

Kubernetes

Introduction à l'orchestration de conteneurs

Hugo Blanc

Université Lyon 1

Présentation	4
Introduction à Kubernetes	12
Premiers pas avec Kubernetes	40
Gestion déclarative avec YAML	77
Gestion des données et configuration	111
Networking et exposition des services	149
Sécurité dans Kubernetes	193
Licence	242

- Avoir une machine GNU/Linux en état de fonctionnement
- Avoir un utilisateur différent de root appartenant au groupe sudo
- Avoir une connexion à Internet
- Avoir des connaissances de base sur Linux et le terminal
- Avoir des notions de conteneurisation (Docker)
- Comprendre les bases des réseaux informatiques

Important

Ce cours nécessite un accès à un cluster Kubernetes. Nous utiliserons des outils minikube pour créer un environnement local de développement.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Hugo Blanc

Hugo Blanc

→ Platform Security Engineer @ Doctolib

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

Hugo Blanc

→ Platform Security Engineer @ Doctolib

→ Enseignant @ UCBL depuis 2022

Adepte du tutoiement :)

- Linux and containerized workloads hardening
- Networking security, detection automation
- Kubernetes & Cloud security
- Blue team & forensics
- Incident management & response

- Site Reliability Engineer (aka Cloud sysadmin) @ [Virtuo](#)
- DevOps & Security @ [DevOps.Works](#)
- Étudiant @ LP ESSIR :)

En cas de questions sur les cours (ou Linux/infosec/cloud en général), ne pas hésiter:
hugo.blanc@univ-lyon1.fr

Best effort pour les réponses :)

Où trouver les cours ?

Fascicules et slides:

⇒ <https://syscall.cafe/t/>

Les dernières versions y seront publiées avant le début de chaque module.

Ce cours sera évalué par :

- Participation et implication en classe
- QCM de 30 minutes en fin de module

- L'utilisation des LLMs (et autres outils) est **déconseillée** car ils ne participent pas à la réflexion et à l'apprentissage.
- Pas de sanctions si usage des « modéré » des LLMs **pour les TP notés seulement**. Des questions orales de validation des acquis peuvent être posées lors de la restitution.
- Tout aide durant les contrôles sur table sera considérée comme de la triche, et mènera à une note de zéro.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Définition

Kubernetes (souvent abrégé K8s) est une plateforme open-source qui automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.

Définition

Kubernetes (souvent abrégé K8s) est une plateforme open-source qui automatise le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.

Imaginez que vous ayez des dizaines d'applications à faire tourner sur plusieurs serveurs. Sans Kubernetes, vous devriez manuellement :

- Déployer chaque application sur les bons serveurs
- Surveiller leur état de santé
- Les redémarrer si elles plantent
- Gérer les mises à jour
- Équilibrer la charge entre les instances

Kubernetes fait tout cela automatiquement pour vous !

Pourquoi utiliser Kubernetes ?

Les défis sans orchestration

- **Déploiement manuel** : placer les conteneurs sur les serveurs à la main
- **Haute disponibilité** : si une application plante, personne ne la redémarre
- **Mise à l'échelle** : ajouter des instances manuellement quand la charge augmente
- **Load balancing** : configurer manuellement la répartition de charge
- **Mise à jour** : arrêter, mettre à jour et redémarrer chaque instance

Pourquoi utiliser Kubernetes ?

Les avantages de Kubernetes

- **Orchestration automatisée** : placement intelligent des conteneurs
- **Auto-guérison** : redémarrage automatique des applications qui crashent
- **Mise à l'échelle automatique** : ajout/suppression d'instances selon la charge
- **Équilibrage de charge** : distribution automatique du trafic
- **Déploiements progressifs** : mises à jour sans interruption de service
- **Déclaratif** : vous décrivez l'état souhaité, Kubernetes s'occupe du reste

Kubernetes est utilisé dans de nombreux contextes :

- **Applications web** : sites e-commerce, plateformes SaaS
- **Microservices** : architectures distribuées complexes
- **Traitement de données** : pipelines ETL, analytics
- **APIs et services** : backends d'applications mobiles
- **Environnements de développement** : staging, tests automatisés
- **Applications legacy** : modernisation d'anciennes applications
- **Monitoring et observabilité** : déployer Prometheus, Grafana, ELK Stack

Note

Kubernetes excelle particulièrement pour les applications qui doivent être scalables, résilientes et déployées fréquemment.

Kubernetes fonctionne selon une architecture maître-esclave avec deux types de composants principaux :

Le Control Plane (le cerveau du cluster)

Le Control Plane orchestre tout le cluster et contient :

- **API Server** (kube-apiserver)
 - Point d'entrée unique pour toutes les commandes
 - Valide et stocke les configurations
 - Interface REST/JSON pour tous les clients
- **etcd**
 - Base de données distribuée clé-valeur
 - Stocke tout l'état du cluster de façon persistante
 - Seul composant vraiment stateful

- **Scheduler** (kube-scheduler)
 - Décide sur quels nodes placer les nouveaux Pods
 - Prend en compte les ressources, contraintes, affinités
- **Controller Manager** (kube-controller-manager)
 - Surveille l'état actuel vs l'état désiré
 - Contient de nombreux contrôleurs (Deployment, ReplicaSet, etc.)
- **Cloud Controller Manager**
 - Interface avec les APIs des fournisseurs cloud
 - Gère les load balancers, volumes cloud, etc.

Les nodes

Chaque node exécute vos applications et contient :

- **kubelet**
 - Agent principal du node
 - Communique avec le Control Plane
 - Lance et surveille les conteneurs
- **kube-proxy**
 - Composant réseau du node
 - Gère l'équilibrage de charge local
 - Implémente les Services
- **Container Runtime**
 - Moteur qui fait tourner les conteneurs
 - Docker, containerd, CRI-O...

Remarque

Le Control Plane peut être hébergé sur des nodes dédiés ou partagés avec les workloads. En production, on sépare généralement Control Plane et nodes de travail.

Tout passe par l'API Server

- Les utilisateur.rices communiquent via `kubectl` ou l'API REST
- Les nodes s'enregistrent et récupèrent leurs instructions
- Tous les composants du Control Plane passent par l'API Server
- Seul l'API Server lit/écrit dans `etcd`

Modèle déclaratif vs impératif

Approche impérative (traditionnelle) :

```
$ ssh server
$ docker run --name nginx-1 nginx:1.29.1
$ docker run --name nginx-2 nginx:1.29.1

# en cas de crash, il faut redémarrer à la main
$ docker restart nginx-1
```

Approche déclarative (Kubernetes) : **Approche déclarative** (Kubernetes) :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.29.1
```

Approche déclarative (Kubernetes) :

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

Approche déclarative (Kubernetes) :

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

Important

Avec Kubernetes, vous décrivez l'état souhaité (2 instances nginx), pas les étapes pour y arriver. Kubernetes se charge de la réconciliation continue entre l'état actuel et l'état désiré.

Pod : l'unité atomique

Concept K8s : Pod

Un Pod est la plus petite unité déployable dans Kubernetes. Il contient un ou plusieurs conteneurs qui travaillent ensemble et partagent des ressources.

Pod : l'unité atomique

Concept K8s : Pod

Un Pod est la plus petite unité déployable dans Kubernetes. Il contient un ou plusieurs conteneurs qui travaillent ensemble et partagent des ressources.

Caractéristiques d'un Pod :

- Partage la même adresse IP
- Partage les mêmes volumes de stockage
- Partage le même cycle de vie
- Communication inter-conteneurs via localhost

```
apiVersion: v1
kind: Pod
metadata:
  name: mon-pod
spec:
  containers:
    - name: nginx
      image: nginx:1.29.1
      ports:
        - containerPort: 80
    - name: sidecar
      image: busybox
      command: ["sleep", "3600"]
```

Remarque

En pratique, vous déployez rarement des Pods directement. Vous utilisez des contrôleurs comme Deployment qui gèrent les Pods pour vous.

ReplicaSet : la garantie de réplication

Concept K8s : ReplicaSet

Un ReplicaSet s'assure qu'un nombre précis de Pods identiques tournent en permanence.

ReplicaSet : la garantie de réplication

Concept K8s : ReplicaSet

Un ReplicaSet s'assure qu'un nombre précis de Pods identiques tournent en permanence.

Si vous voulez 3 copies de votre application web :

- Le ReplicaSet maintient exactement 3 Pods
- Si un Pod plante, il en crée un nouveau automatiquement
- Si vous en supprimez un manuellement, il le recrée
- Si vous mettez à jour le nombre de réplicas, il ajuste automatiquement

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.29.1
```

Deployment : la gestion des versions

Concept K8s : Deployment

Un Deployment gère les ReplicaSets et vous permet de faire des mises à jour en douceur avec historique et rollback.

Deployment : la gestion des versions

Concept K8s : Deployment

Un Deployment gère les ReplicaSets et vous permet de faire des mises à jour en douceur avec historique et rollback.

