unique

Principe et Intérêt des Jobs

Un Job dans Kubernetes crée un ou plusieurs pods et s'assure qu'un certain nombre de ces pods se terminent avec succès. Une fois les pods terminés avec succès, le Job est considéré comme terminé. Les Jobs sont utiles pour des tâches à exécution unique comme les traitements par lots, les tâches de maintenance, ou les opérations de calcul ponctuelles.

Lancer des Jobs

Pour lancer un Job, vous définissez un objet Job dans un fichier YAML. Voici un exemple :

```
apiVersion: batch/v1
kind: Job
metadata:
  name: mon-job
spec:
  template:
    spec:
      containers:
        - name: mon-conteneur
          image: monimage
          command: ["sh", "-c", "echo Hello Kubernetes! && sleep 30"]
  restartPolicy: Never
```

Dans cet exemple, le Job lance un pod qui exécute une simple commande shell (`echo Hello Kubernetes! && sleep 30`). Le `restartPolicy: Never` indique que le pod ne doit pas être redémarré une fois qu'il a terminé son exécution.

Lancer des tâches à exécution unique

Parallélisme et Work-Pools

Kubernetes permet de gérer le parallélisme dans les Jobs pour exécuter plusieurs pods en parallèle. Cela est utile pour accélérer le traitement des tâches dans les work-pools ou pour diviser une grande tâche en plus petites parties exécutées simultanément.

Exemple de Job avec parallélisme :

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-parallel
spec:
  parallelism: 3
  completions: 3
  template:
    spec:
      containers:
        - name: worker
          image: monimage
          command: ["sh", "-c", "echo Traitement en parallèle && sleep 20"]
  restartPolicy: OnFailure
```

Dans cet exemple :

- `parallelism: 3` indique que trois pods peuvent être exécutés en parallèle.
- `completions: 3` signifie que le Job doit s'assurer que trois pods se terminent avec succès pour considérer le Job comme complet.

Ces pods exécuteront la même tâche de manière parallèle, ce qui est utile pour les scénarios de traitement par lots où les tâches sont indépendantes les unes des autres.



Déploiement et partage des éléments de configuration

Déploiement et partage des éléments de configuration

Le déploiement et le partage des éléments de configuration dans Kubernetes sont essentiels pour gérer les applications de manière flexible et sécurisée. Kubernetes utilise des ConfigMaps et des Secrets pour ce faire.

Principe de Généricité et d'Indépendance

Le principe clé ici est de séparer la configuration de l'application de celle de la plateforme. Cela signifie que votre application ne dépend pas directement des configurations spécifiques à l'environnement dans lequel elle s'exécute. Ainsi, vous pouvez déployer la même application dans différents environnements (développement, test, production) sans modifications du code source.

Déploiement et partage des éléments de configuration

Créer des ConfigMaps

Un ConfigMap est utilisé pour stocker des informations de configuration non confidentielles sous forme de paires clé-valeur. Ces informations peuvent être utilisées par les pods Kubernetes.

Exemple de création d'un ConfigMap :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mon-configmap
data:
  param1: valeur1
  param2: valeur2
```

Ce ConfigMap peut ensuite être utilisé par les pods pour récupérer les valeurs de configuration `param1` et `param2`.

Déploiement et partage des éléments de configuration

Différents Moyens de Consommation des ConfigMaps

Les ConfigMaps peuvent être utilisés de plusieurs manières dans les pods :

1. Comme variables d'environnement :

```
env:  
  - name: PARAM1  
    valueFrom:  
      configMapKeyRef:  
        name: mon-configmap  
        key: param1
```

2. Comme fichiers dans un volume :

```
volumes:  
  - name: config-volume  
    configMap:  
      name: mon-configmap  
containers:  
  - name: mon-conteneur  
    volumeMounts:  
      - name: config-volume  
        mountPath: /chemin/de/montage
```

Déploiement et partage des éléments de configuration

Créer et Consommer des Secrets

Les Secrets sont utilisés pour stocker et gérer des informations sensibles, comme des mots de passe, des tokens OAuth, ou des clés SSH.