C'est votre outil principal pour déployer des applications. Il peut :

- Faire des mises à jour progressives (*rolling updates*)
- Revenir en arrière si quelque chose ne va pas
- Garder un historique des versions déployées
- Mettre en pause et reprendre des déploiements

Concepts fondamentaux

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.29.1
          ports:
            - containerPort: 80
```

Service : l'adresse stable

Concept K8s : Service

Un Service expose vos Pods sous une adresse IP et un port fixes, avec équilibrage de charge automatique.

Service : l'adresse stable

Concept K8s : Service

Un Service expose vos Pods sous une adresse IP et un port fixes, avec équilibrage de charge automatique.

Pourquoi avons-nous besoin de Services ?

- Les Pods sont éphémères (peuvent être créés/détruits)
- Leur adresse IP change à chaque fois
- Comment les clients peuvent-ils accéder aux applications ?

Le Service résout ce problème en fournissant :

- Une adresse IP stable (ClusterIP)
- Un nom DNS stable
- Un équilibrage de charge entre les Pods

Types de Services :

- **ClusterIP** : accessible seulement depuis le cluster (défaut)
- **NodePort** : accessible depuis l'extérieur via un port sur chaque node
- **LoadBalancer** : utilise un load balancer externe (cloud)
- **ExternalName** : fait un mapping vers un nom DNS externe

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

Exercice pratique

1. Qu'arrive-t-il si vous supprimez un Pod géré par un Deployment ?
2. Quelle différence entre un Pod et un Deployment ?
3. Pourquoi ne peut-on pas accéder directement aux Pods depuis l'extérieur ?
4. Que se passe-t-il si un node tombe en panne avec des Pods dessus ?

Voici comment les composants interagissent lors du déploiement d'une application :

1. **Utilisateur.rice** crée un Deployment via `kubectl apply`
2. **API Server** valide et stocke la configuration dans **etcd**
3. **Deployment Controller** détecte le nouveau Deployment
4. **Deployment Controller** crée un ReplicaSet
5. **ReplicaSet Controller** crée les Pods requis
6. **Scheduler** assigne les Pods aux nodes disponibles
7. **kubelet** sur chaque node lance les conteneurs
8. **kube-proxy** configure l'équilibrage de charge pour les Services

Important

Cette approche événementielle et déclarative permet à Kubernetes d'être résilient : si un composant tombe, il peut rattraper les événements manqués en comparant l'état actuel avec l'état désiré.

Dans le module suivant, nous allons installer les outils et créer notre premier cluster pour voir tout cela en action.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Avant de pouvoir utiliser Kubernetes, nous devons installer les outils nécessaires et créer un cluster de développement local.

Avant de pouvoir utiliser Kubernetes, nous devons installer les outils nécessaires et créer un cluster de développement local.

Installer kubectl

`kubectl` est l'outil en ligne de commande pour interagir avec Kubernetes. C'est votre interface principale.

```
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
$ chmod +x kubectl
$ sudo mv kubectl /usr/local/bin/

# puis...
kubectl version --client
```

Note

Même si déconseillé, `kubectl` peut être installé via les gestionnaires de packages :
`sudo apt install kubectl` (Ubuntu), `sudo dnf install kubectl` (Fedora), etc.

Note

Même si déconseillé, `kubectl` peut être installé via les gestionnaires de packages :
`sudo apt install kubectl` (Ubuntu), `sudo dnf install kubectl` (Fedora), etc.

Note

Sur beaucoup de systèmes, les utilisateur.rices créent un alias `k` pour `kubectl` pour gagner du temps.

Créer un cluster local avec minikube

Minikube crée un cluster Kubernetes local à des fins de développement et d'apprentissage.

```
$ curl -LO https://storage.googleapis.com/minikube/releases/  
latest/minikube-linux-amd64  
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube  
  
# puis...  
$ minikube start  
$ kubectl cluster-info
```

Sortie attendue :

```
Kubernetes control plane is running at https://127.0.0.1:32776  
CoreDNS is running at https://127.0.0.1:32776/api/v1/  
namespaces/kube-system/services/kube-dns:dns/proxy
```

```
To further debug and diagnose cluster problems, use 'kubectl  
cluster-info dump'.
```

Configurer l'accès au cluster

```
# Voir la configuration actuelle
```

```
$ kubectl config view
```

```
# Lister les clusters disponibles
```

```
$ kubectl config get-contexts
```

```
# Passer d'un cluster à l'autre
```

```
$ kubectl config use-context minikube
```

```
# Voir le cluster actuel
```

```
$ kubectl config current-context
```

Le fichier de configuration se trouve dans `~/.kube/config` et contient :

- Les clusters (URL, certificats)
- Les utilisateur.rices (authentification)
- Les contextes (association cluster + utilisateur.rice)

Informations générales

Infos générales sur le cluster

\$ kubectl cluster-info

Version de Kubernetes (locale + cluster)

\$ kubectl version

État détaillé du cluster

\$ kubectl cluster-info dump

Examiner les nodes

```
# Lister les nodes du cluster
```

```
$ kubectl get nodes
```

```
# Plus de détails sur un node
```

```
$ kubectl describe node minikube
```

```
# Voir les ressources utilisées
```

```
$ kubectl top nodes # Nécessite metrics-server
```

Sortie typique :

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	control-plane	18s	v1.33.1

Explorer les namespaces

Les namespaces permettent d'organiser et d'isoler les ressources. Ils représentent une séparation **logique**.

```
# Lister les namespaces  
$ kubectl get namespaces
```

```
# Voir les pods dans kube-system  
$ kubectl get pods -n kube-system
```

```
# Voir toutes les ressources de tous les namespaces  
$ kubectl get all --all-namespaces
```

Namespaces par défaut :

- `default` : namespace par défaut pour vos applications
- `kube-system` : composants système de Kubernetes
- `kube-public` : ressources publiques accessibles à tous
- `kube-node-lease` : informations de heartbeat des nodes

Créer un Pod impérativement

```
# Créer un Pod avec nginx dans le namespace courant  
$ kubectl run nginx-1 --image=nginx
```

```
# Voir tous les Pods du namespace courant  
$ kubectl get pods
```

```
# Plus de détails sur notre Pod  
$ kubectl describe pod nginx-1
```

```
# Voir les logs du Pod  
$ kubectl logs nginx-1
```

```
# S'exécuter dans le Pod  
$ kubectl exec -it nginx-1 -- bash
```

Exercice pratique

Exercice 1 : Explorer un Pod

1. Créez un Pod avec l'image `nginx:1.29.1`
2. Vérifiez qu'il est en état « Running »
3. Regardez ses logs
4. Connectez-vous dedans et explorez le système de fichiers
5. Sortez et supprimez le Pod

États d'un Pod

Un Pod peut être dans différents états :

- **Pending** : En attente d'être planifié sur un node
- **Running** : Au moins un conteneur fonctionne
- **Succeeded** : Tous les conteneurs se sont terminés avec succès
- **Failed** : Au moins un conteneur a échoué
- **Unknown** : L'état ne peut pas être déterminé

Diagnostiquer les problèmes

Si un Pod ne démarre pas :

```
# Voir les détails et événements
```

```
$ kubectl describe pod nom-du-pod
```

```
# Voir les logs (même si le Pod a échoué)
```

```
$ kubectl logs nom-du-pod
```

```
# Voir les logs du conteneur précédent
```

```
$ kubectl logs nom-du-pod --previous
```

```
# Forcer la suppression d'un Pod bloqué
```

```
$ kubectl delete pod nom-du-pod --force --grace-period=0
```

Les Pods seuls ne sont pas très utiles car ils ne reviennent pas s'ils crashent.
Utilisons des Deployments.

Créer un Deployment

```
# Créer un Deployment
$ kubectl create deployment nginx --image=nginx:1.29.1

# Voir le Deployment créé
$ kubectl get deployments

# Voir les ReplicaSets créés automatiquement
$ kubectl get replicaset

# Voir les Pods créés par le Deployment
$ kubectl get pods
```

Note

Notez qu'un Deployment crée automatiquement un ReplicaSet, qui à son tour crée les Pods. Cette hiérarchie permet la gestion des versions et des mises à jour.

Mise à l'échelle

```
# Passer à 3 replicas
```

```
$ kubectl scale deployment nginx --replicas=3
```

```
# Vérifier qu'on a maintenant 3 Pods
```

```
$ kubectl get pods -l app=nginx
```

```
# Voir la répartition
```

```
$ kubectl get pods -o wide
```

Tester la résilience

Supprimer un Pod manuellement

\$ kubectl delete pod <nom-d-un-pod>

Observer qu'il est recréé automatiquement

\$ kubectl get pods -w # -w pour "watch"

Exercice pratique

Exercice 2 : Résilience des Deployments

1. Créez un Deployment nginx avec 4 replicas
2. Observez les Pods créés
3. Supprimez un Pod et observez qu'il est recréé
4. Mettez à l'échelle à 7 replicas
5. Réduisez à 1 replica

Problème : accéder aux Pods

Les Pods ont des IP éphémères dans le cluster. Comment y accéder depuis l'extérieur ?

```
# Voir les IPs des Pods  
$ kubectl get pods -o wide
```

```
# Essayer d'accéder (ne marchera que depuis le node)  
$ curl <IP-du-pod>:80
```

Solution : créer un Service

```
# Exposer le Deployment avec un Service
$ kubectl expose deployment nginx --port=80 --type=NodePort

# Voir le Service créé
$ kubectl get services

# Voir plus de détails
$ kubectl describe service nginx
```

Le Service NodePort expose l'application sur un port de chaque node du cluster.

Accéder à l'application

```
# Avec minikube, obtenir l'URL d'accès
```

```
$ minikube service nginx --url
```

```
# Ou redirection de port locale
```

```
$ kubectl port-forward service/nginx 8080:80
```

Maintenant vous pouvez ouvrir <http://localhost:8080> dans votre navigateur !

Types de Services

ClusterIP (par défaut) - interne seulement

\$ kubectl expose deployment nginx --port=80 --type=ClusterIP

NodePort - accessible via <NodeIP>:<NodePort>

\$ kubectl expose deployment nginx --port=80 --type=NodePort

LoadBalancer - utilise un LB externe (cloud)

\$ kubectl expose deployment nginx --port=80 --
type=LoadBalancer

Exercice pratique

Exercice 3 : Services et exposition

1. Créez un Deployment avec Apache (`httpd:2.4.65`)
2. Créez un Service NodePort pour l'exposer
3. Testez l'accès via port-forward
4. Créez un second Deployment nginx
5. Observez comment le Service équilibre la charge

Voir les ressources

Vue d'ensemble de tout

```
$ kubectl get all
```

Voir les événements récents

```
$ kubectl get events --sort-by='.lastTimestamp'
```

Ressources avec plus d'informations

```
$ kubectl get pods -o wide
```

```
$ kubectl get services -o wide
```

Format de sortie personnalisé

```
$ kubectl get pods -o custom-  
columns=NAME:.metadata.name,STATUS:.status.phase,IP:.status.podIP
```

Debugging avancé

Examiner la configuration d'une ressource

\$ kubectl get pod my-pod -o yaml

Éditer une ressource en live

\$ kubectl edit deployment nginx

Voir l'utilisation des ressources

\$ kubectl top pods

\$ kubectl top nodes

Suivre les logs en temps réel

\$ kubectl logs -f deployment/nginx

Diagnostiquer les problèmes réseau

\$ kubectl exec -it my-pod -- nslookup kubernetes.default

Nettoyer

Supprimer une ressource spécifique

\$ kubectl delete pod my-pod

\$ kubectl delete service my-svc

\$ kubectl delete deployment my-depl

Supprimer selon les labels

\$ kubectl delete pods -l app=my-app

Supprimer tout dans un namespace

\$ kubectl delete all --all -n mon-namespace

Vider complètement un namespace

\$ kubectl delete namespace mon-namespace

Minikube fournit une interface graphique pour explorer le cluster :

```
# Activer le dashboard
$ minikube addons enable dashboard

# Ouvrir le dashboard
$ minikube dashboard
```

Le dashboard permet de :

- Visualiser toutes les ressources
- Voir les logs et métriques
- Créer des ressources via une interface graphique
- Diagnostiquer les problèmes

Attention

Le dashboard Kubernetes ne doit JAMAIS être exposé sur Internet en production sans authentification appropriée. Utilisez-le uniquement pour le développement local.

Gestion du cluster

\$ kubectl cluster-info	# Infos sur le cluster
\$ kubectl get nodes	# Lister les nodes
\$ kubectl get namespaces	# Lister les namespaces

Gestion des Pods

\$ kubectl run name --image=image	# Créer un Pod
\$ kubectl get pods	# Lister les Pods
\$ kubectl describe pod name	# Détails d'un Pod
\$ kubectl logs name	# Logs d'un Pod
\$ kubectl exec -it name -- bash	# Se connecter dans un Pod
\$ kubectl delete pod name	# Supprimer un Pod

Gestion des Deployments

```
$ kubectl create deployment name --image=image # Créer un
Deployment
$ kubectl get deployments # Lister les
Deployments
$ kubectl scale deployment name --replicas=N # Mise à
l'échelle
$ kubectl delete deployment name # Supprimer un
Deployment
```

Gestion des Services

```
$ kubectl expose deployment name --port=80      # Créer un  
Service  
$ kubectl get services                          # Lister les  
Services  
$ kubectl port-forward service/name 8080:80    # Redirection  
de port  
$ kubectl delete service name                  # Supprimer un  
Service
```

Exercice pratique

Exercice final du module

1. Créez un cluster minikube propre
2. Déployez une application web (nginx ou apache) avec 3 replicas
3. Exposez-la avec un Service NodePort
4. Testez l'accès depuis votre navigateur
5. Simulez une panne en supprimant un Pod
6. Vérifiez que l'application reste accessible
7. Mettez à jour l'image vers une version plus récente
8. Nettoyez toutes les ressources

Dans le module suivant, nous apprendrons à gérer tout cela de façon déclarative avec des fichiers YAML au lieu de commandes impératives.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Pourquoi passer au déclaratif ?

Jusqu'ici nous avons utilisé des commandes impératives (`kubectl run`, `kubectl create`, `kubectl expose`). C'est pratique pour tester et apprendre, mais en production on préfère l'approche déclarative avec des fichiers YAML.

Pourquoi passer au déclaratif ?

Problèmes de l'approche impérative

```
$ kubectl create deployment webapp --image=nginx:1.29.1
$ kubectl scale deployment webapp --replicas=3
$ kubectl expose deployment webapp --port=80 --type=NodePort
$ kubectl set env deployment webapp ENV=production
$ kubectl set resources deployment webapp --
limits=cpu=200m,memory=256Mi
```

- Comment se souvenir de toutes ces commandes ?
- Comment reproduire exactement la même configuration ailleurs ?
- Comment collaborer en équipe sur les configurations ?
- Comment suivre les changements au fil du temps ?

Pourquoi passer au déclaratif ?

Avantages de l'approche déclarative

Avec des fichiers YAML :

- **Versioning** : suivre les changements avec Git
- **Reproductibilité** : même configuration partout
- **Collaboration** : revue de code sur l'infrastructure
- **Documentation** : la configuration est auto-documentée
- **Automatisation** : intégration dans les pipelines CI/CD
- **Rollback** : retour en arrière facile
- **GitOps** : synchronisation automatique Git ↔ Cluster

Pourquoi passer au déclaratif ?

Avantages de l'approche déclarative

Avec des fichiers YAML :

- **Versioning** : suivre les changements avec Git
- **Reproductibilité** : même configuration partout
- **Collaboration** : revue de code sur l'infrastructure
- **Documentation** : la configuration est auto-documentée
- **Automatisation** : intégration dans les pipelines CI/CD
- **Rollback** : retour en arrière facile
- **GitOps** : synchronisation automatique Git ↔ Cluster

Important

L'*Infrastructure as Code* (IaC) est une pratique essentielle en infosec pour assurer la traçabilité, la reproductibilité et la conformité des déploiements.

Anatomy d'un fichier YAML

Tous les objets Kubernetes suivent la même structure de base :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
    environment: production
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.29.1
          ports:
            - containerPort: 80
```

Les quatre sections principales

1. **apiVersion** : Quelle version de l'API Kubernetes utiliser
 - `v1` : API core (Pod, Service, ConfigMap...)
 - `apps/v1` : API apps (Deployment, StatefulSet...)
 - `networking.k8s.io/v1` : API réseau (Ingress, NetworkPolicy...)
2. **kind** : Le type d'objet (Pod, Deployment, Service, ConfigMap...)
3. **metadata** : Informations sur l'objet
 - `name` : nom unique dans le namespace
 - `namespace` : namespace de l'objet (défaut: « default »)
 - `labels` : étiquettes clé-valeur pour organiser
 - `annotations` : métadonnées arbitraires
4. **spec** : Spécification de l'état désiré
 - Structure différente selon le kind
 - Décrit comment l'objet doit être

Trouver la bonne apiVersion et structure

```
# Lister toutes les versions d'API disponibles
$ kubectl api-versions

# Voir le schéma d'un type d'objet
$ kubectl explain deployment
$ kubectl explain deployment.spec
$ kubectl explain deployment.spec.template.spec.containers

# Obtenir un exemple de YAML depuis un objet existant
$ kubectl get deployment mon-deployment -o yaml
```

Les labels : des étiquettes pour tout organiser

Les labels sont des paires clé-valeur attachées aux objets Kubernetes. Ils sont cruciaux pour organiser et sélectionner vos ressources.

```
metadata:  
  labels:  
    app: nginx  
    version: "1.29.1"  
    environment: production  
    team: frontend  
    component: webserver  
    security-scan: "passed"
```

Note

Les labels ne doivent contenir que des caractères alphanumériques, tirets et points. Les valeurs doivent être entre guillemets si elles commencent par un chiffre.

Conventions de labellisation

Bonnes pratiques pour les labels :

```
metadata:
  labels:
    # Labels recommandés par Kubernetes
    app.kubernetes.io/name: nginx
    app.kubernetes.io/instance: nginx-prod
    app.kubernetes.io/version: "1.29.1"
    app.kubernetes.io/component: frontend
    app.kubernetes.io/managed-by: helm

    # Labels personnalisés pour la sécurité
    security.doctolib.com/scan-date: "2025-01-4"
    security.doctolib.com/approved-by: "platform-security"
    compliance.doctolib.com/gdpr: "compliant"
```

Utiliser les sélecteurs

Les sélecteurs permettent de cibler des groupes d'objets basés sur leurs labels :

Tous les Pods de l'application nginx

\$ kubectl get pods -l app=nginx

Pods en production

\$ kubectl get pods -l environment=production

Pods nginx en production

\$ kubectl get pods -l app=nginx,environment=production

Pods avec version différente de 1.20

\$ kubectl get pods -l 'version!=1.20'

Pods avec label environment (peu importe la valeur)

\$ kubectl get pods -l environment

Opérations basées sur les labels

\$ kubectl delete pods -l app=nginx

\$ kubectl scale deployment -l app=nginx --replicas=5

Sélecteurs dans les manifests

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector: # Sélectionne les Pods avec ces labels
    app: nginx
  ports:
    - port: 80
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:      # Doit correspondre aux labels du template
      app: nginx
  template:
    metadata:
      labels:
        app: nginx    # Ces labels doivent matcher le selector
    spec:
      containers:
        - name: nginx
          image: nginx:1.29.1
```

Attention

Le selector d'un Deployment doit correspondre exactement aux labels du template de Pod, sinon le Deployment ne pourra pas gérer ses Pods.

La commande `kubectl apply`

Appliquer un fichier

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

Appliquer tous les fichiers d'un dossier

```
$ kubectl apply -f ./configurations/
```

Appliquer récursivement tous les sous-dossiers

```
$ kubectl apply -f ./configurations/ -R
```

Appliquer depuis une URL (peu recommandé)

```
$ kubectl apply -f https://raw.githubusercontent.com/  
kubernetes/examples/master/nginx-deployment.yaml
```

Différence entre create et apply

kubectl create : impératif, échoue si l'objet existe

\$ kubectl create -f deployment.yaml

kubectl apply : déclaratif, met à jour si l'objet existe

\$ kubectl apply -f deployment.yaml

Voir les différences avant application

```
# Voir ce qui va changer sans appliquer
```

```
$ kubectl diff -f deployment.yaml
```

```
# Simulation sans modification (dry-run)
```

```
$ kubectl apply -f deployment.yaml --dry-run=client
```

```
# Validation du YAML sans création
```

```
$ kubectl apply -f deployment.yaml --dry-run=server --  
validate=true
```

Déploiement d'une application web sécurisée

Exemples pratiques complets

```
# webapp-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-webapp
  labels:
    app: secure-webapp
    version: "1.0"
    environment: production
    security-level: high
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: secure-webapp
  template:
    metadata:
      labels:
        app: secure-webapp
        version: "1.0"
    spec:
      # Contexte de sécurité au niveau Pod
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
        fsGroup: 2000
      containers:
        - name: webapp
          image: nginx:1.29.1-alpine
          ports:
            - containerPort: 8080
```

```
    name: http
  # Contexte de sécurité au niveau conteneur
  securityContext:
    allowPrivilegeEscalation: false
    readOnlyRootFilesystem: true
    runAsNonRoot: true
    capabilities:
      drop:
        - ALL
  # Limites de ressources
  resources:
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
  # Health checks
  livenessProbe:
    httpGet:
      path: /health
      port: 8080
    initialDelaySeconds: 30
    periodSeconds: 10
  readinessProbe:
    httpGet:
      path: /ready
      port: 8080
    initialDelaySeconds: 5
    periodSeconds: 5
  # Variables d'environnement
  env:
    - name: ENVIRONMENT
      value: "production"
    - name: LOG_LEVEL
      value: "info"
```

Service associé