Exemple de création d'un Secret :

```
apiVersion: v1
kind: Secret
metadata:
  name: mon-secret
type: Opaque
data:
  cle: c2VjcmV0dmFsdWU= # "secretvalue" encodé en base64
```

Consommation de Secrets

Les Secrets peuvent être utilisés de la même manière que les ConfigMaps :

1. Comme variables d'environnement :

```
env:
  - name: CLE
    valueFrom:
      secretKeyRef:
        name: mon-secret
        key: cle
```

2. Comme fichiers dans un volume :

```
volumes:
  - name: secret-volume
    secret:
      secretName: mon-secret
containers:
  - name: mon-conteneur
    volumeMounts:
      - name: secret-volume
        mountPath: /chemin/secret
```



Gérer les données persistantes et les applications stateful

Gérer les données persistantes et les applications stateful

Problématique du Stockage Persistant dans le Cloud

Dans les environnements cloud, la persistance des données pose un défi car les conteneurs sont éphémères par nature. Si un conteneur est arrêté ou redémarré, toutes les données stockées à l'intérieur sont perdues. Pour résoudre ce problème, Kubernetes utilise des volumes pour fournir un stockage persistant.

Introduction aux Volumes

Un volume dans Kubernetes est une abstraction qui permet de stocker des données et de les rendre accessibles au pod, indépendamment du cycle de vie du conteneur. Il existe plusieurs types de volumes :

1. **emptyDir** : Un volume temporaire qui est créé lorsqu'un pod est assigné à un nœud et existe tant que ce pod s'exécute sur ce nœud.
2. **hostPath** : Montre un fichier ou un répertoire du système de fichiers de l'hôte vers un pod.
3. **nfs** : Montre un système de fichiers NFS partagé.
4. **persistentVolumeClaim (PVC)** : Permet aux utilisateurs de réclamer un stockage abstrait (PersistentVolume) dans un cluster.
5. **configMap, secret** : Utilisés pour exposer des données de configuration et des secrets comme des volumes à l'intérieur d'un pod.
6. **cloudProvider-specific storage** : Comme AWS EBS, Azure Disk, ou Google Persistent Disk.

Gérer les données persistantes et les applications stateful

Créer une Application en Exploitant un Volume `emptyDir`

Voici un exemple de déploiement d'une application qui utilise un volume `emptyDir` :

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod
spec:
  containers:
    - name: mon-conteneur
      image: monimage
      volumeMounts:
        - mountPath: /mon/volume
          name: mon-volume
  volumes:
    - name: mon-volume
      emptyDir: {}
```

Dans cet exemple, un volume `emptyDir` est créé et monté dans le conteneur sous `/mon/volume`. Toutes les données

Monter un ConfigMap en tant que Volume

Vous pouvez monter un ConfigMap comme volume pour fournir des fichiers de configuration au pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod-configmap
spec:
  containers:
    - name: mon-conteneur
      image: monimage
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: mon-configmap
```

Ici, `mon-configmap` est monté dans le conteneur sous `/etc/config`.

Gérer les données persistantes et les applications stateful

Utiliser du Stockage Distant

1. Définir un PersistentVolume :

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mon-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

2. Créer un PersistentVolumeClaim :

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

3. Utiliser le PVC dans un pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod-pvc
spec:
  containers:
    - name: mon-conteneur
      image: monimage
      volumeMounts:
        - mountPath: "/mon/data"
          name: mon-volume
  volumes:
    - name: mon-volume
      persistentVolumeClaim:
        claimName: mon-pvc
```



Sécurité dans Kubernetes

Sécurité dans Kubernetes

La sécurité dans Kubernetes est un aspect essentiel pour protéger les ressources et les données. Elle s'étend de la gestion des accès et permissions (RBAC) jusqu'à la sécurité au niveau du réseau (NetworkPolicies) et des politiques de sécurité des pods (PodSecurityPolicies).

Introduction et Manipulation de l'API RBAC (Role-Based Access Control)

RBAC (Role-Based Access Control) est un mécanisme pour réguler l'accès aux ressources informatiques en Kubernetes basé sur les rôles des utilisateurs individuels au sein de votre organisation.

Sécurité dans Kubernetes

Exemple de création d'un Role et d'une RoleBinding :

1. Créer un Role :

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

Ce Role permet à l'utilisateur de lire les informations des pods dans le namespace **default**.

2. Créer une RoleBinding :

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: utilisateur1
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Cette RoleBinding lie le Role **pod-reader** à l'utilisateur **utilisateur1**.

Sécurité dans Kubernetes

Introduction aux NetworkPolicies

Les NetworkPolicies permettent de contrôler le trafic réseau vers et depuis les pods dans un cluster Kubernetes.

Exemple de NetworkPolicy :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy-example
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
  ports:
  - protocol: TCP
    port: 3306
```

Cette politique autorise les pods avec le label `role: frontend` à se connecter aux pods avec le label `role: db` sur le port TCP 3306.