```
# webapp-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: secure-webapp-service
  labels:
    app: secure-webapp
spec:
  type: ClusterIP
  selector:
    app: secure-webapp
  ports:
    - name: http
      port: 80
      targetPort: 8080
      protocol: TCP
```

Organiser avec plusieurs objets dans un fichier

```
# webapp-complete.yaml - Tous les objets
ensemble
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-webapp
spec:
  # ... spec du deployment
---
```

```
Séparateur entre objets
apiVersion: v1
kind: Service
metadata:
  name: secure-webapp-service
spec:
```

```
# ... spec du service
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
data:
  nginx.conf: |
    server {
      listen 8080;
      location / {
        return 200 'Hello Secure
World!';
      }
    }
```

Exercice pratique

Exercice 1 : Conversion impératif → déclaratif

1. Créez un Deployment nginx avec `kubectl create deployment`
2. Exportez-le en YAML avec `-o yaml`
3. Sauvegardez dans un fichier et nettoyez-le
4. Supprimez le Deployment original
5. Recréez-le avec `kubectl apply -f`
6. Modifiez le fichier pour changer le nombre de réplicas
7. Appliquez les changements

Créer et utiliser des namespaces

```
# namespace-dev.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    environment: dev
    team: engineering
```

Créer le namespace

```
$ kubectl apply -f namespace-dev.yaml
```

Déployer dans un namespace spécifique

```
$ kubectl apply -f deployment.yaml -n development
```

Ou spécifier dans le YAML

```
metadata:
```

```
  name: my-app
```

```
  namespace: development
```

Organiser par environnements

```
projet/
├── base/
│   ├── deployment.yaml
│   └── service.yaml
├── environments/
│   ├── development/
│   │   ├── namespace.yaml
│   │   └── kustomization.yaml
│   ├── staging/
│   │   ├── namespace.yaml
│   │   └── kustomization.yaml
│   └── production/
│       ├── namespace.yaml
│       └── kustomization.yaml
```

Erreurs communes

```
# ❌ Erreur d'indentation
spec:
replicas: 3      # Manque
d'indentation
```

```
# ✅ Correct
spec:
  replicas: 3
```

```
# ❌ Type de données incorrect
spec:
  replicas: "3"   # String au lieu
d'integer
```

```
# ✅ Correct
```

```
spec:
  replicas: 3
```

```
# ❌ Labels selector ne correspondent
pas
```

```
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        application: nginx #
Différent du selector !
```

Outils de validation

```
# Validation basique du YAML
```

```
$ kubectl apply --dry-run=client -f deployment.yaml
```

```
# Validation côté serveur
```

```
$ kubectl apply --dry-run=server -f deployment.yaml
```

Debugging des applications déclaratives

Voir les différences avec l'état actuel

\$ kubectl diff -f deployment.yaml

Historique des révisions d'un Deployment

\$ kubectl rollout history deployment/my-app

Revenir à la révision précédente

\$ kubectl rollout undo deployment/my-app

Voir les événements liés à un objet

\$ kubectl describe deployment my-app | grep -A 10 Events

Suivre le status d'un rollout

\$ kubectl rollout status deployment/my-app

Exercice pratique

Exercice 2 : Application complète en YAML

1. Créez un fichier YAML avec un Deployment pour Apache
2. Ajoutez des labels appropriés (app, version, environment)
3. Configurez des limites de ressources
4. Ajoutez un Service ClusterIP
5. Créez un namespace « web-servers »
6. Déployez tout avec `kubectl apply`
7. Testez l'accès avec `port-forward`
8. Modifiez l'image vers une version plus récente
9. Observez le rolling update

Structure des projets

```
k8s-manifests/  
├── namespaces/  
│   ├── development.yaml  
│   ├── staging.yaml  
│   └── production.yaml  
├── configmaps/  
│   ├── app-config.yaml  
│   └── nginx-config.yaml  
├── deployments/  
│   ├── frontend.yaml  
│   ├── backend.yaml  
│   └── database.yaml  
├── services/  
│   ├── frontend-svc.yaml  
│   └── backend-svc.yaml  
└── ingress/  
    └── app-ingress.yaml
```

Conventions de nommage

```
metadata:
  name: webapp-frontend-deployment      # Descriptif et
explicite
  labels:
    app.kubernetes.io/name: webapp
    app.kubernetes.io/component: frontend
    app.kubernetes.io/instance: webapp-prod
```

Commentaires et documentation

```
# Deployment pour l'application web frontend
# Maintenu par l'équipe engineering
# Dernière mise à jour : 2024-01-15
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-frontend
  annotations:
    deployment.kubernetes.io/revision: "3"
    maintainer: "team-engineering@company.com"
    documentation: "https://wiki.company.com/webapp-frontend"
spec:
  replicas: 3 # Minimum pour la haute disponibilité
  # ... reste de la configuration
```

Sécurité dans les YAML

```
spec:
  template:
    spec:
      # Toujours définir un contexte de
      # sécurité
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
      containers:
        - name: app
          image: nginx:1.29.1-alpine #
          # Version spécifique, pas latest
          securityContext:
            allowPrivilegeEscalation:
              false
            readOnlyRootFilesystem: true
            capabilities:
              drop: ["ALL"]
```

```
      # Toujours définir des limites
      # de ressources
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
      # Ne jamais mettre de secrets en
      # dur
      env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: password
```

Important

En sécu, la configuration déclarative permet de :

- Auditer toutes les configurations
- Appliquer des politiques de sécurité automatiquement
- Détecter les dérives de configuration
- Garantir la conformité réglementaire

Dans le module suivant, nous verrons comment gérer les données persistantes et la configuration des applications avec les volumes, ConfigMaps et Secrets.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Dans Kubernetes, les conteneurs sont éphémères par design. Cela pose des défis pour :

- **Données applicatives** : bases de données, fichiers utilisateurs
- **Configuration** : fichiers de config, paramètres d'application
- **Logs** : journaux d'audit et de sécurité
- **Secrets** : mots de passe, certificats, clés API

Sans mécanisme de persistance, toutes ces données sont perdues à chaque redémarrage de Pod.

Types de volumes Kubernetes

Kubernetes propose plusieurs types de volumes selon les besoins :

emptyDir : stockage temporaire partagé

Utile pour partager des données entre conteneurs d'un même Pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: shared-storage-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.21
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: content-generator
    image: busybox
```

```
command:
- /bin/sh
- -c
- |
  while true; do
    echo "Mise à jour: $(date)" > /
data/index.html
    sleep 30
  done
volumeMounts:
- name: shared-data
  mountPath: /data
volumes:
- name: shared-data
  emptyDir: {}
```

Caractéristiques d'emptyDir :

- Créé quand le Pod démarre
- Détruit quand le Pod est supprimé
- Partagé entre tous les conteneurs du Pod
- Stocké sur le disque local du node

hostPath : accès au système de fichiers du node

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-pod
spec:
  containers:
  - name: log-reader
    image: busybox
    command: ["tail", "-f", "/logs/system.log"]
    volumeMounts:
    - name: host-logs
      mountPath: /logs
  volumes:
  - name: host-logs
    hostPath:
      path: /var/log
      type: Directory
```

Attention

`hostPath` présente des risques de sécurité car il donne accès au système de fichiers du node. À utiliser avec précaution et uniquement pour des cas spécifiques (monitoring, logging système).

Persistent Volumes (PV) et Persistent Volume Claims (PVC)

Pour un stockage persistant robuste, Kubernetes utilise le concept de PV/PVC :

- **PersistentVolume (PV)** : ressource de stockage dans le cluster
- **PersistentVolumeClaim (PVC)** : demande de stockage par un utilisateur.rice

Types de volumes Kubernetes

```
# pv-example.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce          # Un seul
                             # mode en lecture/écriture
  persistentVolumeReclaimPolicy:
    Retain
```

```
storageClassName: local-storage
local:
  path: /mnt/data
nodeAffinity:
  required:
    nodeSelectorTerms:
      - matchExpressions:
          - key: kubernetes.io/
            hostname
            operator: In
            values:
              - minikube
```

Types de volumes Kubernetes

```
# pvc-example.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: local-storage
```

Types de volumes Kubernetes

```
# pod-with-pvc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-with-storage
spec:
  containers:
  - name: app
    image: nginx:1.21
    volumeMounts:
    - name: data-storage
      mountPath: /var/www/html
  volumes:
  - name: data-storage
    persistentVolumeClaim:
      claimName: data-pvc
```

Storage Classes : provisioning dynamique

Les Storage Classes permettent le provisioning automatique de volumes :

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/no-provisioner
parameters:
  type: pd-ssd
  zones: us-central1-a, us-central1-b
reclaimPolicy: Delete
allowVolumeExpansion: true
```

Avec une Storage Class, pas besoin de créer les PV manuellement :

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-storage
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: fast-ssd      # Référence la Storage Class
  resources:
    requests:
      storage: 20Gi
```

Exercice pratique

Exercice 1 : Stockage persistant

1. Créez un PV local de 1Gi
2. Créez un PVC qui demande 512Mi
3. Déployez un Pod nginx qui utilise ce PVC
4. Créez un fichier dans le volume
5. Supprimez et recréez le Pod
6. Vérifiez que le fichier est toujours présent

Qu'est-ce qu'une ConfigMap ?

Concept K8s : ConfigMap

Une ConfigMap stocke des données de configuration non confidentielles sous forme de paires clé-valeur. Elle découple la configuration du code des applications.

Créer des ConfigMaps

Plusieurs façons de créer une ConfigMap :

```
# Depuis des valeurs littérales
```

```
$ kubectl create configmap app-config \
  --from-literal=database_url=postgresql://localhost:5432/mydb \
  --from-literal=debug_mode=true \
  --from-literal=max_connections=100
```

```
# Depuis un fichier
```

```
$ kubectl create configmap nginx-config --from-file=nginx.conf
```

```
# Depuis un dossier
```

```
$ kubectl create configmap web-config --from-file=./config-
files/
```

Ou de façon déclarative :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  labels:
    app: webapp
data:
  # Clé-valeur simples
  database_url: "postgresql://db:5432/
webapp"
  debug_mode: "true"
  max_connections: "100"

  # Fichier de configuration complet
  app.properties: |
```

```
# Configuration de l'application
server.port=8080
server.host=0.0.0.0

# Base de données
db.host=postgresql
db.port=5432
db.name=webapp

# Configuration nginx
nginx.conf: |
  server {
    listen 80;
    // config...
  }
```

Utiliser les ConfigMaps

Trois façons principales d'utiliser une ConfigMap :

1. Variables d'environnement

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: webapp:latest
    env:
      # Variable spécifique depuis ConfigMap
      - name: DATABASE_URL
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: database_url
      # Toutes les clés de la ConfigMap
    envFrom:
      - configMapRef:
          name: app-config
```

2. Fichiers montés

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.21
    volumeMounts:
    - name: config-volume
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf          # Monte seulement ce fichier
  volumes:
  - name: config-volume
    configMap:
      name: app-config
```

3. Arguments de commande

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: webapp:latest
    command:
    - "/bin/app"
    - "--config=/etc/config/
      app.properties"
    - "--port=$(SERVER_PORT)"
```

```
env:
- name: SERVER_PORT
  valueFrom:
    configMapKeyRef:
      name: app-config
      key: server_port
volumeMounts:
- name: config
  mountPath: /etc/config
volumes:
- name: config
  configMap:
    name: app-config
```

Qu'est-ce qu'un Secret ?

Concept K8s : Secret

Un Secret stocke et gère des informations sensibles comme des mots de passe, des tokens OAuth, des clés SSH ou des certificats TLS.

Qu'est-ce qu'un Secret ?

Concept K8s : Secret

Un Secret stocke et gère des informations sensibles comme des mots de passe, des tokens OAuth, des clés SSH ou des certificats TLS.

Attention

Les Secrets Kubernetes ne sont que de l'encodage base64, pas du chiffrement. En production, utilisez des outils comme Sealed Secrets, External Secrets Operator, ou Vault.

Types de Secrets

Secret générique (Opaque)

```
$ kubectl create secret generic db-credentials \  
  --from-literal=username=admin \  
  --from-literal=password=motdepasse123
```

Secret TLS

```
$ kubectl create secret tls webapp-tls \  
  --cert=webapp.crt \  
  --key=webapp.key
```

Secret Docker Registry

```
$ kubectl create secret docker-registry my-registry-secret \  
  --docker-server=registry.company.com \  
  --docker-username=user \  
  --docker-password=password \  
  --docker-email=user@company.com
```

Créer des Secrets déclarativement

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: YWRtaW4=          # echo -n 'admin' | base64
  password: bW90ZGVwYXNzZTEyMw== # echo -n 'motdepasse123' |
base64
```

Utiliser les Secrets

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: webapp:latest
          env:
            # Variable depuis Secret
            - name: DB_PASSWORD
```

```
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: password
      # Toutes les clés du Secret
      envFrom:
        - secretRef:
            name: db-credentials
      volumeMounts:
        # Secret monté comme fichiers
        - name: tls-certs
          mountPath: /etc/tls
          readOnly: true
      volumes:
        - name: tls-certs
          secret:
            secretName: webapp-tls
      # Utiliser un Secret pour Docker
      Registry
      imagePullSecrets:
        - name: my-registry-secret
```

Exercice pratique

Exercice 2 : ConfigMap et Secret

1. Créez une ConfigMap avec une configuration nginx
2. Créez un Secret avec des identifiants de base de données
3. Déployez nginx en utilisant la ConfigMap pour sa config
4. Déployez une application qui utilise le Secret via variables d'environnement
5. Vérifiez que les configurations sont bien appliquées

Différence avec les Deployments

Toutes les applications ne sont pas identiques :

Applications sans état (Stateless) :

- Les instances sont interchangeables
- Pas d'ordre de démarrage important
- Stockage partagé ou temporaire
- Exemples : serveurs web, APIs REST

Applications avec état (Stateful) :

- Chaque instance a une identité unique
- Ordre de démarrage/arrêt important
- Stockage persistant individuel
- Exemples : bases de données, systèmes distribués

Caractéristiques des StatefulSets

Concept K8s : StatefulSet

Un StatefulSet gère le déploiement et la mise à l'échelle d'un ensemble de Pods avec des garanties d'ordre et d'unicité. Idéal pour les applications qui nécessitent une identité réseau stable et un stockage persistant.

Caractéristiques des StatefulSets

Concept K8s : StatefulSet

Un StatefulSet gère le déploiement et la mise à l'échelle d'un ensemble de Pods avec des garanties d'ordre et d'unicité. Idéal pour les applications qui nécessitent une identité réseau stable et un stockage persistant.

Garanties fournies :

- Identité réseau stable : podname-0, podname-1, podname-2...
- Stockage persistant : chaque Pod a son propre volume
- Déploiement et mise à l'échelle ordonnés
- Mises à jour progressives contrôlées

Exemple : Cluster PostgreSQL

```
# postgres-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgres
spec:
  serviceName: postgres-headless # Service
  headless: true
  replicas: 3
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:13-alpine
          env:
            - name: POSTGRES_DB
              value: webapp
            - name: POSTGRES_USER
              value: webapp
            - name: POSTGRES_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: postgres-secret
                  key: password
            - name: PGDATA
              value: /var/lib/postgresql/data/pgdata
```

```
ports:
  - containerPort: 5432
volumeMounts:
  - name: postgres-storage
    mountPath: /var/lib/postgresql/data
# Health checks importants pour les bases de
données
livenessProbe:
  exec:
    command:
      - pg_isready
      - -U
      - webapp
    initialDelaySeconds: 30
    periodSeconds: 10
  readinessProbe:
    exec:
      command:
        - pg_isready
        - -U
        - webapp
      initialDelaySeconds: 5
      periodSeconds: 5
# Volume Claim Templates : un PVC par Pod
volumeClaimTemplates:
  - metadata:
      name: postgres-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Service Headless

Les StatefulSets utilisent souvent des Services « headless » :

```
# postgres-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: postgres-headless
spec:
  clusterIP: None          # Service
  headless
  selector:
    app: postgres
  ports:
    - port: 5432
      targetPort: 5432

---
```

```
# Service normal pour l'accès
externe
apiVersion: v1
kind: Service
metadata:
  name: postgres
spec:
  selector:
    app: postgres
  ports:
    - port: 5432
      targetPort: 5432
```

Le Service headless permet d'accéder à chaque Pod individuellement :

- postgres-0.postgres-headless.default.svc.cluster.local
- postgres-1.postgres-headless.default.svc.cluster.local
- postgres-2.postgres-headless.default.svc.cluster.local

Secret pour PostgreSQL

```
# postgres-secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: postgres-secret
type: Opaque
stringData:
  password: "SuperSecretPassword123!"
  replica-password: "ReplicaPassword456!"
```

Exercice pratique

Exercice 3 : StatefulSet PostgreSQL

1. Créez le Secret PostgreSQL
2. Déployez le StatefulSet PostgreSQL
3. Observez l'ordre de création des Pods (0, puis 1, puis 2)
4. Connectez-vous au premier Pod et créez une base de données
5. Supprimez le Pod postgres-0 et observez qu'il garde son nom
6. Vérifiez que les données sont persistantes
7. Testez la résolution DNS des Pods individuels

Modes d'accès

- **ReadWriteOnce (RWO)** : lecture/écriture par un seul node
- **ReadOnlyMany (ROX)** : lecture par plusieurs nodes
- **ReadWriteMany (RWX)** : lecture/écriture par plusieurs nodes

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-storage
spec:
  accessModes:
    - ReadWriteMany          # Nécessite un stockage supportant
RWX
  resources:
    requests:
      storage: 50Gi
  storageClassName: nfs-client
```

Policies de réclamation

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: important-data-pv
spec:
  capacity:
    storage: 100Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain      # Garde les données
  # persistentVolumeReclaimPolicy: Delete   # Supprime les
données
  # persistentVolumeReclaimPolicy: Recycle  # Nettoie et
réutilise
```

Expansion de volumes

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: expandable-storage
provisioner: kubernetes.io/aws-ebs
allowVolumeExpansion: true    # Permet l'expansion
parameters:
  type: gp2

# Agrandir un PVC existant
$ kubectl patch pvc data-pvc -p '{"spec":{"resources":{"requests":{"storage":"20Gi"}}}}'
```

Chiffrement au repos

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: encrypted-storage
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  encrypted: "true"      # Chiffrement EBS
  kmsKeyId: "arn:aws:kms:us-
east-1:123456789012:key/12345678-1234-1234-1234-123456789012"
```

Limiter l'accès aux volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-app
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000      #
  # Propriétaire des volumes
  containers:
  - name: app
    image: webapp:latest
```

```
securityContext:
  readOnlyRootFilesystem: true
volumeMounts:
- name: data
  mountPath: /data
- name: tmp
  mountPath: /tmp
volumes:
- name: data
  persistentVolumeClaim:
    claimName: app-data
- name: tmp
  emptyDir: {}
```

Rotation des Secrets

```
# Script de rotation automatisée
$ kubectl create secret generic db-credentials-new \
  --from-literal=username=admin \
  --from-literal=password=$(openssl rand -base64 32)

# Mise à jour progressive
$ kubectl set env deployment/webapp --from=secret/db-
credentials-new

# Suppression de l'ancien secret après validation
$ kubectl delete secret db-credentials-old
```

Important

En sécu, la gestion des données sensibles doit inclure :

- Chiffrement au repos et en transit
- Rotation régulière des secrets
- Limitation des accès selon le principe du moindre privilège
- Audit et traçabilité de tous les accès
- Sauvegarde et plan de récupération

Dans le module suivant, nous explorerons le networking Kubernetes et comment exposer nos applications de manière sécurisée.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

Principes fondamentaux

Le modèle réseau Kubernetes repose sur quatre règles simples :

1. **Chaque Pod a sa propre adresse IP** (pas de NAT entre Pods)
2. **Tous les Pods peuvent communiquer** entre eux sans NAT
3. **Tous les nœuds peuvent communiquer** avec tous les Pods sans NAT
4. **L'IP vue par un Pod** est la même que celle vue par les autres

Cette simplicité conceptuelle cache une complexité d'implémentation gérée par les CNI (Container Network Interface).

Container Network Interface (CNI)

Le CNI est un standard qui définit comment les plugins réseau s'intègrent avec Kubernetes :

- **Flannel** : overlay réseau simple avec VXLAN
- **Calico** : réseau L3 avec politiques de sécurité avancées
- **Weave** : mesh network automatique
- **Cilium** : networking et sécurité basés sur eBPF
- **Canal** : combinaison Flannel + Calico