Sécurité dans Kubernetes

Cloisonner une Application avec une NetworkPolicy

Pour cloisonner une application, vous créez une NetworkPolicy qui définit les règles de trafic entrant et/ou sortant pour les pods de cette application.

Exemple de cloisonnement :

Imaginons une application avec un frontend et un backend. Vous pouvez définir une NetworkPolicy qui n'autorise que le trafic du frontend vers le backend :

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-policy
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
```

Sécurité dans Kubernetes

Introduction aux PodSecurityPolicies

Les PodSecurityPolicies (PSP) sont des objets qui contrôlent les paramètres de sécurité sensibles et les autorisations pour les pods. **Notez que les PSP sont en cours de dépréciation dans les versions récentes de Kubernetes.**

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-example
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  volumes:
    - 'configMap'
    - 'emptyDir'
    - 'projected'
    - 'secret'
    - 'downwardAPI'
    - 'persistentVolumeClaim'
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'RunAsAny'
  seLinux:
    rule: 'RunAsAny'
```



Kubernetes et son écosystème

Kubernetes et son écosystème

1. Kubernetes et son écosystème

- **Helm:** Helm est souvent appelé le gestionnaire de paquets de Kubernetes.
- **Charts:** Ce sont des packages de ressources Kubernetes. Avec Helm, les développeurs peuvent créer des charts reproductibles pour leurs applications.
- **Déploiement simplifié:** Helm facilite le déploiement, la mise à jour et la suppression d'applications sur un cluster Kubernetes.

2. Prometheus

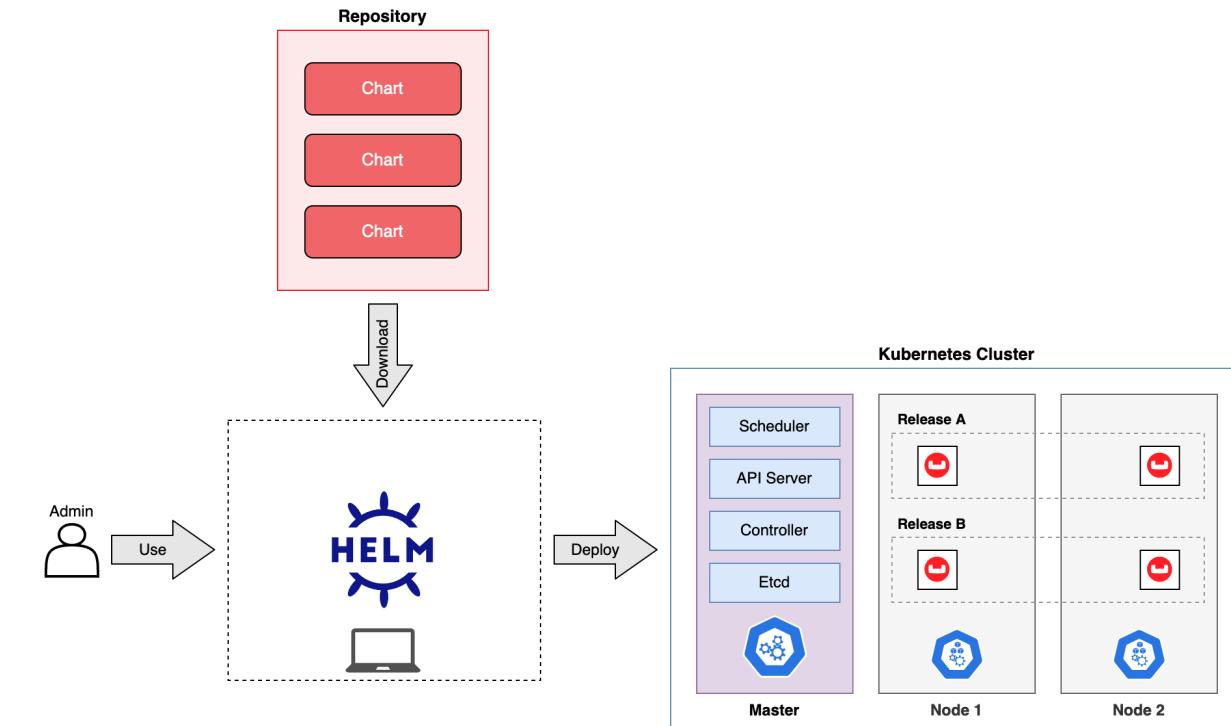
- C'est un outil de surveillance et d'alerte open-source.
 - **Intégration étroite avec Kubernetes:** Prometheus peut automatiquement découvrir des services et des métriques.
 - **Requêtes puissantes:** Son langage de requête permet aux utilisateurs d'inspecter leurs métriques et de générer des alertes.

Kubernetes et son écosystème

3. Helm

Helm est un gestionnaire de paquets pour Kubernetes, qui vous permet de définir, installer et mettre à jour des applications Kubernetes. Helm utilise des "charts" qui sont des paquets de fichiers YAML pré-configurés pour déployer des applications ou des services dans un cluster Kubernetes. Voici un exemple simple pour illustrer comment Helm est utilisé pour déployer une application.

```
apiVersion: v2
name: nginx-app
description: A Helm chart for Kubernetes
version: 0.1.0
```



Kubernetes et son écosystème

Déployer des Solutions avec Helm

Helm est un gestionnaire de paquets pour Kubernetes, qui permet de définir, installer et mettre à jour des applications Kubernetes. Helm utilise des "charts", qui sont des collections de fichiers YAML décrivant un ensemble de ressources Kubernetes associées.

Exemple de déploiement d'une application avec Helm :

- 1. Installation de Helm** : Tout d'abord, vous devez installer Helm sur votre machine. Vous pouvez le télécharger depuis [le site officiel de Helm](#).
- 2. Recherche d'un Chart** : Trouvez le chart Helm pour l'application que vous souhaitez déployer. Vous pouvez utiliser la commande `helm search repo` pour rechercher dans les dépôts Helm publics.
- 3. Installation d'un Chart** : Une fois que vous avez trouvé le chart, vous pouvez l'installer avec la commande `helm install`. Par exemple :

```
helm install mon-application stable/mon-chart
```

Cette commande déploie `mon-chart` dans votre cluster Kubernetes.

- 4. Mise à jour et Gestion** : Vous pouvez gérer le cycle de vie de votre déploiement avec les commandes `helm upgrade` et `helm rollback` pour mettre à jour ou revenir à une version antérieure de votre application.

Kubernetes et son écosystème

Etcd, Gestion de Configuration Distribuée

etcd est une base de données clé-valeur distribuée et un élément essentiel de l'architecture Kubernetes. Elle stocke les informations de configuration et l'état du cluster, servant de vérité source pour votre cluster Kubernetes.

Utilisation d'etcd dans Kubernetes :

- Stockage de l'état du cluster** : Kubernetes utilise etcd pour stocker l'ensemble de l'état du cluster, y compris les informations sur les pods, les services, et les politiques.
- Haute disponibilité** : etcd est conçu pour être fiable et répartir ses données sur plusieurs nœuds, ce qui est crucial pour la haute disponibilité du cluster Kubernetes.
- Interactions avec etcd** : Bien que vous interagissiez rarement directement avec etcd dans les opérations Kubernetes quotidiennes, il est important de sauvegarder régulièrement les données d'etcd et de comprendre son rôle dans le cluster.
- Sécurité** : Assurez-vous que votre installation d'etcd est sécurisée, car elle contient toutes les informations critiques sur votre cluster.

Kubernetes et son écosystème

Prometheus / Grafana

Prometheus est un système de surveillance et d'alerte open-source largement utilisé, particulièrement adapté pour les environnements basés sur des conteneurs comme Kubernetes. Il offre une solution robuste pour surveiller non seulement les machines et services, mais aussi les métriques spécifiques à Kubernetes et les états des objets Kubernetes.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  labels:
    app: prometheus
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus:v2.26.0
          ports:
            - containerPort: 9090
```

Kubernetes et son écosystème

4. Prometheus / Grafana