```
# Voir le plugin CNI utilisé
$ kubectl get nodes -o wide
$ kubectl describe node minikube | grep -i cni

# Dans minikube, généralement kindnet ou bridge
```

Communication Pod-to-Pod

```
# Exemple de deux Pods qui
communiquent
apiVersion: v1
kind: Pod
metadata:
  name: client-pod
  labels:
    app: client
spec:
  containers:
  - name: client
    image: busybox
    command: ["sleep", "3600"]
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: server-pod
  labels:
    app: server
spec:
  containers:
  - name: server
    image: nginx:1.21
    ports:
    - containerPort: 80
```

Test de communication :

```
$ kubectl get pod server-pod -o wide
```

```
$ kubectl exec -it client-pod -- wget -qO- <IP-du-server>:80
```

Note

Cette communication directe Pod-to-Pod n'est généralement pas utilisée en production. On préfère passer par des Services pour l'abstraction et la résilience.

Le problème sans Services

Les Pods sont éphémères :

- Leur IP change à chaque redémarrage
- Les clients ne peuvent pas s'appuyer sur une IP fixe
- Comment faire du load balancing entre plusieurs instances ?
- Comment découvrir les services disponibles ?

Solution : les Services Kubernetes

Concept K8s : Service

Un Service Kubernetes expose un ensemble de Pods sous une adresse IP stable avec découverte de service automatique et équilibrage de charge.

Types de Services

ClusterIP : accès interne uniquement

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-internal
spec:
  type: ClusterIP          # Défaut si non spécifié
  selector:
    app: webapp
  ports:
    - name: http
      port: 80              # Port du Service
      targetPort: 8080      # Port du conteneur
      protocol: TCP
```

Caractéristiques :

- IP virtuelle accessible seulement depuis le cluster
- DNS automatique : `webapp-internal.default.svc.cluster.local`
- Équilibrage de charge entre les Pods sélectionnés

NodePort : accès externe via les nœuds

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-external
spec:
  type: NodePort
  selector:
    app: webapp
  ports:
    - name: http
      port: 80
      targetPort: 8080
      nodePort: 32080          # Port sur chaque nœud
(30000-32767)
```

Accessible via : <NodeIP>:32080 depuis l'extérieur.

```
# Avec minikube, obtenir l'URL d'accès  
minikube service webapp-external --url
```

LoadBalancer : intégration cloud

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-lb
spec:
  type: LoadBalancer
  selector:
    app: webapp
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

Le fournisseur cloud (AWS, GCP, Azure) crée automatiquement un load balancer externe.

ExternalName : alias DNS

```
apiVersion: v1
kind: Service
metadata:
  name: external-db
spec:
  type: ExternalName
  externalName: database.company.com
```

Permet d'accéder à `external-db.default.svc.cluster.local` qui redirige vers `database.company.com`.

Service Discovery et DNS

Kubernetes fournit un DNS automatique pour tous les Services :

```
# Format des noms DNS  
<service-name>.<namespace>.svc.cluster.local  
  
# Exemples  
webapp.default.svc.cluster.local  
database.production.svc.cluster.local
```

```
# Test de résolution DNS depuis un Pod
$ kubectl exec -it client-pod -- nslookup webapp-internal
$ kubectl exec -it client-pod -- dig webapp-internal.default.svc.cluster.local
```

Exercice pratique

Exercice 1 : Types de Services

1. Créez un Deployment nginx avec 3 répliques
2. Créez un Service ClusterIP pour l'exposer
3. Testez l'accès depuis un Pod client
4. Changez le type vers NodePort
5. Testez l'accès depuis l'extérieur du cluster
6. Observez l'équilibrage de charge entre les Pods

Limitations des Services

Les Services NodePort et LoadBalancer ont des inconvénients :

- Un port/LoadBalancer par service (coûteux)
- Pas de routage basé sur le hostname ou le path
- Pas de terminaison TLS centralisée
- Pas de réécriture d'URL ou redirection

Qu'est-ce qu'un Ingress ?

Concept K8s : Ingress

Un Ingress expose des routes HTTP/HTTPS de l'extérieur du cluster vers des Services à l'intérieur du cluster. Il fournit un routage basé sur les règles de nom d'hôte et de chemin.

Architecture Ingress :

Internet



Ingress Controller (nginx, traefik, istio...)



Ingress Rules



Services → Pods

Installer un Ingress Controller

```
# Avec minikube, activer l'addon  
minikube addons enable ingress
```

```
# Vérifier que le contrôleur est déployé  
$ kubectl get pods -n ingress-nginx
```

Exemple d'Ingress simple

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

Configuration DNS locale (pour les tests) :

```
# Ajouter à /etc/hosts  
echo "$(minikube ip) webapp.local" | sudo tee -a /etc/hosts
```

Ingress avec routage avancé

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: multi-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  tls:
  - hosts:
    - webapp.company.com
    - api.company.com
    secretName: company-tls-cert
  rules:
    # Application web frontend
    - host: webapp.company.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
    # API backend
    - host: api.company.com
      http:
        paths:
          - path: /v1
            pathType: Prefix
            backend:
              service:
                name: api-v1-service
```

```
      port:
        number: 80
    - path: /v2
      pathType: Prefix
      backend:
        service:
          name: api-v2-service
          port:
            number: 80
    # Routage par path sur le même domaine
    - host: company.com
      http:
        paths:
          - path: /app
            pathType: Prefix
            backend:
              service:
                name: webapp-service
                port:
                  number: 80
          - path: /blog
            pathType: Prefix
            backend:
              service:
                name: blog-service
                port:
                  number: 80
          - path: /admin
            pathType: Prefix
            backend:
              service:
                name: admin-service
                port:
                  number: 80
```

Terminaison TLS avec Ingress

```
# Créer un certificat auto-signé pour les tests
apiVersion: v1
kind: Secret
metadata:
  name: webapp-tls-secret
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUdJTi... # Base64 du certificat
  tls.key: LS0tLS1CRUdJTi... # Base64 de la clé privée
```

```
# Générer un certificat auto-signé
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout webapp.key -out webapp.crt \
  -subj "/CN=webapp.local"

# Créer le Secret
$ kubectl create secret tls webapp-tls-secret \
  --cert=webapp.crt --key=webapp.key
```

Ingress : routage HTTP

```
# Ingress avec TLS
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: webapp-tls-ingress
spec:
  tls:
  - hosts:
    - webapp.local
    secretName: webapp-tls-secret
  rules:
  - host: webapp.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: webapp-service
            port:
              number: 80
```

Annotations communes pour nginx-ingress

```
metadata:
  annotations:
    # Réécriture d'URL
    nginx.ingress.kubernetes.io/rewrite-target: /

    # Redirection HTTPS
    nginx.ingress.kubernetes.io/ssl-redirect: "true"

    # CORS
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"

    # Limite de taille
    nginx.ingress.kubernetes.io/proxy-body-size: "10m"

    # Timeout
    nginx.ingress.kubernetes.io/proxy-read-timeout: "300"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "300"

    # Authentification basique
    nginx.ingress.kubernetes.io/auth-type: basic
    nginx.ingress.kubernetes.io/auth-secret: basic-auth

    # Rate limiting
    nginx.ingress.kubernetes.io/rate-limit: "100"
    nginx.ingress.kubernetes.io/rate-limit-window: "1m"

    # Whitelist d'IP
    nginx.ingress.kubernetes.io/whitelist-source-range:
      "10.0.0.0/8,192.168.0.0/16"
```

Exercice pratique

Exercice 2 : Ingress

1. Créez deux applications différentes (nginx et apache)
2. Exposez-les via des Services ClusterIP
3. Créez un Ingress qui route selon le path (/nginx, /apache)
4. Testez l'accès aux deux applications
5. Ajoutez un certificat TLS auto-signé
6. Configurez la redirection HTTPS

Problème de sécurité par défaut

Par défaut, Kubernetes autorise toute communication :

- Tous les Pods peuvent parler à tous les autres Pods
- Aucun firewall interne
- Modèle « flat network » sans segmentation

En production, c'est un risque de sécurité qu'il faut maîtriser !

Qu'est-ce qu'une Network Policy ?

Concept K8s : Network Policy

Une Network Policy définit des règles de communication réseau pour un ensemble de Pods, implémentant une segmentation micro-réseau basée sur des labels.

Important

Les Network Policies nécessitent un plugin CNI qui les supporte : Calico, Cilium, Weave Net. Le plugin par défaut de minikube (kindnet) ne les supporte PAS.

Installation de Calico sur minikube

```
# Démarrer minikube avec Calico  
minikube start --cni=calico
```

```
# Ou installer Calico sur un cluster existant  
$ kubectl apply -f https://docs.projectcalico.org/v3.20/manifests/calico.yaml
```

Politique « Deny All » par défaut

```
# deny-all-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: default
spec:
  podSelector: {}           # Sélectionne tous les Pods
  policyTypes:
  - Ingress                 # Bloque tout trafic entrant
```

```
# deny-all-egress.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-egress
  namespace: default
spec:
  podSelector: {}
  policyTypes:
    - Egress                                # Bloque tout trafic sortant
```

Autoriser des communications spécifiques

```
# allow-frontend-to-backend.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
spec:
  podSelector:
    matchLabels:
      role: backend          # S'applique aux Pods "backend"
  policyTypes:
    - Ingress
  ingress:
    - from:
      - podSelector:
          matchLabels:
            role: frontend    # Autorise depuis les Pods
                              "frontend"
      ports:
        - protocol: TCP
          port: 8080
```

Politique complète multi-tiers

```
# Application 3-tiers sécurisée
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: webapp-network-policy
spec:
  podSelector:
    matchLabels:
      app: webapp
  policyTypes:
  - Ingress
  - Egress
  ingress:
    # Autoriser trafic depuis l'ingress controller
    - from:
      - namespaceSelector:
          matchLabels:
            name: ingress-nginx
      ports:
      - protocol: TCP
        port: 80
  egress:
    # Autoriser accès à la base de données
    - to:
      - podSelector:
          matchLabels:
            app: database
```

```
    ports:
      - protocol: TCP
        port: 5432
    # Autoriser DNS (nécessaire pour la résolution de
noms)
    - to: []
      ports:
      - protocol: UDP
        port: 53
  ---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-network-policy
spec:
  podSelector:
    matchLabels:
      app: database
  policyTypes:
  - Ingress
  ingress:
    # Autoriser accès seulement depuis webapp
    - from:
      - podSelector:
          matchLabels:
            app: webapp
      ports:
      - protocol: TCP
        port: 5432
```

Autoriser communications entre namespaces

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-monitoring
  namespace: production
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: monitoring      # Depuis le namespace monitoring
  ports:
  - protocol: TCP
    port: 9090                  # Port Prometheus
```

Debugging des Network Policies

Lister toutes les Network Policies

\$ kubectl get networkpolicies --all-namespaces

Voir les détails d'une policy

\$ kubectl describe networkpolicy deny-all-ingress

Tester la connectivité

\$ kubectl exec -it frontend-pod -- wget -qO- backend-service:8080

Voir les logs Calico (si utilisé)

\$ kubectl logs -n kube-system -l k8s-app=calico-node

Exercice pratique

Exercice 3 : Segmentation réseau

1. Installez Calico ou configurez un cluster qui supporte les Network Policies
2. Déployez une application frontend et une base de données
3. Testez que la communication fonctionne sans policies
4. Appliquez une politique « deny-all »
5. Vérifiez que la communication est bloquée
6. Créez des policies pour autoriser seulement les communications légitimes
7. Testez que seules les communications autorisées fonctionnent

Surveiller le trafic réseau

```
# Statistiques réseau des Pods
$ kubectl top pods --sort-by=memory
$ kubectl top nodes

# Voir les connexions réseau d'un Pod
$ kubectl exec -it mon-pod -- netstat -tulpn
$ kubectl exec -it mon-pod -- ss -tulpn
```

Outils de debugging réseau

```
# Pod de debugging réseau
apiVersion: v1
kind: Pod
metadata:
  name: netshoot
spec:
  containers:
  - name: netshoot
    image: nicolaka/netshoot
    command: ["sleep", "3600"]
```

Outils disponibles dans netshoot :

```
$ kubectl exec -it netshoot -- nslookup service-name
$ kubectl exec -it netshoot -- dig service-
name.namespace.svc.cluster.local
$ kubectl exec -it netshoot -- curl -I http://service-name
$ kubectl exec -it netshoot -- tcpdump -i eth0
$ kubectl exec -it netshoot -- nmap -p 80,443 service-name
```

Principe de moindre privilège

1. **Deny by default** : commencer par bloquer tout
2. **Autoriser explicitement** : n'ouvrir que ce qui est nécessaire
3. **Segmentation** : séparer les environnements et applications
4. **Monitoring** : surveiller les communications anormales

Important

En production :

- Toujours implémenter des Network Policies restrictives
- Chiffrer le trafic sensible inter-services (service mesh)
- Monitorer les communications pour détecter les anomalies
- Séparer les réseaux par environnement (dev, staging, prod)
- Documenter et revoir régulièrement les règles de communication

Dans le module suivant, nous approfondirons les aspects de sécurité de Kubernetes : RBAC, Pod Security, et autres mécanismes de protection pour des déploiements robustes.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

La sécurité dans Kubernetes suit le principe de « défense en profondeur » avec plusieurs couches de protection. Un cluster Kubernetes mal sécurisé peut exposer des données sensibles ou permettre des accès non autorisés.

Pourquoi la sécurité K8s est critique

- Les clusters exposent souvent des applications sensibles
- Un Pod compromis peut potentiellement accéder à d'autres ressources
- Les configurations par défaut ne sont pas toujours sécurisées
- La surface d'attaque est large (API server, etcd, nœuds, réseau...)

Modèle de sécurité Kubernetes

Kubernetes implémente plusieurs mécanismes de sécurité :

1. **Authentication** : Qui peut accéder au cluster ?
2. **Authorization** : Que peuvent faire les utilisateurs authentifiés ?
3. **Admission Control** : Validation et modification des requêtes
4. **Pod Security** : Restrictions sur les capacités des Pods
5. **Network Security** : Isolation réseau entre les workloads
6. **Secrets Management** : Protection des données sensibles

Concept K8s : Security Context

Un Security Context définit les privilèges et paramètres de sécurité pour un Pod ou conteneur individuel.

Configuration au niveau Pod

Configuration YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000          # UID
  utilisateur
    runAsGroup: 3000        # GID groupe
    fsGroup: 2000          # Groupe
  propriétaire des volumes
    fsGroupChangePolicy: Always
    runAsNonRoot: true     # Interdire
  root
```

```
containers:
- name: sec-ctx-demo
  image: busybox
  command: ["sh", "-c", "sleep 1h"]
  securityContext:
    allowPrivilegeEscalation: false
    readOnlyRootFilesystem: true
    capabilities:
      drop:
        - ALL              # Supprimer
  toutes les capacités
      add:
        - NET_BIND_SERVICE # Autoriser bind
  sur ports < 1024
```

Bonnes pratiques Security Context

Important

Toujours appliquer ces règles de sécurité :

- `runAsNonRoot: true` - jamais en root
- `allowPrivilegeEscalation: false` - pas d'escalade
- `readOnlyRootFilesystem: true` - système de fichiers en lecture seule
- `capabilities: drop: [ALL]` - supprimer toutes les capabilities

Exemple d'application sécurisée

Configuration YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: secure-nginx
  template:
    metadata:
      labels:
        app: secure-nginx
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 101
        runAsGroup: 101
        fsGroup: 101
      containers:
        - name: nginx
          image: nginx:1.21
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop:
                - ALL
              add:
                - NET_BIND_SERVICE
          volumeMounts:
            - name: tmp-volume
              mountPath: /tmp
            - name: var-cache-nginx
              mountPath: /var/cache/nginx
            - name: var-run
              mountPath: /var/run
          volumes:
            - name: tmp-volume
              emptyDir: {}
            - name: var-cache-nginx
              emptyDir: {}
            - name: var-run
              emptyDir: {}
```

Remarque

Nginx a besoin de répertoires temporaires en écriture. On les monte comme volumes emptyDir pour respecter `readOnlyRootFilesystem: true`.

Concept K8s : Pod Security Standards

Les Pod Security Standards remplacent les Pod Security Policies et définissent trois profils de sécurité : restricted, baseline, et privileged.

Les trois profils

1. **Privileged** : Pas de restrictions (équivalent au comportement par défaut)
2. **Baseline** : Restrictions minimales, empêche les escalades connues
3. **Restricted** : Restrictions strictes selon les bonnes pratiques

Configuration par namespace

Configuration YAML

```
apiVersion: v1
kind: Namespace
metadata:
  name: secure-apps
  labels:
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/warn: restricted
```

Profil Baseline - Restrictions

Le profil baseline interdit :

- Conteneurs privilégiés
- Ajout de capabilities dangereuses
- Accès à hostPath, hostNetwork, hostPID
- Sysctls dangereux
- SELinux custom
- AppArmor overrides

Configuration YAML

```
# Ce Pod sera REJETÉ en mode baseline
apiVersion: v1
kind: Pod
metadata:
  name: privileged-pod
spec:
  containers:
  - name: nginx
    image: nginx
    securityContext:
      privileged: true      # INTERDIT en baseline
```

Profil Restricted - Stricte sécurité

Le profil restricted ajoute les restrictions :

- Doit tourner en non-root
- Pas d'escalade de privilège
- Capabilities limitées
- SeccompProfile obligatoire

Configuration YAML

Pod Security Standards

```
# Pod conforme au profil restricted
apiVersion: v1
kind: Pod
metadata:
  name: restricted-pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: nginx
    image: nginx:1.21
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
      readOnlyRootFilesystem: true
```

Exercice pratique

Exercice : Appliquer Pod Security Standards

1. Créez un namespace test-security avec le profil restricted
2. Tentez de déployer un Pod non conforme (avec `privileged: true`)
3. Modifiez le Pod pour le rendre conforme au profil restricted
4. Vérifiez qu'il se déploie correctement

Concept K8s : ServiceAccount

Un ServiceAccount fournit une identité pour les processus qui s'exécutent dans un Pod.

ServiceAccount par défaut

Chaque namespace a un ServiceAccount default automatiquement créé :

Commande

```
$ kubectl get serviceaccounts -A  
$ kubectl describe serviceaccount default
```

Création d'un ServiceAccount personnalisé

Configuration YAML

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: monitoring-sa
  namespace: monitoring
automountServiceAccountToken: false # Sécurité : pas de montage auto
---
apiVersion: v1
kind: Secret
metadata:
  name: monitoring-sa-secret
  namespace: monitoring
  annotations:
    kubernetes.io/service-account.name: monitoring-sa
type: kubernetes.io/service-account-token
```

Utilisation dans un Pod

Configuration YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: monitoring-pod
spec:
  serviceAccountName: monitoring-sa
  containers:
  - name: prometheus
    image: prom/prometheus:latest
  automountServiceAccountToken: true
```

Remarque

Par défaut, Kubernetes monte automatiquement le token du ServiceAccount. Pour plus de sécurité, désactivez le montage automatique et montez explicitement quand nécessaire.

Concept K8s : RBAC

RBAC permet de définir qui peut faire quoi sur quelles ressources dans Kubernetes.

Concepts RBAC

- **Role/ClusterRole** : Définit un ensemble de permissions
- **RoleBinding/ClusterRoleBinding** : Attache un Role à des utilisateurs/SA
- **Subject** : Utilisateur, groupe ou ServiceAccount

Différence Role vs ClusterRole

- **Role** : Permissions dans UN namespace spécifique
- **ClusterRole** : Permissions au niveau cluster ou multi-namespaces

Création d'un Role

Configuration YAML

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

Création d'un RoleBinding

Configuration YAML

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

Exemple complet - ServiceAccount pour monitoring

Configuration YAML

```
# ServiceAccount
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus-sa
  namespace: monitoring
---
# ClusterRole pour lire les métriques
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus-reader
rules:
- apiGroups: [""]
  resources: ["nodes", "services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["extensions", "apps"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "list", "watch"]
---
# ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus-reader
subjects:
- kind: ServiceAccount
  name: prometheus-sa
  namespace: monitoring
```

Vérification des permissions

Commande

```
# Vérifier si un SA peut faire une action
$ kubectl auth can-i get pods --
as=system:serviceaccount:monitoring:prometheus-sa

# Lister toutes les permissions d'un SA
$ kubectl auth can-i --list --
as=system:serviceaccount:monitoring:prometheus-sa
```

Exercice pratique

Exercice : Configuration RBAC pour développeurs

1. Créez un namespace development
2. Créez un ServiceAccount developer-sa
3. Créez un Role permettant de gérer Pods, Services et ConfigMaps
4. Créez un RoleBinding pour associer le Role au ServiceAccount
5. Testez les permissions avec `$ kubectl auth can-i`

Concept K8s : Network Policy

Les Network Policies contrôlent le trafic réseau entre les Pods en définissant des règles d'ingress et egress.

Important

Les Network Policies nécessitent un CNI compatible (Calico, Cilium, Weave...). Elles ne fonctionnent pas avec tous les réseaux Kubernetes.

Politique par défaut - Tout refuser

Configuration YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: production
spec:
  podSelector: {}          # Sélectionne tous les Pods
  policyTypes:
    - Ingress
    - Egress
  # Pas de règles allow = tout est refusé
```

Autoriser seulement le trafic nécessaire

Configuration YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-web-to-api
  namespace: production
spec:
  podSelector:
    matchLabels:
      tier: api          # S'applique aux Pods API
  policyTypes:
    - Ingress
  ingress:
    - from:
      - podSelector:
          matchLabels:
            tier: web      # Autorise depuis les Pods web
    ports:
      - protocol: TCP
        port: 8080
```

Isolation complète d'un namespace

Configuration YAML

```
# Empêche tout trafic entrant depuis d'autres namespaces
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: namespace-isolation
  namespace: secure-zone
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: secure-zone      # Seulement depuis le même namespace
```

Exemple pour application 3-tiers

Configuration YAML

```
# Politique pour la base de données
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: database-policy
spec:
  podSelector:
    matchLabels:
      tier: database
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: api
    ports:
    - protocol: TCP
      port: 5432
  egress:
  - to: [] # DNS uniquement
    ports:
    - protocol: UDP
      port: 53
  ---
# Politique pour l'API
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
metadata:
  name: api-policy
spec:
  podSelector:
    matchLabels:
      tier: api
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          tier: web
    ports:
    - protocol: TCP
      port: 8080
  egress:
  - to:
    - podSelector:
        matchLabels:
          tier: database
    ports:
    - protocol: TCP
      port: 5432
  - to: [] # DNS
    ports:
    - protocol: UDP
      port: 53
```

Concept K8s : Secret

Les Secrets stockent des données sensibles comme des mots de passe, tokens, clés SSH de manière sécurisée.

Concept K8s : Secret

Les Secrets stockent des données sensibles comme des mots de passe, tokens, clés SSH de manière sécurisée.

Types de Secrets

- Opaque : données arbitraires (défaut)
- `kubernetes.io/dockerconfigjson` : authentication registry Docker
- `kubernetes.io/tls` : certificats TLS
- `kubernetes.io/service-account-token` : tokens ServiceAccount

Création d'un Secret

Commande

```
# Depuis la ligne de commande
$ kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=secretpassword123

# Depuis un fichier
echo -n "admin" > username.txt
echo -n "secretpassword123" > password.txt
$ kubectl create secret generic db-secret \
  --from-file=username=username.txt \
  --from-file=password=password.txt
```

Secret YAML (base64 encodé)

Configuration YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=          # base64 de "admin"
  password: c2VjcmV0cGFzc3dvcmQxMjM= # base64 de "secretpassword123"
```

Attention

Base64 n'est PAS du chiffrement ! Les Secrets sont stockés en base64 dans etcd.
Utilisez le chiffrement etcd en production.

Utilisation dans un Pod

Configuration YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: app-with-secret
spec:
  containers:
  - name: app
    image: nginx
    env:
      # Méthode 1: Variables d'environnement
      - name: DB_USERNAME
        valueFrom:
          secretKeyRef:
            name: db-secret
            key: username
      - name: DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: db-secret
            key: password
    volumeMounts:
      # Méthode 2: Montage en fichiers
      - name: secret-volume
        mountPath: "/etc/secrets"
        readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: db-secret
```

Secret pour Docker Registry

Configuration YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: regcred
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: <base64-encoded-docker-config>
---
# Utilisation dans un Pod
apiVersion: v1
kind: Pod
metadata:
  name: private-image-pod
spec:
  imagePullSecrets:
    - name: regcred
  containers:
    - name: app
      image: private-registry.com/myapp:latest
```

Commande

```
# Création d'un secret registry
$ kubectl create secret docker-registry regcred \
  --docker-server=private-registry.com \
  --docker-username=myuser \
  --docker-password=mypassword \
  --docker-email=myuser@company.com
```

Bonnes pratiques Secrets

Important

Sécurisation des Secrets :

- Activez le chiffrement etcd au repos
- Utilisez des outils externes (HashiCorp Vault, AWS Secrets Manager)
- Limitez l'accès via RBAC
- Auditez l'accès aux Secrets
- Rotation régulière des secrets
- Évitez les variables d'environnement (visibles dans les processus)

Concept K8s : Admission Controllers

Les Admission Controllers sont des plugins qui interceptent les requêtes vers l'API Server pour les valider ou les modifier.

Admission Controllers par défaut

Contrôleurs activés par défaut (non exhaustif) :

- `NamespaceLifecycle` : empêche la création d'objets dans des namespaces en suppression
- `LimitRanger` : applique les `LimitRanges`
- `ResourceQuota` : applique les quotas de ressources
- `PodSecurityPolicy` : applique les politiques de sécurité (déprécié)
- `NodeRestriction` : limite les permissions des kubelets

Validating Admission Webhooks

Exemple d'un webhook qui valide que tous les Pods ont des resource limits :

Configuration YAML

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingAdmissionWebhook
metadata:
  name: require-resource-limits
webhooks:
- name: validate-resources.example.com
  clientConfig:
    service:
      name: validation-webhook
      namespace: webhook-system
      path: "/validate"
  rules:
  - operations: ["CREATE", "UPDATE"]
    apiGroups: [""]
    apiVersions: ["v1"]
    resources: ["pods"]
  admissionReviewVersions: ["v1"]
```

Open Policy Agent (OPA) Gatekeeper

OPA Gatekeeper est un contrôleur d'admission populaire basé sur OPA :

Commande

```
# Installation de Gatekeeper
$ kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.14/deploy/gatekeeper.yaml
```

Configuration YAML

```
# Politique : tous les Pods doivent avoir des
labels requis
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8srequiredlabels
spec:
  crd:
    spec:
      names:
        kind: K8sRequiredLabels
      validation:
        type: object
        properties:
          labels:
            type: array
            items:
              type: string
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8srequiredlabels
```

```
violation[{"msg": msg}] {
  required := input.parameters.labels
  provided :=
input.review.object.metadata.labels
  missing := required[_]
  not provided[missing]
  msg := sprintf("Missing required label:
%v", [missing])
}
---
# Application de la politique
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sRequiredLabels
metadata:
  name: must-have-environment
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
  parameters:
    labels: ["environment", "owner"]
```

Image Scanning

Important

Scannez toujours vos images de conteneur pour détecter les vulnérabilités avant le déploiement.

Outils populaires de scanning :

- trivy : scanner local de Aqua Security
- gype : scanner d'Anchore
- snyk : service commercial
- clair : scanner CoreOS

Commande

```
# Scan avec Trivy
trivy image nginx:1.21
trivy image --severity HIGH,CRITICAL nginx:1.21

# Scan avec Gype
gype nginx:1.21
```

Admission Controller pour images

Configuration YAML

```
# Politique OPA : interdire les images sans tag ou
avec :latest
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8simagetagging
spec:
  crd:
    spec:
      names:
        kind: K8sImageTagging
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8simagetagging

        violation[{"msg": msg}] {
          container :=
input.review.object.spec.containers[_]
          image := container.image
          not contains(image, ":")
          msg := sprintf("Image '%v' must have a
tag", [image])
```

```
        }

        violation[{"msg": msg}] {
          container :=
input.review.object.spec.containers[_]
          image := container.image
          endsWith(image, ":latest")
          msg := sprintf("Image '%v' cannot
use :latest tag", [image])
        }

---
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sImageTagging
metadata:
  name: image-must-have-specific-tag
spec:
  match:
    kinds:
      - apiGroups: ["apps"]
        kinds: ["Deployment"]
      - apiGroups: [""]
        kinds: ["Pod"]
```

Image Pull Policies

Configuration YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-app
spec:
  template:
    spec:
      containers:
      - name: app
        image: myregistry.com/myapp:v1.2.3
        imagePullPolicy: Always      # Toujours vérifier les mises à
jour                                jour
                                    # ou IfNotPresent (défaut)
                                    # ou Never
```

Exercice pratique

Exercice : Pipeline de sécurité complet

1. Créez un namespace avec Pod Security Standard restricted
2. Configurez Network Policies pour isoler le namespace
3. Créez un ServiceAccount avec permissions RBAC minimales
4. Déployez une application avec Security Context strict
5. Testez que l'application fonctionne malgré les restrictions
6. Vérifiez qu'un Pod non-conforme est rejeté

Remarque

La sécurité Kubernetes est un domaine en évolution constante. Restez informé des dernières recommandations via les guides CIS Kubernetes Benchmark et la documentation officielle.

Ce module couvre les aspects essentiels de la sécurité Kubernetes.

Présentation

Introduction à Kubernetes

Premiers pas avec Kubernetes

Gestion déclarative avec YAML

Gestion des données et configuration

Networking et exposition des services

Sécurité dans Kubernetes

Licence

© Hugo Blanc, 2025

Ce document peut être distribué librement, selon les termes de la version 4.0 de la licence Creative Commons Attribution-ShareAlike: <http://creativecommons.org/licenses/by-sa/4.0/>.

Vous êtes libres de reproduire, distribuer et communiquer ce document au public et de modifier ce document, selon les conditions suivantes :

- **Paternité.** Vous devez citer le nom de l’auteur original.
- **Partage des Conditions Initiales à l’Identique.** Si vous modifiez, transformez ou adaptez cette création, vous n’avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.
- A chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les conditions contractuelles de mise à disposition de cette création. Chacune de ces conditions peut être levée si vous obtenez l’autorisation du titulaire des droits.